Assessment 2 component: Overseer

Summary

The overseer acts as a management service and administration console for the entire building's systems. It is used to connect controllers together - for example, card readers and doors, or destination select panels and elevators - and perform other logic that requires having a whole-of-system perspective. It is also responsible for security and authorisation. Finally, through a text-based interface it allows users to manually control certain systems like doors and cameras.

Program name

```
overseer
```

Command-line arguments

```
{address:port} {door open duration (in microseconds)} {datagram resend delay (in microseconds)} {aut
horisation file} {connections file} {layout file} {shared memory path} {shared memory offset}
```

Shared memory structure

```
struct {
    char security_alarm; // '-' if inactive, 'A' if active
    pthread_mutex_t mutex;
    pthread_cond_t cond;
};
```

Initialisation

On startup, this component will bind its port on both TCP and UDP, and listen on the TCP port for incoming connections.

Normal operation

The overseer is in charge of many systems and needs to do several things at once, so use of threads is almost mandatory. For example, you will probably want a thread to deal with incoming TCP connections, a thread to deal with UDP datagrams (from temperature sensors), and other threads to spin off as necessary. This section is broken up into subsections for the different things the overseer is in charge of.

Component initialisation

Most of the components in the system will send a message to the overseer after they start up, so the overseer can keep track of which components are currently part of the system. Components that do so are:

- **Card reader controller** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-card-reader-controller) - `CARDREADER {id} HELLO#`
- **Door controller** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-door-controller) - `DOOR {id} {address:port} {FAIL_SAFE | FAIL_SECURE}#`
- **Fire alarm unit** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-fire-alarm-unit) - `FIREALARM {address:port} HELLO#`
- **Camera controller** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-camera-controller-2) - `CAMERA {id} {address:port} HELLO#`
- **Elevator controller** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-elevator-controller-2) - `ELEVATOR {id} {address:port} HELLO#`
- **Destination select controller** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-destination-select-controller) - `DESTSELECT {id} HELLO#`

This provides the address and port that the component can be contacted via, and for most of these an identification number as well. Note that doors are special - instead of sending HELLO, they will send FAIL_SAFE or FAIL_SECURE, depending on how the door is configured. This needs to be kept track of as well - doors marked FAIL_SAFE need to be sent to the fire alarm unit for registration so that it can open them in the case of an emergency, while doors marked FAIL_SECURE may need to be closed permanently in the case of a security breach.

Once the fire alarm unit connects, the overseer must send it a UDP datagram for every FAIL_SAFE door that has been initialised so far. (See '"DOOR": Door registration datagram' on the page for **Assessment 2 component: Fire alarm unit** (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-fire-alarm-unit?wrap=1)

for the format of this datagram.) If new FAIL_SAFE doors are initialised after the fire alarm unit has been, send datagrams for those as well. If the fire alarm unit does not respond to a given door registration datagram within {datagram resend delay} microseconds with the appropriate confirmation diagram, send it again.

Door access

When a card reader sends a SCANNED message via TCP (see **Card reader controller (https://canvas.qut.edu.au/courses/14485/pages/assessment-2-component-card-reader-controller)** for details), the overseer needs to do the following:

- Look up the scanned code in the authorisation file (see 'Authorisation file' below) to determine which door IDs the user has access to
- Look up the card reader's ID in the connections file (see 'Connections file' below) to determine which door ID this card reader corresponds to
- If the scanned code is in the authorisation file and if it has access to this door:
  - Reply to the card reader controller with ALLOWED#
  - Close the connection
  - Open a connection to the door controller (which should have previously initialised, so the overseer knows its IP address and port)
  - Send an OPEN# message
  - If the door controller responds with OPENING#
    - Wait until the door controller responds with OPENED#
    - Close the connection
    - Wait for {door open duration} microseconds
    - Open the connection again and send CLOSE#
- If the scanned code does not have access to this door, send DENIED# instead and close the connection

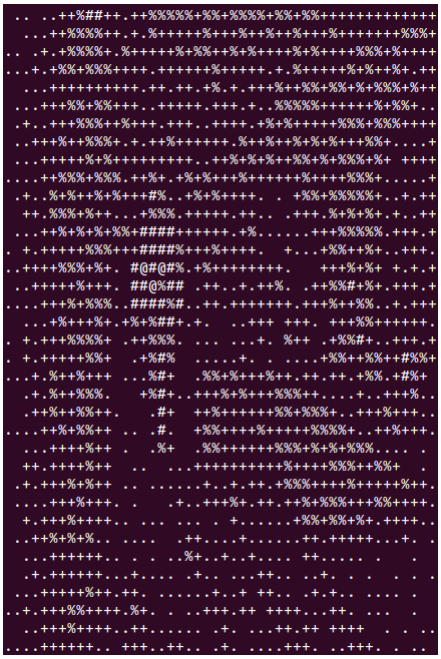
(Note that this may happen simultaneously with other events, including other doors opening and closing, which is one reason why multiple threads are important for the overseer.)

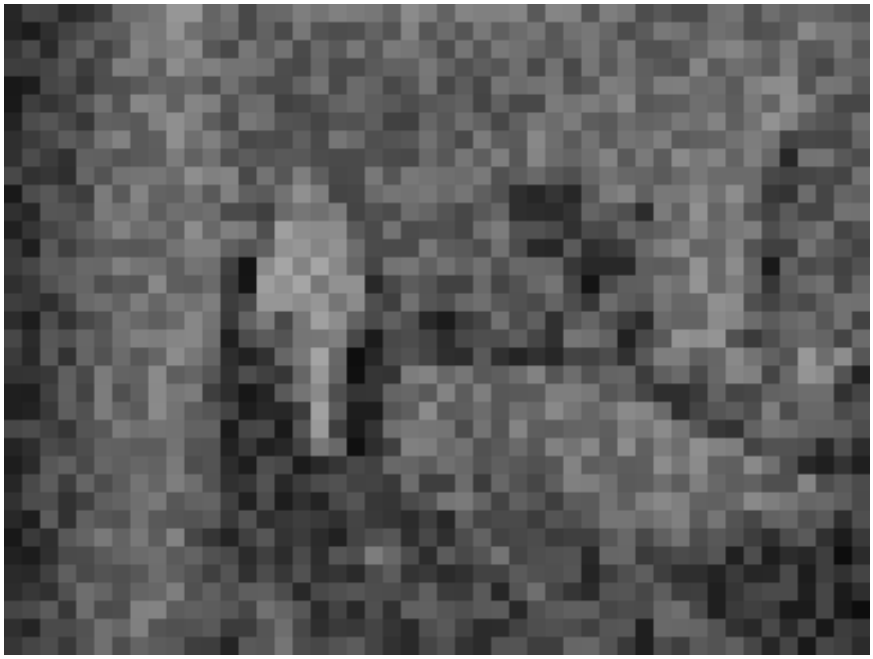
Manual access

The overseer will present a prompt to the user, allowing the user to type commands in to manually control certain parts of the building. Commands will correspond to a particular component type - for instance, all commands relating to doors will begin with DOOR.

- **DOOR LIST**
  Display a list of doors in the system- their IDs, IP addresses, ports and whether they are fail-safe or fail-secure. The exact formatting of this does not matter as long as the information is there

- **DOOR OPEN** *{id}*
  Opens the door with the given id. The door will not close automatically if opened this way.
- **DOOR CLOSE** *{id}*
  Closes the door with the given id. This may be used to close doors that have been opened either with DOOR OPEN {id} or through a user manually accessing them.
- **TEMPSENSOR LIST**
  Displays a list of (known) temperature sensors and the most recent temperature they recorded. Note that temperature sensors do not initialise, so the overseer will only know about a temperature sensor from when it first send an update.
- **CAMERA LIST**
  Display a list of cameras in the system- their IDs and IP addresses. The exact formatting of this does not matter as long as the information is there
- **CAMERA ON** *{id}*
  Turns the camera with the given id on.
- **CAMERA OFF** *{id}*
  Turns the camera with the given id off.
- **CAMERA ANGLE** *{id} {angle}*
  Tells the camera with the given id to turn to the given angle.
- **CAMERA TRACKING** *{id}*
  Tells the camera with the given id to begin tracking (panning left to right and right to left continuously.)
- **CAMERA DISPLAY** *{id}*
  Print out a depiction - as best as you can through text - of the current frame being captured by that camera. Hint: these characters (`.+%#@`) are approximately ordered by brightness, so you can map the pixels returned from the camera into this range and get something that looks similar to this:





(If you want to get particularly fancy, you can look up **terminal formatting escape sequences (https://www.nayab.dev/linux/misc/escape-codes)** and use foreground/background colour to make the render more faithful to the original image, but this is not required.)

- **FIRE ALARM**
  Issuing this command will cause a fire alarm by sending an appropriate UDP datagram to the fire alarm unit - as with the call-point controller, the datagram should be resent every {datagram resend delay} microseconds.
- **SECURITY ALARM**
  Issuing this command will manually raise a security alarm, involving making the write to shared memory, signalling the condition variable and closing all the fail-secure doors - see below for the exact steps required.
- **EXIT**
  This command will cause the overseer to terminate

The output to display from any of these commands is up to you - you can display a message showing that issuing the fire alarm or opening the door was successful, or that the door was already open depending on the response received from the controller etc. You may also add additional commands to the overseer.

Temperature monitoring

UDP packets will occasionally come in with temperature updates - check the timestamp to make sure that you are storing the most recent temperature you have for each temperature sensor. These values are then displayed when issuing the TEMPSENSOR LIST command.

Elevator control

Similarly to card readers and doors, the overseer is in charge of managing the interactions between destination select controllers and elevator controllers, although the overseer's access to the elevator is less direct - the overseer simply tells the elevator which floors to go to and the elevator is in charge of its own queuing.

When a destination select controller sends a `DESTSELECT {id} SELECT {scanned} {floor}#` message, the overseer will look for the following information from the authorisation and connections files, specifically:

- From the authorisation file - which floors does this user have access to (you need to have access to the destination floor but not the floor you're calling the elevator from)
- From the connections file - what floor is this destination select panel on?
- Also from the connections file - what elevator is this destination select panel associated with?

Then, if access was granted, the overseer will reply to the destination select controller with `ALLOWED#` and tell the elevator to take the user from the floor containing that destination select panel to the requested destination floor with a `FROM {from floor} TO {destination floor}#` message. If access was not granted, reply with DENIED# instead.

Camera surveillance

While the cameras can be analysed manually via the command interface, they will also automatically report when motion is detected. This can be used to automatically detect potential security breaches.

While the overseer does not have perfect knowledge of where everyone is, it does have partial knowledge, and this is enough for detecting certain types of security breaches. Employees, contractors and guests all know that they are **required** to always swipe through an access door or elevator, even if they are with someone else. As a result, we know when we detect a card being used (on a card reader or destination select panel):

- That the user with that particular card is in the location of the card reader / destination select panel
- That the user **may** move to the location they just opened up (either the other side of the door or the elevator location on the destination floor)

This means that, for a given user, we can know of **two locations** where that user might be (with authorisation). This means that, if a camera detects motion in any location that no user could possibly be, there is a potential security breach and the security alarm needs to be raised. The layout and connections files contain the necessary information for determining where cameras, card readers and destination select panels are and where security doors and elevators lead to.

(Note that, to simplify this logic, we are assuming that a user will not hit multiple buttons on an elevator and then pick one the floors to go to.)

Raising the security alarm

In the case of either a user manually raising a security alarm from the overseer's prompt **or** unauthorised motion being detected automatically, the overseer will raise a security alarm by taking the following steps:

- Lock the shared memory mutex
- Set 'security_alarm' to A
- Unlock the mutex
- Signal the condition variable
- Loop through every FAIL_SECURE door, sending each one the CLOSE_SECURE# message. This will lock that door permanently (until the security team arrives to unlock it and reset the system.)

Authorisation file

The authorisation file is a plain text file that is passed to the overseer and includes which floors and security doors each user has access to. Due to privacy concerns, no personal information appears in the file - just the user's access card code and the floors and doors the user has access to.

Each line has the following format

```
{access code} {DOOR:# | FLOOR:# ...}
```

For example:

```
db4ed0a0bfbb00ac DOOR:101 DOOR:201 FLOOR:1 FLOOR:2 FLOOR:3
cae0efe252087308 DOOR:101 DOOR:301 FLOOR:1 FLOOR:3
4f1d3f68e4ca6d80 FLOOR:1 FLOOR:2 FLOOR:4
2214a7ba5943d923 DOOR:101 DOOR:102
4b6f9c1d4d55506c DOOR:501 FLOOR:1 FLOOR:5
```

In the above example the user with the code `db4ed0a0bfbb00ac` can get through the security doors with IDs 101 and 201 and can use any elevator to get to floors 1, 2 and 3.

Connections file

The connections file indicates which card readers open which doors and which destination select panels are associated with which elevators (and which floors the panels are on).

It is a plain text file consisting of a line for each connection, and there are two different types of lines that may appear in it:

```
DOOR {card reader id} {door id}
ELEVATOR {destination select id} {floor number} {elevator id}
```

Card readers have a many-to-one relationship with doors - a given door can be opened by multiple card readers (usually two- one on each side), but a given card reader will only open one door. The same is true for elevators- one elevator can have multiple destination select panels (usually one per floor) but a single destination select panel will only ever call one elevator.

Here is an example of a possible connections file:

```
DOOR 101 101
DOOR 102 101
DOOR 103 102
DOOR 104 102
ELEVATOR 101 1 1
ELEVATOR 201 2 1
ELEVATOR 301 3 1
ELEVATOR 202 2 2
ELEVATOR 302 3 2
```

In the above example, card readers 101 and 102 both open door 101 while card readers 103 and 104 both open door 102. There are two elevators- 1 and 2. Elevator 1 has destination select

panels 101, 201 and 301 on floors 1, 2 and 3 respectively, while elevator 2 has destination select panels 202 and 302 on floors 2 and 3 respectively.

Layout file

The layout file divides the building up into numbered **sectors**. Each **sector** is a contiguous space within the building (which may consist of multiple rooms, corridors etc.) such that moving through either a **secure door** or an **elevator** is necessary to move to another sector.

Each line of the layout file has one of the following formats:

```
CARDREADER {id} {sector}
CAMERA {id} {sector}
DESTSELECT {id} {sector}
```

- Card readers, cameras and destination select panels each occupy a single sector.
- Doors are not listed here because doors exist between sectors and are considered to connect the sectors where the corresponding card readers are located - if, for example, door 101 can be opened by card readers 101 and 102, and door 101 is in sector 0 while door 102 is in sector 1, someone opening door 101 by swiping their card at card reader 101 is considered to be in either sector 0 or 1 after this.
- Elevators are not listed here because it is assumed that if you take a particular elevator to a certain floor, the floor you end up on will have the destination select panel for that elevator on that floor.

Here is an example of a possible layout file:

```
CARDREADER 101 0
CARDREADER 102 1
CARDREADER 103 1
CARDREADER 104 2
DESTSELECT 101 1
DESTSELECT 201 3
DESTSELECT 301 4
CAMERA 101 2
CAMERA 201 3
```

In this example, assuming the connections from the earlier connections file example apply here, there are 5 sectors- 0 through 4. A user can get from sector 0 to sector 1 by swiping their card through card reader 101, and can then either swipe their card through card reader 103 to get to sector 2 or use the destination select panel in sector 1 to access sector 3 or 4 (assuming the user has the correct authorisation).

Note that sector 0 is a little bit special - your code does not need to take this into account at all, but sector 0 acts as a foyer area anyone can access. Nobody in sector 0 is considered trespassing and there are never any security cameras in sector 0.