EE 565: Computer Communication Networks I
Winter 2023

Project 2: Transport System

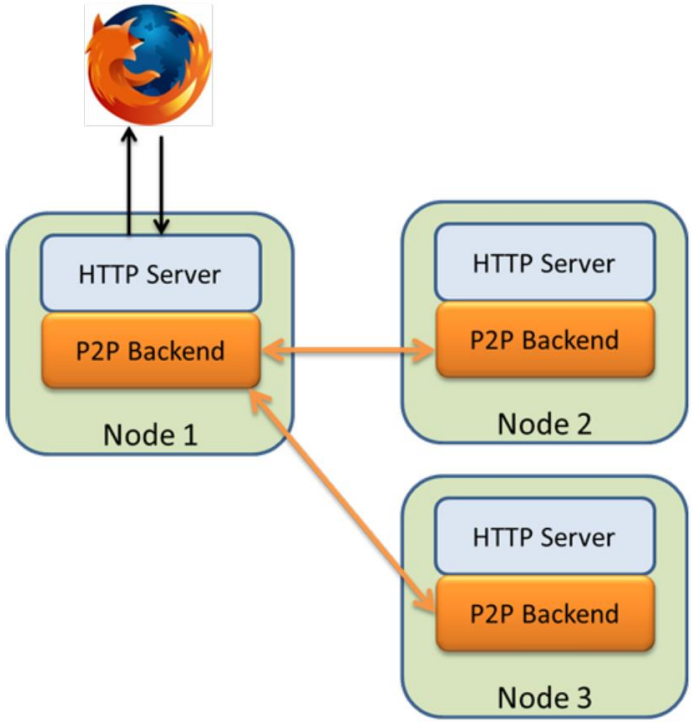
Deadline: Feb 4, 2023. 11:59 PM PST

Questions? Canvas Discussion => Canvas Message => TA Coding Hours

1. HTTP Server

In this project, we will extend your server from Project 1 and implement an additional transport layer to transfer the files between multiple peer nodes. Recall that in our system, each node consists of two components: The front-end component which interacts with users and the backend component which takes care of the overlay P2P network and content sharing/transfer.

In your first project, you have implemented the front-end component of the system, which can interact with users through a web browser. In this project, you will design the back-end component of the system. Our initial version of the back-end includes a custom transport protocol suitable for transferring video contents between peer nodes as shown in the figure below.



You also have to extend your front-end HTTP server by recognizing custom URIs. These URIs allow users to utilize the back-end without a need to know the detail or implementation of the underlying components.

The back-end transport protocol will be built on top of UDP. You should understand the concept of transport layer in order to successfully implement the backend. You should also be able to apply techniques from datalink layer and ARQ to the backend transport protocol as well.

A HTTP server, often known as a web server, provides contents needed for web browsers. These contents could be a HTML file, a plain text file, an image, audio, video, or just a binary file. In a nutshell, our HTTP server will act as a simple file server. It will take a resource identifier (URI), and locate the content for the URI in a directory relative to a content root. Then generate a HTTP response message that contains information about the content, and the content itself.

2. Project Specification

2.1 Objectives

Implement a reliable transport layer based on UDP, which is able to transfer the content between multiple peers.

2.2 Extra Port

Your server should take an extra argument, which designates the back-end communication port. For example, the following command should start the server, which listen to HTTP on TCP port 8345, and listen to your back-end protocol on UDP port 8346.

$ vodserver 8345 8346

$ java vodServer 8345 8346

2.3 HTTP Interface

Our test client will interact with your peer node using HTTP protocol. You need to modify your project 1 server to recognize these URIs.

2.3.1 **Associate peer node with content [Required]**

/peer/add?path=<contentpath>&host=<IP/Hostname>&port=<backendport>&rate=<kbps>

This URI let the node knows that the specified remote content with the given path could be found on another node with the given IP address or hostname and port. The server should generate a 200 OK response with valid text/html content. However, we do not concern about the actual content of the page for this URI. You could use this to your advantage by generating any useful debugging information.

This URI let the node knows that the specified remote content with the given path could be found on another node with the given IP address or hostname and port. The server should generate a 200 OK response with valid text/html content. However, we do not concern about the actual content of the page for this URI. You could use this to your advantage by generating any useful debugging information.

Note that we should be able to request this URI multiple times with the same path to give a list of peers which have the same content. Since we do not have peer exchange / discovery protocol at this point, this URI only serves as a shortcut to locate peer with the content. It will be deprecated in later projects.

Example:

http://localhost:8345/peer/add?path=content/video.ogg&host=pi.ece.cmu.edu&port=8346&rate=1600

This URI tells the node that the file “content/video.ogg” could be found on pi.ece.cmu.edu. The content path is relative to the directory where we start the peer. The back-end port is 8346. The average bit rate for the content is 1600 kilobits per second.

2.3.2 **View content [Required]**

/peer/view/<contentpath>

This URI could be called by a web browser to retrieve the remote content using the specified path. The response and entity body should be the same as if the content is located locally on the node. All requirements from project 1 applied (e.g. you must handle Range request or partial content properly.)

When implementing this support, please keep in mind that for later projects, this URI should work without requiring the /peer/add to be requested first.

Example:

After we enter this URIs (use the browser)

http://localhost:8345/peer/add?path=content/video.ogg&host=pi.ece.cmu.edu&port=8346

Then this URI should (immediately) play the video.

http://localhost:8345/peer/view/content/video.ogg

2.3.3 **Configure back-end transfer rate [Required]**

/peer/config?rate=<bytes/s>

This URI specified the bandwidth limit (in bytes per second) that the backend transport can be used for content transfer. We will calculate the rate by averaging the amount of traffic on the transport port (not HTTP port) over 10-second interval. If rate = 0, or if we did not request this URI, then there is no limit on traffic.

The server should generate a 200 OK response with valid text/html content. We do not concern about the actual HTML content for this request.

Example:

http://localhost:8345/peer/config?rate=8192

Set a bandwidth limit on the backend transport to 8192 bytes/s.

2.3.4 **Display transfer status [Optional]**

/peer/status

Sometimes, we would like to see the status of the current transfers. It would be useful if the node could generate an HTML page which tracks the progress and bit rate for current transfers. The specification of the format/output to be shown by this HTML page is left to you.

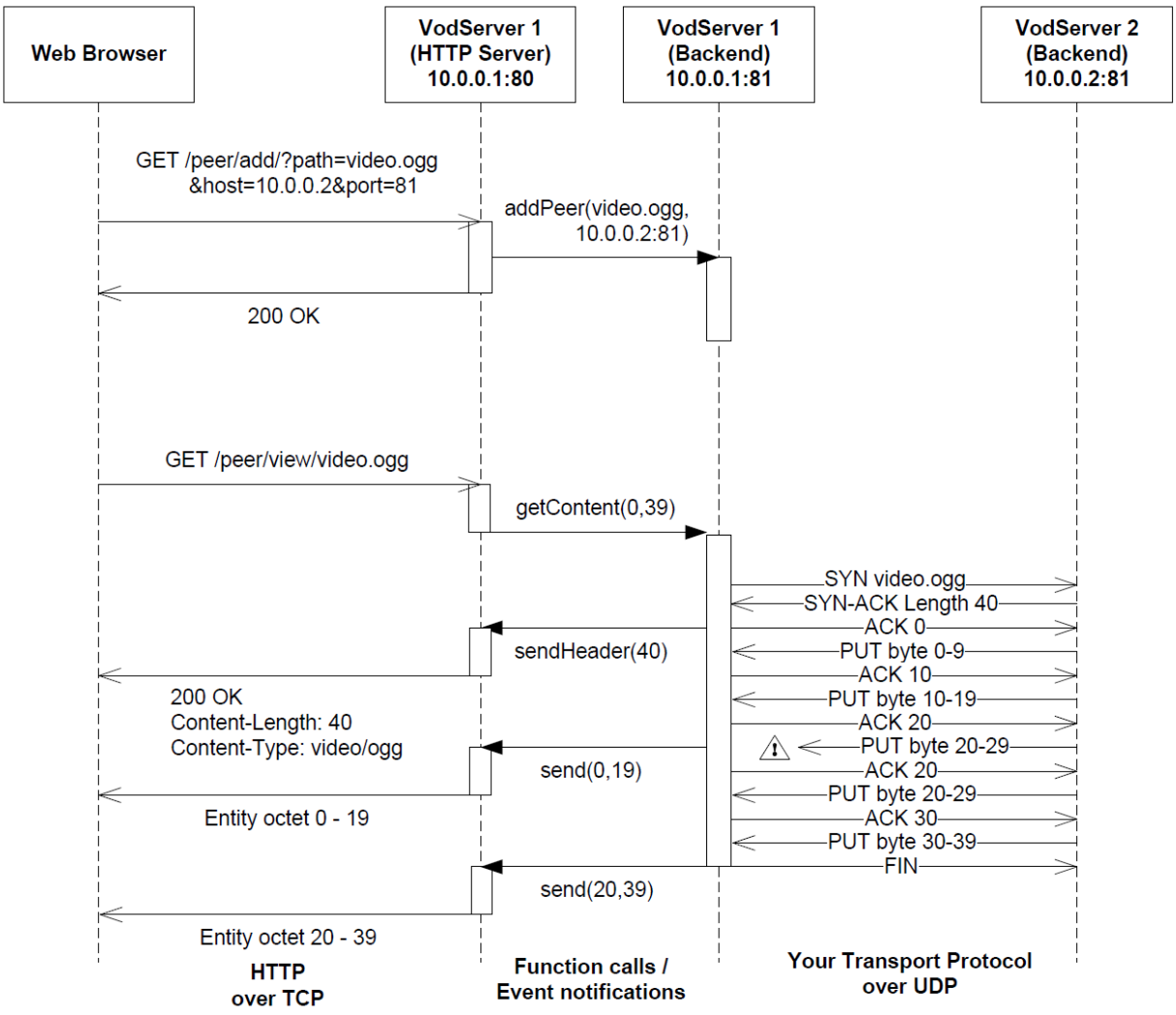
2.4 Back-end Transport subsystem

In this project, you will implement a back-end subsystem which utilizes a custom transport protocol on top of UDP. The purpose of the transport protocol is to reliably transfer content amongst peer nodes.

You are welcome to optimize your protocol for transferring Ogg stream. There is no specification on your protocol or packet format. However, make sure your back-end subsystem can achieve the following requirements:

- **Multiplex:** You are allowed to use only one port for the backend transport (include sending & receiving). All backend traffic (even simultaneous transfers) should go through this port.
- **Robustness:** Can reliably transfer the content, that is it can endure at least 5% packet loss
- **Bandwidth Limit:** Can maintain the bandwidth specified in /peer/config option (can stay within 10% of the maximum bandwidth)
- **Simultaneous transfer**: Can simultaneously transfer multiple (at least five) content requests at the same time.
- **Content chunking:** When the same content is found on multiple peers (by sending multiple /peer/add requests with the same content path), the backend should be able to retrieve different parts of the content from different peers.

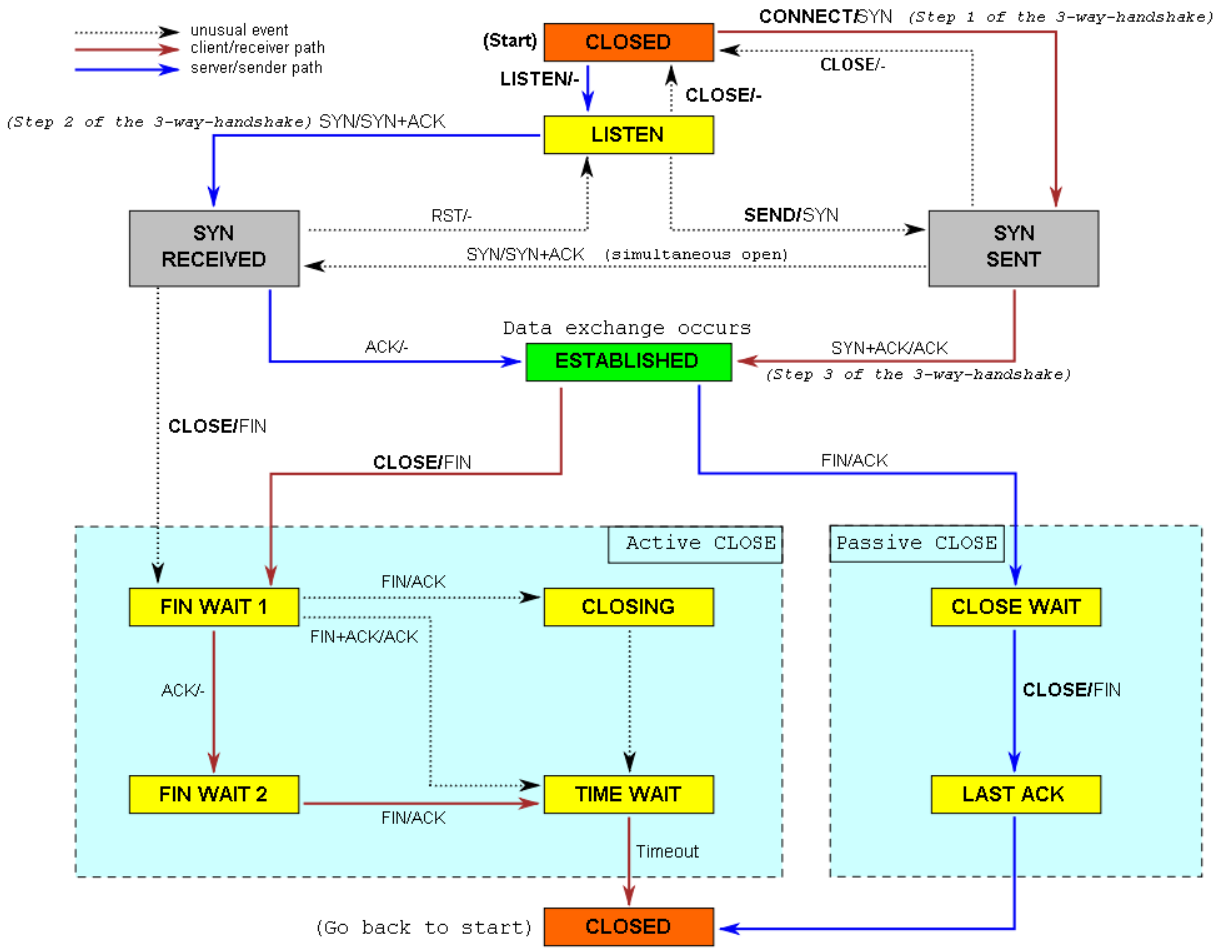
We will not directly interact with your backend. However, we will use the HTTP interface in conjunction with packet monitoring tools to check whether your back-end satisfies the requirement.



3 Discussion

There are many ways to complete this project. However, before diving in and implementing, it is often more helpful to identify possible communication steps. Once you have figured out the possible cases, you should be able to identify the states for your protocol and design a proper packet format for each state.

For example, a simple communication diagram for a short content transfer (in its entirety) is shown below. Note that you need to design your own transport protocol and back-end system. Be advised that you should be able to utilize the same port for all backend traffic.



One important clue is that your transport protocol could be optimized for transferring Ogg stream (http://www.xiph.org/ogg/doc/). You could leverage this knowledge and implement a transport protocol much simpler and more efficient than TCP.

You may also want to consider whether the request functionalities should go into the transport protocol. Or it could be implemented at the application level. For example, to support content chunking, the backend application could utilize multiple transport channels to transfer different parts of the files from different peer nodes. Or you could design a transport layer which supports many-to-one file transfer. Try to figure out which approach is simpler, more extensible, or more compatible with the front-end requests.

4 Deliverable Items

Due to the complexity of the project, we sincerely request you start early! **(Deadline : Feb 4, 2023. 11:59 PM PST)**

We suggest that you submit two versions of the Project:

4.1 Checkpoint

This project requires significantly more effort than project 1. Especially, the backend transport system could get really complicated. You should be able to deliver at least a 'work-in-progress' version by the checkpoint deadline. Please use the final submission guideline for submitting files. Your project must contain an initial version of the backend and must be able to perform the following tasks:

- Recognize the URIs for HTTP interface
- Send and receive a small file in its entirety (less than 64 KB in size, without HTTP range request)
- Simultaneous transfer
- Initial design document of your transport protocol (design.txt, or design.pdf). This document should include packet format and state transition diagram. For example, the TCP header packet format and its state transition diagram is shown below (note that your protocol could be much simpler than TCP)

TCP Header

Bit offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Source port																Destination port															
32	Sequence number																															
64	Acknowledgment number																															
96	Data offset				Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
128	Checksum																Urgent pointer															
160 …	Options (if Data Offset > 5) …																															

4.2 Final Deliverable Items

The deliverable items are enumerated as follows:

- The source code for the entire project (if you use external libraries unavailable on the cluster, provide the binaries needed for compilation of your project)

- Makefile - You should prepare a makefile for C make (or build.xml for Java Ant) which generates the executable file for the project. We expect the file to work from your submission directory. For example, the following calls should generate an executable vodserver (for C) or VodServer.class respectively. “make” and “ant” commands will be attempted on your submission directory, depending on your choice of programming language.
- Do not submit the executable files (we will DELETE them and deduct your logistic points). However, we must be able to invoke the one of the following commands (after invoking make/ant) from your submission directory to start your server on the given port number (e.g. 8345 & 8346 in this case).
  $ vodserver 8345 8346
  $ java vodServer 8345 8346
- A brief design document regarding your server design in design.txt / design.pdf in the submission directory (2-3 pages). Apart from the team information, tell us about the following,
  - A summary of your backend system (how did it communicate with the front-end or the others? How did you utilize the transport protocol?)
  - Final protocol design
    - Packet Format
    - State transition diagram
    - Changes to protocol, comparing to the checkpoint
  - Libraries used (optional; name, version, homepage-URL; if available)
  - Extra capabilities you implemented (optional): Please specify in this section if you have implemented any extra credit options (or want to enter the competition)
  - Extra instructions on how to execute the code, if needed

5 Grading

Partial credits will be available based on the following grading scheme:

Items	Points (100 + 20)
Checkpoint submission - Protocol design document - HTTP interface - Partial transport system	**20** 10 5 5
Logistics - Successful submission & compilation - Design documents	**15** 5 10
Transport System - Port multiplex (use one port for back-end) - Small file transfer (64 KB) - Large file transfer (400 MB) - Simultaneous transfer (at least 5 x 400 MB transfers) - Robustness (complete transfer with 5% packet loss) - Bandwidth limit - Content chunking	**60** 5 5 10 10 10 10 10
Integration - Video playback with /peer/view	**5** 5
Extra achievements - Cool GUI* - Video play & seek with /peer/view (up to 5 clients per node, without users being annoyed**) - Extra robust (complete transfer with 50% packet loss) - Instant play (takes less than 5 seconds to begin playing the big_buck_bunny video with /peer/view)	**20** 5 5 5 5

*Take sometimes off debugging and write a cool HTML home page for users to interact with your node! It is a web server after all. Make sure you have the /peer/status page.

** Your node should be able to serve Firefox_Final_VO.ogv to 5 clients simultaneously, without any users seeing the 'buffering…' after they start playing the video.

*** Lowest ratio between backend traffic divides by content length. To qualify, the system must pass the robustness requirement

**** Given a bandwidth limit of 80 Mbps, how many videos with average bit rate of 2.2Mbps videos you can play simultaneously with /peer/view? (try big_buck_bunny_480p_stereo.ogg)

6 Controlled Packet drop

In a local area network such as the ECE cluster environment, packet dropping is quite an unusual event.

To facilitate testing and debugging of your project, we have provided a controlled packet dropper tool (nf in Project2/nf), which can (reliably) intercept and drop your packet at a designated rate.

The tool must be used in conjunction with iptables (http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html). You will also need libnetfilter_queue package (Fedora/Redhat) or libnetfilter-queue1 (Ubuntu/Debian) and an administrator privilege to use the tool. iptables is used to identify packet. It is often used to implement a firewall in modern Linux system.

iptables can identify a packet (which match a user-defined rule) and assign it to a custom queue. Our tool will control the queue and decide whether to drop the packet.
For example, if your back-end is using port 8346 and you want to enable packet dropping at a 30% rate, you can issue the following commands (with administrator privilege)

```
# iptables –I INPUT –p udp –m udp --dport 8346 -j NFQUEUE --queue-num 0

# iptables –I OUTPUT –p udp –m udp --sport 8346 -j NFQUEUE --queue-num 0

# ./nf 30
```

After you are done with testing, you can end the nf process and remove the rules.

```
# iptables –D INPUT –p udp –m udp --dport 8346 -j NFQUEUE --queue-num 0

# iptables –D OUTPUT –p udp –m udp --sport 8346 -j NFQUEUE --queue-num 0
```