# Classes

## Node

Node is an abstract parent class of Client, Router and Server. It has virtual functions send() (both versions), receive() and print():

- send() pops a packet pointer from queue_, and sends it forward
- send(Packet* packet) is used for cheat packets (see doc/README)
- receive(Packet* packet) is called when a packet arrives at the node. It's checked, that adding the packet's size_ to current_capacity does not exceed max_capacity_ and if exceeds, the packet is dropped (not sent forward and not enqueued (also note that the packet has already disappeared from the sending queue))
- print() prints different interesting information about the node, such as id_, capacities, and connection information

As these are virtual functions, they will have a little different implementation on the different derived classes.

There are also non-virtual functions, such as change_max_capacity(). They are mostly self-explanatory, after we know the class members:

- unsigned short id_, unique identifier for the object. CANNOT BE 0 (ZERO)
- std::queue<Packet*> packets_, normal queue for packet pointers, FIFO
- unsigned int max_capacity_, represents how many kilobytes of data can be stored in node
- unsigned int current_capacity_, how many kilobytes of data are currently stored in node
- unsigned short connection_id_, all end nodes (client, server) are always connected to only one router, and routers can only have one end node. Not all routers have end nodes connected to them, and in that case its connection_id_ is 0
- std::pair<unsigned int, unsigned short> connection_cost_, bandwidth, latency (Kb/s, ms) speed of link to end node. If not connected to end node, 0 and 0

## Client

Clients are end nodes which send packets to servers. It has unique class members:

- std::string name_, human-readable name that is used in packet creation

Client has unique functions:

- create_packet() creates a packet based on input from GUI. The packet is sent forward immediately (with send(Packet* packet)) if it's a cheat packet, else the packet is enqueued.

Client's implementations for the virtual functions are as follows:

- send() pops a packet pointer from queue_, gets the correct router from main map using connection_id_ as key, calls packet's wait() and then calls the router's receive()
- send(Packet* packet) is called when a cheat packet is created in create_packet(). Nothing is popped from the queue, as cheat packets bypass them. Otherwise, forwarding is done the same as above.
- receive(Packet* packet) prints useful information about the received packet. The packet is deleted after this.
- print() prints in addition to other stuff name of the client

# Router

Routers are not end nodes, but have unique class members:

- std::map<unsigned short, unsigned short> routing_table_, <node_id, node_id> final destination, where to send pairs.
- std::map<unsigned short, std::pair<unsigned int, unsigned short>> neighbors_, < node_id, <bandwidth (Kb/s), latency (ms)> > pairs of pairs, list of direct connections to neighbouring routers (recall that end node information is in connection_id_ and connection_cost_)

Routers have unique functions:

- create_routing_table() using Dijkstra and predecessor lists, establish routing tables so that the cheapest paths to nodes not even neighbouring this router are known.
- print_routing_table() prints the table in human-readable format.
- send_to_end_node() if the packet's receiver_id_ is the same as this router's this function is called, and the packet is forwarded using connection_id_ and connection_cost_.

Router's implementations for the virtual functions are as follows:

- send(), pop from queue, check receiver_id_(if same call send_to_end_node()), use that as key to routing table, which gives the correct router, and call its receive
- send(Packet* packet) same as above, but no popping just use the arg
- receive(Packet* packet) maybe print information and enqueue or send forward, depending on flags.
- print() prints connections to other routers

# Server

Servers change packet data on arrival. Class members:

- name_, name of server, such as Netflix or Youtube etc.
- content_type_, string, this is appended to the content. So Youtube would have content_type_ “mp4”, Instagram “.jpg” etc.
- content_size_, int, represents the requested data’s size (in KB), so Instagram would have some hundreds of kilobytes, and Netflix could have a million kilobyte content_size_ (as movies are larger files than pictures)

No unique functions, but receive() is quite different:

- send()(s) are the same as client
- receive(Packet* packet), swaps the packet’s sender_ and receiver_ids, appends content_ with content_type_, and changes the packet size_ to content_size_. Then enqueues the packet or sends it forward.

# Packet

Packet, or its pointer is the object that is moved between nodes’ queues. Packets are created in client’s create_packet() function and deleted in client’s receive() function.
Packets have options: debug packet prints useful data along the way (where it arrived and from where etc.) and cheat packets bypass queues. Cheat packets are used in debugging to ease showing of different features.
Class members:

- unsigned int size_, represents the packet’s size as kilobytes
- unsigned short sender_id_ and receiver_id_, the end nodes’ connections ids (router ids)
- std::string content_, string representing the requested data, for example a packet to Instagram could have content “doge”. This is mostly for immersion reasons, but it’s also easier to see that the server works as intended.
- time_t time_sent_, timestamp created with <time.h> library, timestamps creation moment. When a packet arrives to client in the end, RTT is calculated based on this
- bool debug_, is this packet a debug packet or not
- bool cheat_, is this a cheat packet or not

Packet also has a function wait(), which simulates transmission and propagation delays, and is called when a packet is sent. When we initially implemented this function, we used the <unistd.h> library, but that was not suitable for the

GUI-version, therefore in the GUI code the waiting works little differently, but the logic is the same.

- args: bandwidth (Kb/s) and latency (ms)
- convert size_ (KB) to Kb and calculate transmission delay
    - If less than 1 (seconds), sleep microseconds
    - else, sleep seconds
- wait propagation delay, convert latency to microseconds and use usleep