Cygnus - Distributed/Decentralized Cloud Gaming Service

Daniel Brodsky, James Cho, Rachel Yeo, Dmitry Narkevich
c8z9a, q1v0b, c7l0b, v7w9a
Github: https://github.ugrad.cs.ubc.ca/CPSC416-2018W-T1/P2-q1v0b-c7l0b-c8z9a-v7w9a

Introduction:

Cloud gaming is becoming a popular option amongst gamers right now, largely due to improvements made in protocol latency and Internet bandwidth for transfering the game feed to other machines in real-time. Nvidia, being the largest manufacturer of GPUs, initially created a platform for playing video games on your mobile device that were being hosted on your personal machine. They have since created a new platform currently in beta, which enables you to play games using one of their server machines connected with a high-end graphics card. In doing so, what would have cost hundreds or thousands of dollars to experience, such as playing the newest AAA games with all settings maxed-out, is now only as costly as what Nvidia charges for the usage of their hardware. However, because of Nvidia's centralization in certain geographic regions, latency on their platform can become a significant issue for people who live far from Nvidia's servers. This cancels out the benefits of not having to own the hardware, and leaves many of these people with no alternative other than to buy the hardware themselves.

The platform we wish to build is a partially-decentralized cloud gaming platform where users can act as hosts and allow other user to play games using their hardware. In doing so, latency in sending the video data can be heavily reduced by having the host closer to the client than if there was a single location for the server like for Nvidia's cloud gaming platform. By creating a system which connects clients to nearby hosts, we can provide a higher-quality service to users.

Technical Description

Components:

Our system consists of a peer-to-peer network of nodes. A node is initialized with a list of peer nodes to connect to, an ID and a blacklist. The provided peers are assumed to already be in the network, and the given ID is assumed to be unique. When a node first joins the network, it sends a message to its peers in order to notify them of its presence. Nodes in the system can either be a host node or a client node. These roles are assigned upon initialization of a node and do not change. These two roles are described in further detail below.

- **Host nodes:**
  A host node is a node that allows a client to play games using their hardware. When requests for hosts are made, host nodes respond to the client with their availability. If the host is available, they also ping the client to find the average RTT, and send it back to the client as well. Host nodes use their blacklist to accept or reject hosts that clients choose.

- **Client nodes:**
  A client node is a node that is playing a game using a host's hardware. Before streaming, the client is responsible for sending out a host request to find itself a host to stream from. Clients always respond that they are unavailable when requests for hosts are sent out, but they still participate in the blacklisting of hosts.

Client-Host Establishment:

Establishment of a client-host pairing takes place over three phases. The first phase allows the client to see which nodes are available and willing to host for them. The second phase is where the client sends their choice out to the network and consensus is performed regarding the client's choice. Finally, if there is a majority agreement, the client floods the confirmed pairing to the network. This allows the chosen host to prepare themselves for the incoming client connection, and the client is then able to connect to the chosen host and begin streaming.

Phase 1 - Establish which hosts are available

A client first floods a request for a host to other nodes in the network. This request contains the client's IP address. When other nodes receive this request, they will respond back to the client over a UDP connection. These responses come in two forms:

(1) If the node is unable to host, either it is a client node or is currently already hosting a game for another node, it will send a response indicating that it is are unavailable.
(2) If the node is currently available, it will ping the client three times over UDP. Once this is done, the average RTT of the pings is sent back to the client, along with an indication that the responding node is available.

After sending out the request, the client waits and collects responses for a total of five seconds. Any responses sent after this time window are not taken into consideration.

Phase 2 - Choose the best host

Once the responses are collected, the client chooses the available node with the lowest RTT. If there are no available hosts in the network, the client sees an error and is unable to stream. After making their selection, the client then floods their choice back out to the network. At this step, consensus is done with the other nodes, and they vote on the host that was chosen. Only nodes whose responses were received in phase 1 are considered when tallying votes. The nodes vote on the chosen host using a blacklist system. If the chosen host is in their blacklist, they will reject the host selection. Otherwise, they approve of the host selection.

The client tallies the votes. The selection is finalized if there is a majority agreement on the chosen host. If a majority of the nodes disagree with the chosen host, the client gets shown an error message and is unable to stream. If they still wish to find a host, they are to repeat the host search process from the beginning again.

Phase 3 - Notify the chosen host

After the decision is finalized, the client floods the client-host pair to the network. When the chosen host receives this message, there are two possibilities:

(1) There was a race condition, and the chosen host has already been taken by another client. In this case, the host will send an acknowledgement back to the requesting client, indicating that they will not be accepting the client.

(2) The host is still available, in which case they will mark themselves as taken and will start indicating that they are unavailable when requests for hosts are made. The chosen host will also send an acknowledgement back to the client, indicating that they are willing to accept the client.

The client waits a maximum of five seconds for the chosen host to acknowledge the client-host pairing. This is to account for the cases where the chosen host has failed or left the network. If the client receives acknowledgement that the chosen host will accept them, they make the connection to the host and begin streaming. If, however, the client doesn't receive an acknowledgement before timeout, or are told that the chosen host will not accept them, they are shown an error message and are unable to stream. The search process must then be executed from the beginning again if they wish to find a new host.

Blacklist:

Each node is initialized with a host blacklist which contains host IDs. The purpose of this blacklist is to restrict which nodes are allowed to host. For example, if a node has a reputation of being unreliable and experiences many failures, it might not be the best choice for a host. This blacklist comes into play when consensus is performed on a client's host selection. When a node sees the host selection, it will consult its blacklist to see if the selected host is on the list. If it is on the list, they reject the host selection. If not, they accept the selection. The node's decision is then sent back to the client.

Since the client looks for a majority agreement before finalizing their host selection, this allows the filtering out of bad hosts in the network.

Failures:

Nodes monitor each other, so in the case where a node fails, the failure is detected by all peer nodes. When this happens, the peer nodes removes the failed node from their list of other peers, and no longer sends messages to it. For host and client connections, failure is detected on both the host and client side when they stops receiving the continuous stream of data from the node they're connected to. The data being sent from the client to the host consists of key inputs. In the case that the client is inactive but remains connected to the host, no-op instructions are sent to the host in order to prevent an unwanted timeout. Data being sent from the host to the client is the video stream for the game that is being played. In this case, the stream acts as a heartbeat and failure to receive the stream after a set timeout results in a failure being detected. If this failure is detected by the host node, the host node will change to an available state and wait to receive a new client from the network. If the failure is detected by the client during gameplay, the client will attempt to find a new host by flooding a new host request to the network. If there are no available nodes remaining for the client to connect to, the client receives an error message, and is no longer able to stream their game.

Disconnects:

Disconnects are handled the same way failures are. When a node wishes to leave the network, it simply does so. There is no need for it to notify the other nodes. Other nodes detect the node’s departure through the failure detection system, and handle the node’s absence accordingly.

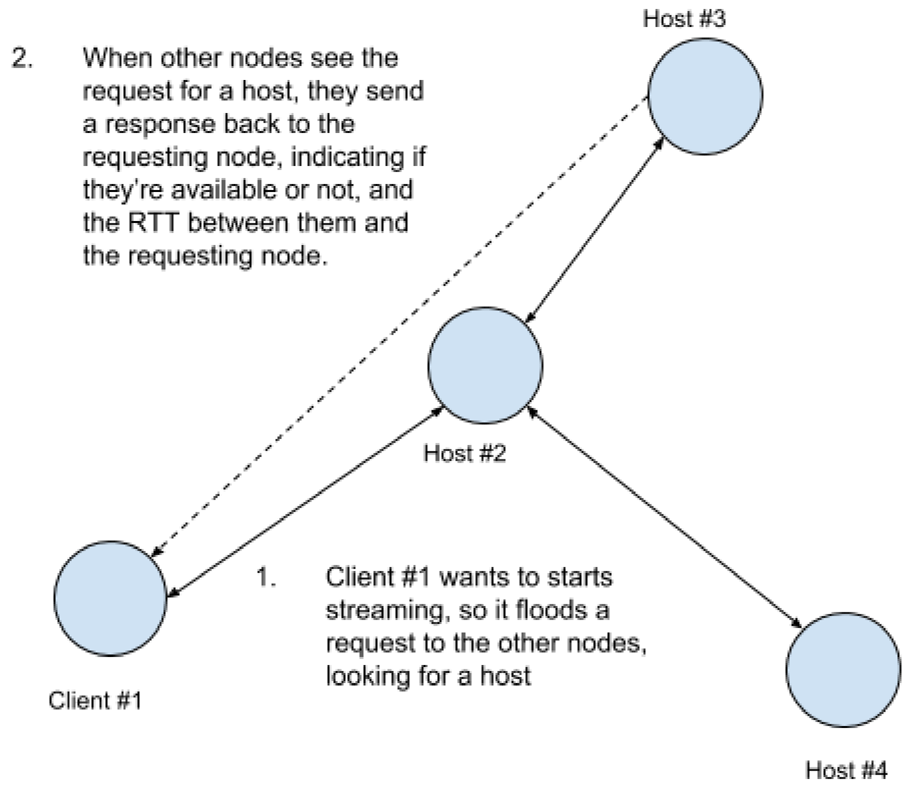
Streaming specifications

Video from X11 on the host is captured and streamed by ffmpeg over the RTP protocol. The client views the stream using ffplay, which generates an additional output of the current status of the tool. This output is parsed for events indicating that the tool is still receiving video data, and is how the system detects failures on the host side.

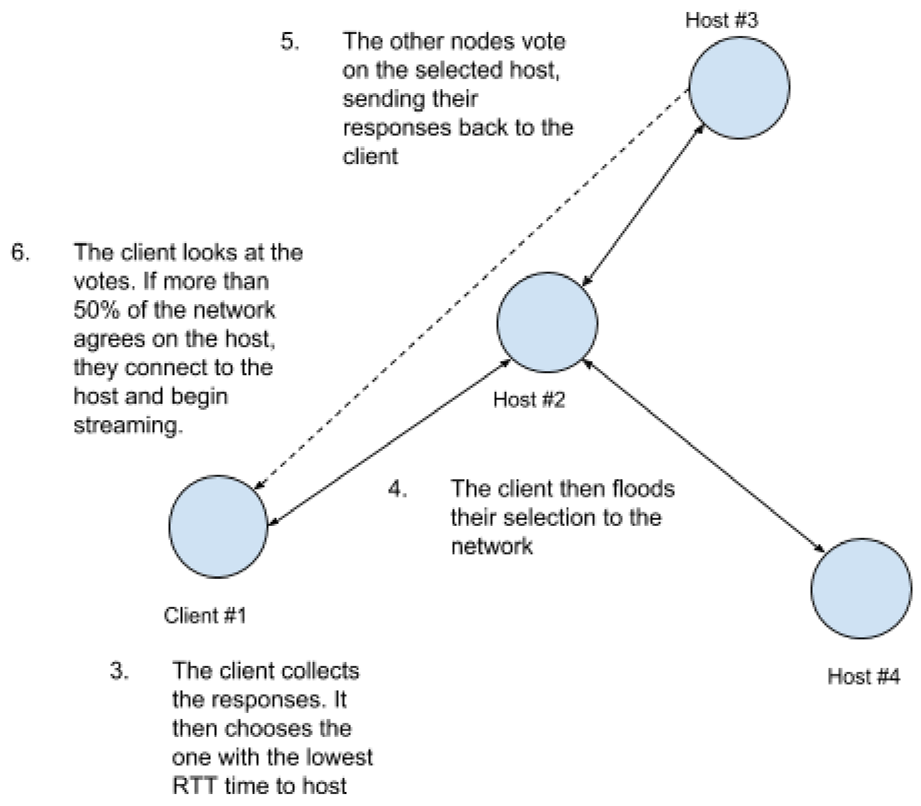
Keyboard and mouse events are captured on the client by parsing the output of xinput, extracting the event type and details, then sending it serialized as JSON over a TCP connection. On the host side, events are simulated using xdotool, which supports sending input events to a specific window, protecting against the client opening a shell and taking over the host. As of now, client mouse and key inputs are being correctly sent and received on the host end, but only the key inputs are being replicated on the host machine.

Connection Schematic diagram

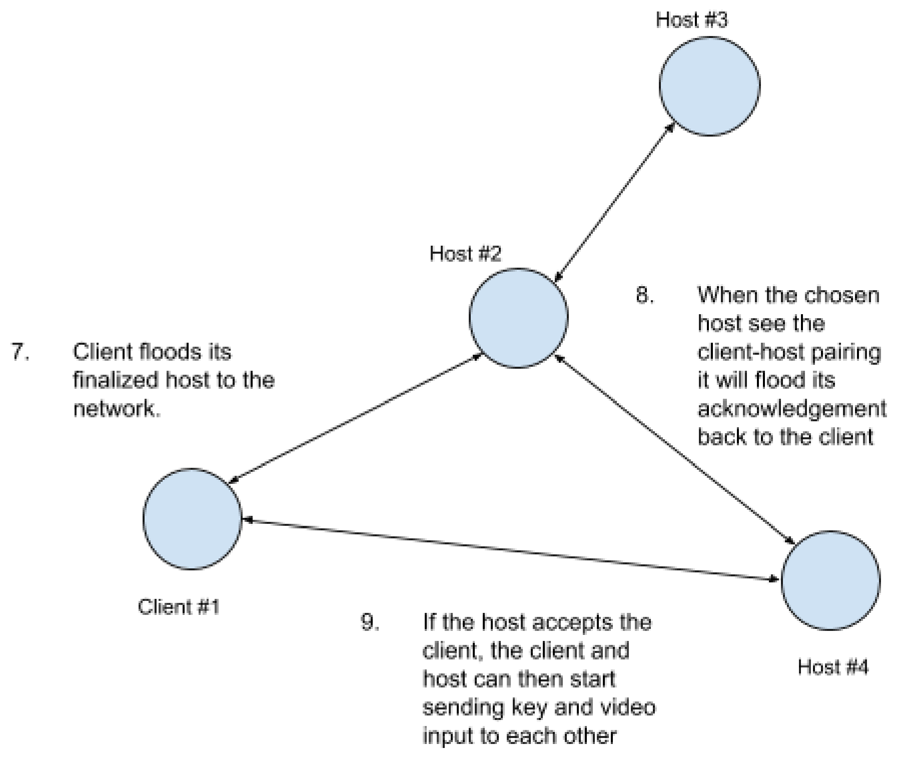
Phase 1


Host #3
2. When other nodes see the request for a host, they send a response back to the requesting node, indicating if they're available or not, and the RTT between them and the requesting node.
Host #2
1. Client #1 wants to starts streaming, so it floods a request to the other nodes, looking for a host
Client #1
Host #4

Phase 2


Host #3
5. The other nodes vote on the selected host, sending their responses back to the client
6. The client looks at the votes. If more than 50% of the network agrees on the host, they connect to the host and begin streaming.
Host #2
4. The client then floods their selection to the network
Client #1
Host #4
3. The client collects the responses. It then chooses the one with the lowest RTT time to host

Phase 3


Host #3
Host #2
8. When the chosen host see the client-host pairing it will flood its acknowledgement back to the client
7. Client floods its finalized host to the network.
Client #1
9. If the host accepts the client, the client and host can then start sending key and video input to each other
Host #4

Evaluation specifications

Evaluation was done using both local machines and Azure VMs. Azure VMs run in a headless state and do not have a physical screen, so we used RDP to create a virtual display for the VMs in order to demonstrate the functionality of our system. Since the lack of a dedicated graphics card on these machines drastically reduced streaming quality, we tested graphics using a more powerful machine of our own that was connected to a client running on the same network. This machine then ran a pre-installed game from its Steam library and performance was evaluated.

Additional evaluation was performed to make sure that hosts were selected correctly, and clients were actually choosing hosts that would give them the lowest latency. We also ran different test cases varying the blacklists of each of the nodes, in order to check that the nodes were coming to the correct consensus about a host, and clients were heeding the reached decision. Finally, we tested node churn, checking if proper behaviour was taken when new nodes were added and when existing nodes left the network or failed.

Demo Evaluation

Part 1 - basic functionality

- A host node will be set up on an Azure VM and a node that wishes to become a client will be initialized on a different Azure VM and will connect to the host node
- A demonstration of how the client requests the host for a streaming host, the host selects itself, and the client and host are able to send key & video input to each other

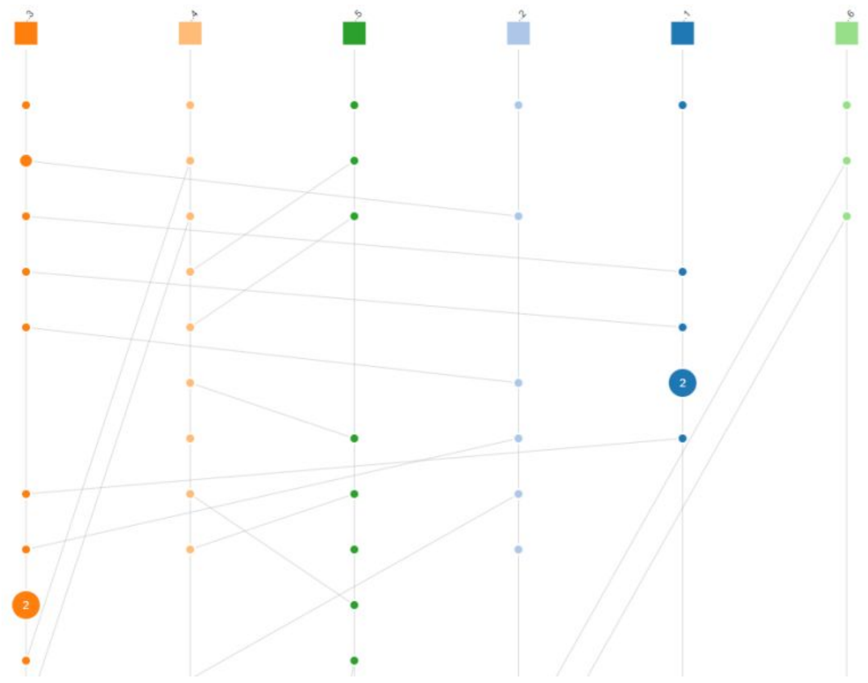
Part 2 - consensus

- A network will be configured with 5 nodes, where nodes are connected in a cross fashion
- One of the nodes will be blacklisted by a majority of the nodes in the network
- When a client tries to choose this node as a host, the consensus fails and the client is asked to choose a different host
- When the proposed host is not blacklisted, the accepter nodes reach a majority and accept the host

Part 3 - failure handling

- When a client and host are streaming and the host fails, the client will find a new host
- When a client and host are streaming and the client fails, the host becomes available and can receive a new client
- Race conditions: when two clients try to pair with the same host, one of the clients will receive an error and the other will successfully connect with its chosen host

ShiViz Diagram



Since our implementation for finding the ideal host for a client uses a flooding protocol, our ShiViz diagram doesn't aid very much in understanding what our network is doing. It does however make it clear which nodes are connected to each other and that each node is being reached and responding during flooding.

Currently in Progress

Mouse inputs are currently being captured, but are not being reproduced as inputs on the remote machine. We also intend to have host machines log the amount of time they were in use for by a client and flood these logs to other hosts in order to maintain a distributed ledger. This can be used by hosts or clients as a resilient form of usage tracking.

Instructor Testing

When looking at our code repository, you'll find the majority of our network code in 3 files:

- **node.go**

  This file is our network library and handles all RPC calls sent and received by the node (both host and client). A node is initialized using the Initialize() method, which connects the node to the nodes specified in the json file located in the tests directory. To connect a client node to a best host, a call is made to waitForBestHost(), which floods the request down to connected nodes, waits a set period of time for nodes to respond to the open UDP port, and then picks the node with the lowest latency as the proposed host. This proposed host is then flooded again to nodes in the network to agree on whether this host is acceptable. Once a majority of hosts have agreed, a request is sent to the host to confirm the host and client connection.

- **clientStream.go**

  This file contains the ClientStream struct, which each node uses when sending and receiving data from the host. This struct is contained within the node struct and is called to initialize a connection with a host. The method ReceiveHostStream() is used to set ffplay to listen on the designated port and the method SendInputToHost() is used to send keystrokes and mouse events received from xinput to the host. When a failure is detected, this file has a channel which will stop all goroutines from running and signal the node to find a new host.
- **hostStream.go**

  This file contains the HostStream struct, which each node uses when sending and receiving data from the client. The method ReceiveInputFromClient() is used to receive keystrokes and mouse events from the client and then replicate them using xdotools. The method SendStreamToClient() is used to send the host screen to the client using ffmpeg. This file also has a channel to stop goroutines on failure.