CSCI 4220 Assignment 3

Assignment 3: Mobile Sensor Network Relay

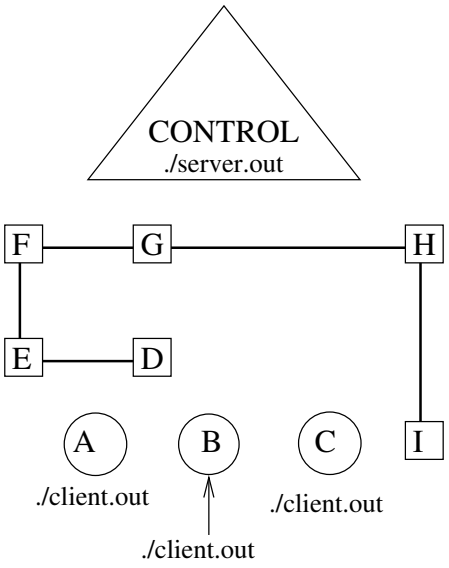
Due Date: Tuesday, November 12th, 11:59:59 PM

This is a team-based (max. 2) assignment.

Overview

Consider a mobile sensor application running over TCP with the topology (layout) below. The figure is drawn to scale. Each of the circles (`A`,`B`,`C`) represents a "sensor" which can move around in 2-D space (x and y coordinates) and runs your client application, **client.out** The squares represent "base stations" (`D`,`E`,`F`,`G`,`H`,`I`) which do not run any programs and do not move. Finally, the large triangle (`CONTROL`) is an abstract "control server" (it has no position and can reach all base statiosn and all clients) that will run your server application **server.out** . The control server will be used to relay TCP messages between sensors (which never directly communicate) and to manage messages that are moving between base stations (since base stations do not have any code and do not use sockets).

Every base station or sensor has a unique ID consisting of only letters and numbers. The control server will always have an ID of `CONTROL`. You can assume we will not provide invalid IDs or duplicate IDs.



Every sensor has a maximum communication range, and sensors can only send or receive messages if **both** the sender and receiver can reach each other (have communcation ranges greater than or equal to the distance between sender and receiver). However, instead of directly sending to or receiving from another sensor/base station, they will always communicate to/from the server. If a base station wants to send a message to a sensor, then the control server will be the sender and the sensor will be the receiver. Base stations can only

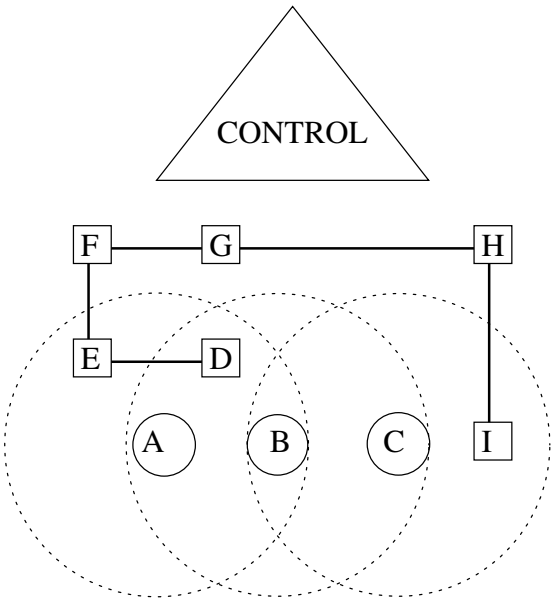
communicate with other base stations that are connected - this is represented through a "base station file" which is described later in this handout. When communicating with sensors base stations are assumed to have a range of infinity.

If a base station or sensor has a message it wants to send, it decides where to send it based on a three step decision process. First if the intended recipient (who the message is addressed to) is in range (meaning both the sender and destination can reach each other directly), then the message is sent to the recipient. Otherwise, the sender sorts all in-range sensors and base stations by distance to the destination (using the "Cartesian distance"). The sender then sends the message to the base station or sensor with the lowest distance to the intended recipient (assuming this would not cause a cycle, see below), resolving ties by choosing the lexicographically smaller ID.

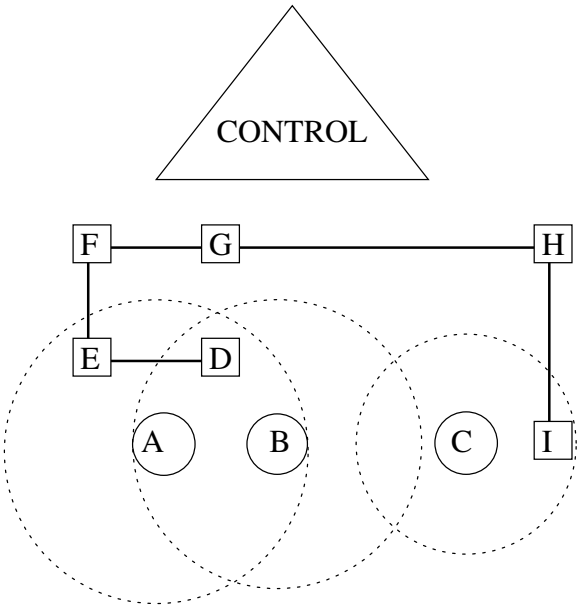
You can assume that we will not give any incorrectly formatted input, and we will not give any IDs that do not exist. You can also assume that clients will not disconnect until the end of the program. You may use any of the techniques we have discussed in this class so far: `fork()`, `select()` and `pthreads` are allowed, as is use of the book library (**unp.h** will be provided during compilation).

In summary, the control server listens for incoming connections, processes commands given via standard input, and reacts to messages from sensors. (Base station to base station communication can be done without any sockets since it's all in the same ./server.out process) The sensors running (./client.out) process commands given via standard input, and react to messages received from the control server.

Examples of Communication



In the diagram shown above, a message from `B` to `C` can be directly delivered (so it will be sent from `B` to `CONTROL` to `C`). A message from `A` to `C` cannot be directly delivered, since that would require `A` and `C` being in range of each other. Instead, `A` will pass the message to `B`, and `B` will pass it to `C`. Recall that sensors do not directly communicate with each other, so the message will actually travel from `A` to `CONTROL` to `B` to `CONTROL` to `C`.



The diagram above pictures a slightly different topology in which even `B` cannot directly reach `C`. If `B` wants to send a message to `C`, it will first send to `D` (which is the reachable sensor/base station that is closest to `C`). `D` will send to `A` (sending back to `B` would cause a cycle, and `A` is slightly closer than `E` is to `C`). `A` will have no choice but to send to `E` (anything else would cause a cycle). `E` will send to `F` (also because anything else will cause a cycle), and the remaining steps follow similarly: `F` to `G` to `H` to `I` to `C`.

However, because sensors do not directly communicate, the actual path is:
`B` to `CONTROL` (handling `D`) to `A` to `CONTROL` (handling `E` to `F` to `G` to `H` to `I`, no socket communication needed!) to `C`.

Base Station File Format

The base station file contains the following information in each row (ending in a newline character `\n`), with each field separated by a space:

- `[BaseID] [XPos] [YPos] [NumLinks] [ListOfLinks]`
- `[BaseID]` is the unique string identifier for the base station, and only contains letters and numbers.
- `[XPos]` is a floating point number representing the x-coordinate of this base station
- `[YPos]` is a floating point number representing the y-coordinate of this base station
- `[NumLinks]` is an integer that is the degree of the base station (i.e. how many other base stations it is connected to)
- `[ListOfLinks]` is a space-separated list of base station IDs that this base station is connected to. All connections are undirected, (i.e. BaseA is connected to BaseB iff BaseB is connected to BaseA).

You can assume that from any base station it is possible to reach all other base stations eventually, though it may require using multiple base stations along the way. In graph theory words, the graph is all one giant component.

Message Formatting

Messages that are being sent to another base station/sensor will look as follows, with each field separated by a space:

- `DATAMESSAGE [OriginID] [NextID] [DestinationID] [HopListLength] [HopList]`
- `[OriginID]` is the ID of the ~~node~~ base station/sensor that initially sent the data message.
- `[NextID]` is the ID that the data message is currently being sent to.
- `[DestinationID]` is the ID of the final destination that the data message is trying to be sent to.
- `[HopListLength]` is the length of `[HopList]`
- `[HopList]` contains the IDs that have been used so far to deliver the data message

Control Server Arguments, Input and Messages

The server is invoked by running: `./server.out [control port] [base station file]`

- `[control port]` is the port the server should listen on for sensors to connect to.
- `[base station file]` is the name of the file the server should read to get information about base stations and the base station network.

The following commands can be read from standard input:

- `SENDDATA [OriginID] [destination ID]` causes a new DATAMESSAGE to be created. If the `[OriginID]` is `CONTROL` then when deciding the next hop, all base stations should be considered reachable, and the `[NextID]` must be a base station. If the `[OriginID]` is a base station, then the next hop should be decided based on what is reachable from the base station with that `[BaseID]`. The `[OriginID]` will never be a sensor in this command.
- `QUIT` causes the server to clean up any memory and any sockets that are in use, and then terminate.
- **(updated 11/1) :** If the server receives a `WHERE` message from a sensor, it should respond with a message in the following format:
  `THERE [NodeID] [XPosition] [YPosition]` where `[XPosition]` and `[YPosition]` are the x and y-coordinates of the requested base station/sensor and `[NodeID]` is its ID. Remember that we are assuming all IDs provided are valid.
- If the server receives an `UPDATEPOSITION` message from a sensor, it should respond with a message in the following format: `REACHABLE [NumReachable] [ReachableList]`
  where `[ReachableList]` contains all sensor and base station IDs and positions that are currently reachable by the sensor, and `[NumReachable]` is the number of IDs in `[ReachableList]`. Each entry in ReachableList is actually 3 strings: `[ID] [XPosition] [YPosition]`. Entries are separated by a space.
- If the server receives a data message with a ~~matching~~ `[NextID]` that is a sensor's ID, the server is acting as a sensor-to-sensor relay, and it should deliver the message to the destination. **(updated 11/3)**

If a base station receives a data message (either the server receives a message, or we are internally passing a message from one base station to another base station):

- If the `[DestinationID]` matches its `[BaseID]` it should print the following to standard output: `[BaseID]: Message from [OriginID] to [DestinationID] succesfully received.`
- Otherwise, if a base station with ID `[BaseID]` receives a data message (starts with `DATAMESSAGE`) with a `[DestinationID]` that is not its own ID, and all reachable sensors/base stations are already in the `[HopList]`, the data message should be discarded and the following should be printed to standard output: `[BaseID]: Message from [OriginID] to [DestinationID] could not be delivered.`

- Otherwise, the base station should send a new `DATAMESSAGE` command to the next sensor/base station based on the decision-making described in the "Overview" section. This `DATAMESSAGE` should look the same, but the `[NextID]` should reflect the new next hop, `[HopList]` should also contain the `[BaseID]` of the base station and the `[HopListLength]` should be increased by 1. A message should be printed to standard output:
  `[BaseID]: Message from [OriginID] to [DestinationID] being forwarded through [NextID]`
- **(new 11/3) :** If the `[OriginID]` is the current base station, and the `[NextID]` and `[DestinationID]` match, then the base station should print the following message to standard output:
  `[BaseID]: Sent a new message bound for [DestinationID].`

Sensor (Client) Arguments, Input and Messages

Sensors invoke the client code by running: `./client.out [control address] [control port] [SensorID] [SensorRange] [InitalXPosition] [InitialYPosition]`

- `[control address]` is the address or hostname the control server can be reached at
- `[control port]` is the port the control server will be listening on for
- `[SensorID]` is the ID of the sensor and only contains letters and numbers.
- `[SensorRange]` is an integer value for the maximum communication range of the sensor
- `[InitalXPosition]` is the initial x-coordinate of the sensor
- `[InitialYPosition]` is the initial y-coordinate of the sensor

When the sensor starts up, before processing any standard input it should first connect to the control server and send an `UPDATEPOSITION` message (see below). The following commands can be read from standard input:

- `MOVE [NewXPosition] [NewYPosition]` causes the sensor to update its location to the x-coordinate specified by `[NewXPosition]` and the y-coordinate specified by `[NewYPosition]`. The sensor should also send an `UPDATEPOSITION` command to the server.
- `SENDDATA [DestinationID]` causes the sensor to generate a new `DATAMESSAGE` with a destination of `[DestinationID]`. The sensor should first send an `UPDATEPOSITION` message to the server to get an up-to-date list of reachable ~~nodes~~ sensors and base stations.
- `QUIT` causes the client program to clean up any memory and any sockets that are in use, and then terminate.
- **(new 10/31) :** Sensors can send a `WHERE` message to the control server:
  `WHERE [SensorID/BaseID]` to get the location of a particular base station or sensor ID from the control server. It should not take any other actions until it gets a `THERE` message back from the server.

Sensors can send an `UPDATEPOSITION` message to the control server:
`UPDATEPOSITION [SensorID] [SensorRange] [CurrentXPosition] [CurrentYPosition]` where the positions are the current x and y coordinates of the sensor. After sending an `UPDATEPOSITION` message, the client should not do anything else until it receives a `REACHABLE` message from the server.

If a sensor receives a `DATAMESSAGE` from the control server:

- If the `[DestinationID]` matches its `[SensorID]` it should print the following to standard output:
  `[SensorID]: Message from [OriginID] to [DestinationID] succesfully received.`
- Otherwise if a sensor with `[SensorID]` receives a data message with a `[DestinationID]` that is not its own ID, and all reachable sensors/base stations are already in the `[HopList]`, the message should be discarded and the following message should be printed to standard output: `[SensorID]: Message from [OriginID] to [DestinationID] could not be delivered.`
- Otherwise, the sensor should first send an `UPDATEPOSITION` message to `CONTROL` (even it hasn't moved) so that it gets an updated list of reachable ~~nodes~~ sensors and base stations. **(new 10/31) :**

The sensor should next send a WHERE message to the server requesting the current position of the `[DestinationID]`. The sensor should then send a new `DATAMESSAGE` command to the next sensor/base station based on the decision-making described in the "Overview" section. This `DATAMESSAGE` should look the same, but the `[NextID]` should reflect the new next hop, `[HopList]` should also contain the `[SensorID]` of the sensor and the `[HopListLength]` should be increased by 1. A message should be printed to standard output:
`[SensorID]: Message from [OriginID] to [DestinationID] being forwarded through [NextID]`

- **IMPORTANT**: Sensors will never directly talk to each other. If a sensor wants to pass a message to another sensor, it will update the `[NextID]` field in the `DATAMESSAGE` to be the sensor it wants to pass the message to, and send the message to `CONTROL`. **(updated 11/1)**
- **(new 11/3) :** If the `[OriginID]` is the current sensor, and the `[NextID]` and `[DestinationID]` match, then the sensor should print the following message to standard output:
`[SensorID]: Sent a new message bound for [DestinationID].`

Submission

You can provide any decisions or other comments for the graders to read in a **README.txt** file. Like Assignments 1 and 2, you should submit a **Makefile**, but this time you will produce two files: **server.out** and **client.out** using C or C++. Also like the previous assignments, we will provide libunp.a, unp.h, and config.h in the same directory as your files are built, and to ensure compilation if using these features you should have libunp.a in your compile line and should use clang instead of gcc.

We will still just invoke `make` so your Makefile should look something like this:

```
default: server.out client.out

client.out: client.c
    gcc client.c -o client.out

server.out: server.c
    gcc server.c -o server.out
```

The filename(s) after the colon show which files will be needed in order to build the file on the left side of the colon, and the indented lines specify commands to run in order to create the filename to the left of the colon, so adjust as needed.