A4: Computer Networking (II)

Computer Systems 2024-25
Department of Computer Science
University of Copenhagen

David Marchant

Due: Sunday, 1st of December, 16:00
Version 2 (November 28, 2024)

This is the fifth assignment in the course on Computer Systems 2022 at DIKU and the second on the topic of Computer Networks. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment belongs to the CN category together with A3. Resubmission is not possible.

It is important to note that only solving the theoretical part will result in 0 points. Thus, the theoretical part can improve you score, but you *have* to give a shot at the programming task.

The web is more a social creation than a technical one. I designed it for a social effect — to help people work together — and not as a technical toy. The ultimate goal of the Web is to support and improve our weblike existence in the world. We clump into families, associations, and companies. We develop trust across the miles and distrust around the corner.

— Tim Berners-Lee, Weaving the Web (1999)

Overview

This assignment has two parts, namely a theoretical part (Section 1) and a programming part (Section 2). The theoretical part deals with questions that have been covered by the lectures. The programming part requires you to complete a file transfer service using socket programming in C. This assignment uses a similar, but not identical, protocol to A3 in order to transfer files according to a peer-to-peer architecture. In this assignment, the programming effort relies on building a peer consisting of concurrent client and server interactions. More details will follow in the programming part (Section 2).

1 Theoretical Part (25%)

You should also hand-in answers to the following questions. These questions are each quite open ended, and unless otherwise noted should take only a paragraph or two to answer.

1.1 Reliable Communication

This course has often presented as though all messages are sent either TCP or UDP. This is largely true, but there has been a trend in recent years of creating TCP alternatives, on top of UDP. These are something similar to the protocols in A3 and A4, e.g. application layer protocols that try to create some semblance of reliable communication manually, without the overhead of TCP. What must these protocols implement, and how would you expect them to achieve reliable communication using the fundamentally unreliable UDP.

1.2 DNS

Breaking news! The island nation of Scotmark has been noticed for the first time, located in the middle of the North Sea (See satelite image below for proof). The nation is conviently very culturally and technologically similar to Denmark, and is already running servers hosting websites to attract tourists. Unfortunatly the nation of Scotmark has not discovered DNS, and so any network addresses aren't discoverable without users manually typing IP addresses. Assign the country a suitable top-level domain, and describe what further steps should be taken so that the wider internet can access all Scotmarks websites from their human-memorable URLs. You should assume in your answer that numerous large companies, universities, government organisations etc already exist in Scotmark.

Figure 1: A satelite view of beautiful Scotmark.

1.3 Deadlock

There are two common computer science problems both known as 'The Santa Claus Problem'. The first deals with routing around locations and is also known as the traveling salesman, but the second is to do with concurrency and is defined in this paper. Pseudo-code for a proposed solution is found below, your task here is to assess if the system will deadlock or not. You should provide diagrams or other notes, showing how you have identified potential deadlocks. You should then reason about if this is a genuine potential deadlock or not. Note that as there are several different components at the same time, depending on how you approach this problem, you *may* find it easier to examine certain combinations of components rather than the whole system. Additionally, note that you do not need to comment on the correctness of the solution, only its potential for deadlock.

```
all_reindeer = []
reindeer_mutex = mutex()
busy_mutex = mutex()
elf_mutex = mutex()

stable_handler(connection)
    reindeer_address = connection.read()
    lock(reindeer_mutex)
    all_reindeer.append(reindeer_address)
    if all_reindeer.length == 9 //If all reindeer in list
        santa_connection = socket(santa_address)
        santa_connection.write("REINDEER READY")
    unlock(reindeer_mutex)

stable(my_address, santa_address)
    listening_socket = socket(my_address)
    while (true)
        connection = listening_socket.accept()
        thread(stable_handler, (connection))

reindeer(my_address, stable_address)
    listening_socket = socket(my_address)
    stable_connection = socket(stable_address)
    stable_connection.write(my_address)
    while (true)
        connection = listening_socket.accept()
        message = connection.read()
        if message == "START DELIVERIES":
            print("Reindeer is out on deliveries")
        if message == "DONE DELIVERIES":
            print("Reindeer is now on their holiday")
            sleep(random(40,120)) //Sleep between 40 and 120 seconds
            stable_connection.write(my_address)

elf(my_address, santa_address)
    listening_socket = socket(my_address)
    santa_connection = socket(santa_address)
```

```
    while (true)
        sleep(random(10,1000)) //Sleep between 10 and 1000 seconds
        lock(elf_mutex)
        santa_connection.write(my_address)
        unlock(elf_mutex)
        connection.read()
        print("Elf is consulting Santa")
        connection.read()
        print("Consultation done")


santa(my_address)
    listening_socket = socket(my_address)
    waiting_elves = []
    while (true)
        connection = listening_socket.accept()
        message = connection.read()
        switch message:
            case "REINDEER READY":
                lock(busy_mutex)
                lock(reindeer_mutex)
                for reindeer_address in all_reindeer:
                    reindeer_connection = socket(reindeer_address)
                    reindeer_connection.write("START DELIVERIES")
                sleep(40)
                for reindeer_address in all_reindeer:
                    reindeer_connection = socket(reindeer_address)
                    reindeer_connection.write("DONE DELIVERIES")
                all_reindeer = []
                unlock(busy_mutex)
                unlock(reindeer_mutex)
            default:
                waiting_elves.append(message)
                if (waiting_elves.length == 3): //Only wait for 3 elves
                    lock(busy_mutex)
                    for elf_address in waiting_elves:
                        elf_connection = socket(elf_address)
                        elf_connection.write(1)
                    lock(elf_mutex)
                    sleep(20)
                    unlock(elf_mutex)
                    for elf_address in waiting_elves:
                        elf_connection = socket(elf_address)
                        elf_connection.write(1)
                    waiting_elves = []
                    unlock(busy_mutex)

main()
    santa_address = 10000
    stable_address = 20000
    reindeer_addresses = 30000
    elf_addresses = 40000
```

```
    Thread(santa, (santa_address)) //Start santa thread
    Thread(stable, (stable_address, santa_address)) //Start stable thread
    for i in range(9): // Start 9 reindeer threads
        Thread(reindeer, (reindeer_addresses+i, stable_address))
    for i in range(12): // Start 12 elf threads
        Thread(elf, (reindeer_addresses+i, santa_address))
```

1.4 Create a Concept Map

Create a concept map for CompSys up to this point. Your concept map should cover the topics covered in the introduction lectures, operating systems, and networking. You may also include material from the architecture part of the course if you so wish. Your map should cover all of the key concepts and topics, and especially how they link together. For those unfamiliar with concept maps you can consult: `https://en.wikipedia.org/wiki/Concept_map`

2 Programming Part (75 %)

For the programming part of this assignment, you will implement a peer in a peer-to-peer file-sharing service. Many of the programming tasks set in the exercises relate to the sub-tasks within this assignment, so if you are stuck at any point consider revisiting them.

2.1 Design and overview

The file-sharing service consists of many identical peer processes. When a peer starts it must be given the address of a peer already on the network, with whom it will make contact to join the network. The first peer to join will not attempt to connect to anyone, as there is no one yet to connet to. This design uses a custom protocol outlined below.

In this protocol each peer can perform either client or server interactions, and must be capable of performing each concurrently. Note that this may also mean that different server interactions may be performed concurrently.

The first peer to join a network cannot perform client interactions, as there would be no peers to interact with. Therefore it will only perform server interactions. We shall refer to this as the *initial peer*, though do note that it is programmatically identical to any other peer. If a new peer, referred to here as the *joining peer* were to attempt to join the network it would first contact the *initial peer* to make itself known to the network. The *initial peer* will then update its record of all peers on the network and reply to the *joining peer* with a complete list of all peers on the network. The *initial peer* will then inform any other peers on the network that the *joining peer* has joined, so that all peers will have an up to date record of all peers on the network. If a third peer or fourth peer joined they could do so in thet same way, using any peer already on the network as their initial peer to contact.

Note that in this assignment we are not overly concerned with security, or if people are allowed to access files. What we are concerned by is the ability to construct a network node (the peer) in such a way that it can act as a client and server at the same time without deadlocking, or causing race conditions.

Note that as in A3, all messsages are limited in how many bytes can sent at once. If a peer requests a file larger than this limit then it will therefore be sent over several messages which the peer will have to reassemble.

For this assignment you are going to implement a peer that registers with a network, retrieves files, and responds appropriatly to those same requests. You can find a complete description of the protocol in Section 3.

2.2 API and Functionality

2.2.1 Key Implementation Tasks

The peer you will be implementing needs to perform the six basic tasks listed below. Each should be completed according to the protocol described in Section 3. The peer should:

2.1. Registering a new peer on the network. The joining peer will send its IP and port to a known peer already on the network, and recieve a complete list of the network as a reply.

2.2. Respond to peer requests to join the network. The peer should add the requesting contact details to its record of the network, and respond with a complete account of the network.

2.3. Retrieve a small file. The user should request a file small enough to fit into a single message. For this the example file tiny.txt has been provided, though you are free to test on others. This should be retrieved by the peer and written into its own file storage. The client should validate the recieved file to check that it has been recieved correctly.

2.4. Respond to requests for files small enough to fit into a single message. For this the example file tiny.txt has been provided, though you are free to test on others.

2.5. Retrieve a large file. The user should request a file large enough to require several messages to completely send. For this the example file hamlet.txt has been provided, though you are free to test on others. This should be retrieved by the peer and written into its own file storage. The client should validate the recieved file to check that it has been recieved correctly. Note that you should not assume that blocks are recieved in order.

2.6. Respond to requests for files large enough to require several messages to completely send. For this the example file hamlet.txt has been provided, though you are free to test on others. Note that the protocol makes no requirement of the order in which blocks are sent, e.g. it is up to your implementation to make a sensible choice.

2.2.2 Test environment

One of the difficulties with developing a system that relies on network communication is that besides a network, you need to have two or more peers running, and potentially debug both simultaneously.

To assist in developing and testing your code, we have provided an implementations of the peer, but written in Python. You can use this to get started with a setup that runs fully contained on your own machine. By running a local Python peer and your own C peer it is possible to debug and monitor all pieces of the communication, which is often not possible with an existing system.

Note that there are two directories within the Python directory, *first_peer* and *second_peer*. This is as you should be testing two sides of interaction with your C peer. E.g. test that your C peer can register with the *first_peer* and retrieve files from it. Also test from the *second_peer* that it can request from your C Peer and have files sent to it when requested. You should also test with all 3 at the same time so as to test your concurrency setup as you CANNOT assume that all client interactions will complete before all server interactions start or vice versa. Adding a sleep into the server interactions will slow down responses, allowing you to check that multiple clients can connect and be responded to at the same time. Note that both Python peers are identical (technically they should be literally the same file just linked), though have been given different config.yaml files and available data files by default. Feel free to alter these configs and files as you wish, they are merely suggestions for some confiugations to test.

You can of course also peek into or alter the Python and get inspiration for implementing you own peer, but beware that what is a sensible design choice in Python *may* not be a good choice for C. You are also reminded that although you can alter the Python as much as you want to provide additional debugging or the like, they are currently correct implementations of the defined protocols. Be careful in making changes that you do not alter the implemenation details, as this may lead your C implementation astray.

To start an appropriate test environment you can run a peer using the command below, run from the *python/first_peer* directory:

```
python3 peer.py config.yaml
```

Assuming you have not altered `python/first_peer/config.yaml`, this will start a peer at 127.0.0.1:12345. This will be capable of serving any files within the *python/first_peer* directory. You can now start another peer to connect to this one. You could use either a second Python peer within *python/second_peer*, or the C. You may need to alter the default configs to get the functionality you require. Note that previously we have used 0.0.0.0 as our address, which tells your system not to use your network driver at all for this connection. Some students have reported this hasn't worked on their machines so we have switched to 127.0.0.1, which will tell your system to use the network driver but route the connection back to the host. Though in practice this means that both are doing essentially the same thing, if you find one does not work try the other, and if that doesn't work talk to a TA.

To make and deploy the C peer run the following within the *src* directory:

```
make
./peer config.yaml
```

By default this will compile your client, though as not yet implemented it will not do anything significant.

Do note that both the Python and C peers as handed out act as simple linear scripts for their client interactions, registering a peer and then retrieving the two required files. This structure has been picked for ease of development and understandability only, and is not intended as demonstration of good practice. If you would rather alter this structure so as to allow for a peer to dynamically request whatever files they wish whenever they wish, then do feel free to do so, though it is not aim or requirement of this assignment.

2.3 Run-down of handed-out code and what is missing

- *src/peer.c* contains the peer, and is where it is expected all of your coding will take place. If you alter any other files, make sure to highlight this in your report. This code contains five guide functions, `server_thread`, `handle_server_request`, `handle_register`, `handle_inform` as well as `handle_retrieve`. These should illustrate where you might begin, but feel free to make whatever alterations to the structure you wish. Your edits WILL NOT be limited to these functions however, and you will NEED to make additional edits to the `send_message` function and potentially others. The `send_message` demonstrates functionality from A3 so you *may* wish to replace it with your own solution to that if you are more famliar with it, though note it does not encompass all of the functionality expected from A4. Your completed system should write any retrieved files to the *src* directory.

- *src/peer.h* contains a number of structs you may find useful in building your client. You are not required to use them as is, or at all if you would rather solve the problem some other way.
- *src/config.yaml* contains the definitions for a peer to be hosted, acting as a joining peer in the network.
- *src/sha256.c* and *src/sha256.h* contain an open-source stand-alone implementation of the SHA256 algorithm, as used throughout the network. You should not need to alter this file in any way.
- *src/common.c* and *src/common.h* contains some helper functions to assist in parsing the configuration file. You should not need to alter this file in any way.
- *src/compsys_helpers.c* and *src/compsys_helpers.h* contains definitions for robust read and write operations. You should not need to alter this file in any way.
- *src/endian.h* contains definitions for converting between big and small endian numbers. This is only useful on a mac, and should not be necessary in the majority of solutions so do not be concerned if you do not use any of these functions. You should not need to alter this file in any way.
- *python/first_peer/peer.py* and *python/second_peer/peer.py* contains the peer, written in Python. This can be run on Python 3.7 or newer, and requires a config file written in the YAML format. You should not need to alter this file, though may find adding more debug print statements helpful.
- *python/first_peer/config.yaml* contains the definitions for a peer to be hosted, acting as an intial peer in the network.
- *python/second_peer/config.yaml* contains the definitions for a peer to be hosted, acting as an joining peer in the network.
- *python/first_peer/tiny.txt* and *python/first_peer/hamlet.txt* are example data files which if you can transfer to and from the peer correctly are enough to say you have met some of the goals of this assignment. Testing using additional files would be advantagous in a report.

- `README` contains the various commands you will need to get this project up and running.

2.4 Testing

For this assignment, it is *not* required of you to write formal, automated tests, but you *should* test your implementation to such a degree that you can justifiably convince yourself (and thus the reader of your report) that each API functionality implemented works, and are able to document those which do not.

Simply running the program, emulating regular user behaviour and making sure to verify the result file should suffice, but remember to note your results. When testing parrallel sytems that may act non-deterministicly any amount of user testing is of limited utility. However, you should attempt to stress-test you system say by running many peers at once, or by maximising chances of race conditions.

One final resource that has been provided to assist you is that an instance of peer.py has been remotely hosted, and which you can connect to. This is functionally identical to those contained in the handout, with the obvious difference that any communications with it will be properly over a network.

The server is designed to be robust, so should be resilient to any malformed messages you send, but as it is only a single small resource be mindful of swamping it with requests and only use them once you are confident in your system. You can connect to the peer at the following address and port:

```
Test Server: 130.225.104.168:5555
```

Note that depending on whatever firewalls or other network configuration options you have, you may not be able to reach the server from home, but that you should be able to from within the university. Additionally, note that this version of the Peer will slightly differ from the presented Python peer, in that it will automatically remove peers from its network it has not heard from for 5 mins. This will prevent old test trackers from clogging up the network and is not required functionality for you to emulate.

2.5 Recommended implementation progress

As mentioned in the previous section, `src/peer.c` contains five unimplemented functions and at least one partially completed function. The satisfaction of these should yield a functioning peer. Do note that you are not limited to completing these functions and are expected to make changes even through the provided functions so as to ensure functionality that avoids deadlock AND race conditions. It is expected you will stick to modifying peer.c, and should specificly highlight in your report if you deviate from this.

In this section, we give a short recap of your implementation todos; this can serve as a checklist for your project, and *we recommend* that you work on them in the order presented here.

2.1. With the peer acting as a client, request to register with another peer and appropriately manage any feedback

2.2. With the peer acting as a client, request to retrieve both the small and large file from another peer and appropriately manage any feedback

2.3. With the peer acting as a server, respond to inform requests from another peer

2.4. With the peer acting as a server, respond to register requests from another peer and generate appropriate feedback

2.5. With the peer acting as a server, respond to retrieve requests from another peer and generate appropriate feedback

2.6. Throughout these tasks ensure that your peer can act as a client and server AT THE SAME TIME. You must ensure that any race conditions or deadlocks are appriately handled

2.7. Manually test your implementation, documenting bugs found and how you fix them (if you are able to).

2.8. Meanwhile, do not neglect the theoretical questions. They may be relevant to your understanding of the implementation task.

2.6 Report and approximate weight

The following approximate weight sums to 75% and includes the implementation when relevant.

Please include the following points in your report:

- Discuss the provided protocol and how you were able to ensure a thread-safe implementation. For instance, what data needed to be shared between the client and server side of the application, and how did you avoid race conditions? Have you done so in a way that also avoids deadlocks+ What circumstances would stress your implementation either by creating errors, or by degrading performance? (Approx. weight: 15%)
- Document technical implementation you made for the peer - cover in short each of the six tasks (as client register, as client retrieve, as server register, as server inform, as server retrieve), as well as any additional changes you made. Each change made should be breifly justified. You should also describe what input/data your implementation is capable or processing and what it is not. (Approx. weight: 30%)
- Discuss how your design was tested. What data did you use on what machines? Did you automate your testing in any way? Remember that you are not expected to have built a perfect solution that can manage any and all input, but you are always meant to be able to recognise what will break your solution. Testing on the provided two files (tiny.txt and hamlet.txt) is not sufficient, you should also test using data that does not

work, or with requests that are incorrect. You should also test with more than 2 peers running at once. (Approx. weight: 15%)

- Discuss any shortcomings of your implementation, and how these might be fixed. It is not necessarily expected of you to build a fully functional peer, but it *is* expected of you to reflect on the project. Even if you implement the protocol exactly as described throughout this document there will still be problems to identify in terms of usability, concurrency or functionality. (Approx. weight: 15%)

As always, remember that it is also important to document half-finished work. Remember to provide your solutions to the theoretical questions in the report pdf.

3 Protocol description

This section defines the implementation details of the components used in the A4 peer-to-peer file-sharing network.

3.1 File structures of the peer

All peers will serve files relative to their own location.

Both the server and client use a base data directory, defined in `config.yaml`. Any files within the servers base directory can be retrieved. Any files retrieved by the client will be placed within its file base.

3.2 Format of the peer client requests

All client messsages from the peer must contain the same header as follows:

```
16 bytes - The IP address the sending peer is listening on,
        as UTF-8 encoded bytes
4 bytes - The port the sending peer is listening on,
          unsigned integer in network byte-order
4 bytes - The command code desribing the nature of this
          message, unsigned integer in network byte-order
4 bytes - The length of the request body, unsigned integer
          in network byte-order
```

The command codes specified above can only be one of 3 values. These, along with the appropriate request bodies for them are:

```
1 - Register, when a peer is attempting to make first
    contact with a network. No body is provided with this
    command
2 - Retreive, when a peer is requesting a data file to be
    transfered to it from another peer. The request body
    must be the filename/filepath of the requested file
3 - Inform, when a peer is informing another peer of a
    third peer that has just joined the network, The
```

```
   request body must fit the 20 bytes described below:
16 bytes - The IP address of the newly joined peer,
           as UTF-8 encoded bytes
4 bytes - The port of the newly joined peer, unsigned
          integer in network byte-order
```

In the case of the Register and Retreive (1 & 2) commands, or if the command could not be determined, a reply will ALWAYS be expected from the other peer. ONLY in the case of the inform (3) command do we not expect a reply.

3.3 Format of the client server responses

All replies from the server to the client must contain the same header as follows:

```
4 bytes - Length of response body,
          unsigned integer in network byte-order
4 bytes - Status Code of the response,
          unsigned integer in network byte-order
4 bytes - Block number, a zero-based count of which block in
          a potential series of replies this is,
          unsigned integer in network byte-order
4 bytes - Block count, the total number of blocks to be sent,
          unsigned integer in network byte-order
32 bytes - Block hash, a hash of the response data in this
           message only, as UTF-8 encoded bytes
32 bytes - Total hash, a hash of the total data to be sent
           across all blocks, as UTF-8 encoded bytes
```

In addition to the header each response will include an additional payload of the length given in the header. Note that in the case of a small enough response to only require a single reply, then the block hash and total hash will be identical. If the response is to a request for a data file, and it could be retrieved with no errors then the payload will be either all or part of said file, depending on the size of the file. If the response is to registering on the network, the response will be a list of all known peers on the network. Each entry in the list will be formatted as:

```
 16 bytes - The IP address of a network peer,
            as UTF-8 encoded bytes
4 bytes - The port of a network peer,
          unsigned integer in network byte-order
```

The list itself will be some multiple of 20 bytes long, with the first 20 bytes being one peer, the second being another and so on. Note that it is not defined behaviour if this list includes the joining peer, and it is up to you to ensure your network list does not contain duplicate peers. In all other cases the response will be a feedback message explaining any errors or results of other querries.

You may notice that this is suspiciously similar to that of A3, therefore you can probably use some of your code from that if it helps.

The following error codes may be provied in response by the peer:

```
1 - OK (i.e. no problems encountered)
2 - Peer already exists (i.e. could not register a peer as
    they are already registerd with the network)
3 - Bad Request (i.e. the request is coherent but cannot be
    servered as the file doesn't exist or is busy)
4 - Other (i.e. any error not covered by the other status
    codes)
5 - Malformed (i.e. the request is malformed and could not be
    processed)
```

3.4 Requests/Responses and persistent connections

Note that there are no persistent connections between peers. Each time a peer sends a requests, the serving peer will respond with one or more reply messages, but it will then close the connection. To request multiple files, a peer will need to open a separate connection each time.

Submission

You should hand in a ZIP archive containing a `src` directory containing all *relevant* files (no ZIP bomb, no compiled objects, no auxiliary editor files, etc.).

Any Python code should also be included in your submission as it may form part of how you tested your C code (hint). As this course is not a Python course, you will not be marked according to the quality of your Python code.

To `make` this a breeze, we have configured the `src/Makefile` such that you can simply execute the following shell command within the `src` directory to get a `zip`:

```
$ make zip
```

Note that depending on any test files, or how you implemented your solutions you may wish to include other files not zipped by the above command. You are allowed to do so, but are encouraged to be sensible in what you include (E.g. please no 10GB test files).

Alongside a `src.zip` submit a `report.pdf`. For submission, sign up for a group at:
`https://absalon.ku.dk/courses/70194/groups#tab-23701`
Hand-in as a group as in other courses.