SOEN 423, Fall 2024, Assignment 1

Ties Schasfoort

1st October 2024

Overall design:

For this assignment I have decided to use the Java RMI architecture to implement client-server communication. I used this since it made the most sense after attending the lectures and gaining knowledge of it. It is used to initiate the 3 servers and when a client is initiated, it will be connected to one of these 3 depending on their client ID.

For the inter-server communication, I have used UDP/IP. Since the servers have to listen to other servers constantly in case they redirect a request from a client from the same district, I have used UDP sockets.

Design per method:

addResource:

Steps:

1. The client passes the resourceID, resourceName, and duration to the server invoking the addResource method. After this, steps 2-4 are executed by the server.
2. Check resource prefix: The method checks whether the resource's prefix matches the current server's prefix. If the resource belongs to a different server, the request is forwarded using UDP to the appropriate server.
3. Add or update resource: If the resource's prefix matches the current server, the method checks if the resource already exists. If the resource is new it gets added in the resourcesServer map. If the resource exists but with a lower duration, the method updates the resource's duration by increasing it. If the resource exists but the duration is higher, the operation failed since the resource couldn't be added nor updated.
4. Logging: After processing, the method logs the operation to the server's list and after returning to the client the logfile will be added to theirs as well, including the current timestamp, request name, passed arguments, and the success or failure status of the operation. This is a general step taken after every method, so in the upcoming methods I just refer to it as "Logging".

Data structures used:

1. resourcesServer (Map<String, Map<String, Integer>>): This is the primary data structure where all resources of the local server are stored. The outer map's key is the resource name, and the value is another map. This inner map has the resource ID as the key and the resource duration as the value.
2. clientResources (ConcurrentHashMap<String, Map<String, Map<String, Integer>>>): Tracks all the resources held by clients. It maps each client ID to the resources they hold. The inner structure is similar to resourcesServer, mapping resource names and IDs to durations.
3. logfilesServer (List<List<String>>): A list of log entries that track operations performed on the server. Each log entry is a list containing details such as the timestamp, operation type, resource details, the outcome, and the server's response to the client's request.
4. resourceWaitQueue (ConcurrentHashMap<String, Queue<String>>): Stores the waiting clients for a particular resource when it's unavailable. Each resource ID maps to a queue of clients waiting to access that resource.
5. currentClientId (ThreadLocal<String>): Stores the client ID for each thread to ensure thread safety when dealing with concurrent requests from multiple clients.

6. LogfilesClient (List<List<String>>): list used for storing the client's logfiles.

removeResource:

Steps:

1. The client enters the ID of the resource they want to remove. After this they choose whether they want to remove the resource entirely or decrease its duration. The resourceID and duration arguments are passed to the server.
2. Check resource prefix: just like addResource check whether the server needs to let another server perform the method and return its results to the server that got the request.
3. Remove resource or decrease duration: depending on whether the client decided to decrease the duration or remove the entire resource, go through the resources which the server holds and perform this operation. Note: only decrease the duration if the passed duration is <= the duration of the resource held by the server to avoid negative durations which would make no sense.
4. Logging

Data structures used:

1. resourcesServer (Map<String, Map<String, Integer>>)
2. logfilesServer (List<List<String>>)
3. LogfilesClient (List<List<String>>)

listResourceAvailability:

Steps:

1. The client enters the type of resource they want to list (ambulance, firetruck, or personnel). This argument is passed to the server.
2. The server creates an empty string called "resourceAvailability" and first adds the resourceIDs with the durations to the string which that server contains.
3. The server then communicates with the other servers by sending a UDP message to them containing the method name, after which these two servers return the concatenated string of their held resources.
4. The originating server adds the returned strings to resourceAvailability and returns it to the client if it is not empty. Else it will show a message saying there aren't any resources with the specified type.
5. Logging

Data structures used:

1. resourcesServer (Map<String, Map<String, Integer>>)
2. logfilesServer (List<List<String>>)

3. logfilesClient (List<List<String>>)
4. UDPPorts (ArrayList<Integer>): the originating server iterates through this list to get the right port numbers to pass to the sendUDPMessage method.

requestResource:

Steps:

1. The client enter the resourceID and the duration for which they want the specified resource. These arguments plus the clientID are passed to the server when invoking the method.
2. The server checks whether the request is meant for another server based on the resource prefix, if so, the request is sent to the corresponding server and the originating server waits for its response.
3. The server which should have the resource checks resourcesServer to see if it contains the requested resource.
4. If so, it checks whether the given duration is <= than the duration of the resource resourcesServer holds. If it is, the duration of the resource in resourcesServer is decreased by the amount the client requested. Then the resource is given to the client by adding the resource with the clientID to the clientResources hashMap or increase/decrease the duration of the resource if the client was already holding it. It would be decreased if the time held is higher than the time requested.
5. If the duration requested was higher than the duration available or the resource simply isn't available at all, the server returns the question whether the client would like to wait for the resource in a queue.
    a. The client then acts based on this prompt.
    b. If the client says yes, the server will add the clientID to the resourceID in the resourceWaitQueue. Whenever a resource with this ID is returned or added by another client, the server will give the resource to the waiting client at the head of the queue.
6. Logging

Data structures used:

1. resourcesServer (Map<String, Map<String, Integer>>)
2. logfilesServer (List<List<String>>)
3. logfilesClient (List<List<String>>)
4. clientResources (ConcurrentHashMap<String, Map<String, Map<String, Integer>>>)
5. resourceWaitQueue (Map<String, Queue<String>>): first string is the resourceID, then the queue contains the clientIDs of the clients waiting for the resource.

findResource:

Steps:

1. The client passes the resourceName (resource type) of which they want the resourceIDs with their duration displayed by the server (only those which they hold). Along with that they pass the clientID to the server.
2. The server then basically takes the same steps as for the "listResourceAvailability" method. However, instead of checking serverResources, each server checks clientResources.
3. The server returns the string "resourceHeldDetails" to the client, which is basically the same as "resourceAvailability" but then this is the string with the held resources by the client (with the specified resource type).
4. Logging

Data structures used:

1. clientResources (ConcurrentHashMap<String, Map<String, Map<String, Integer>>>)
2. logfilesServer (List<List<String>>)
3. logfilesClient (List<List<String>>)
4. UDPPorts (ArrayList<Integer>)

returnResource:

Steps:

1. The client gives the resourceID of the resource they want to return to the server. It then passes this argument, and the clientID to the server.
2. The server then checks whether this resource should be returned to them or to another server based on the resource's prefix.
3. The correct server then checks if the resource is already in its serverResources. If so, the duration of the returned resource is just added to the already existing resource in serverResources. If not, the resource is added to the map.
4. The server then checks whether another client is waiting for the resource and gives this to them if this is the case.
5. The server then removes the resource from the clientResources map which represents giving away/returning the resource.
6. Logging

Data structures used:

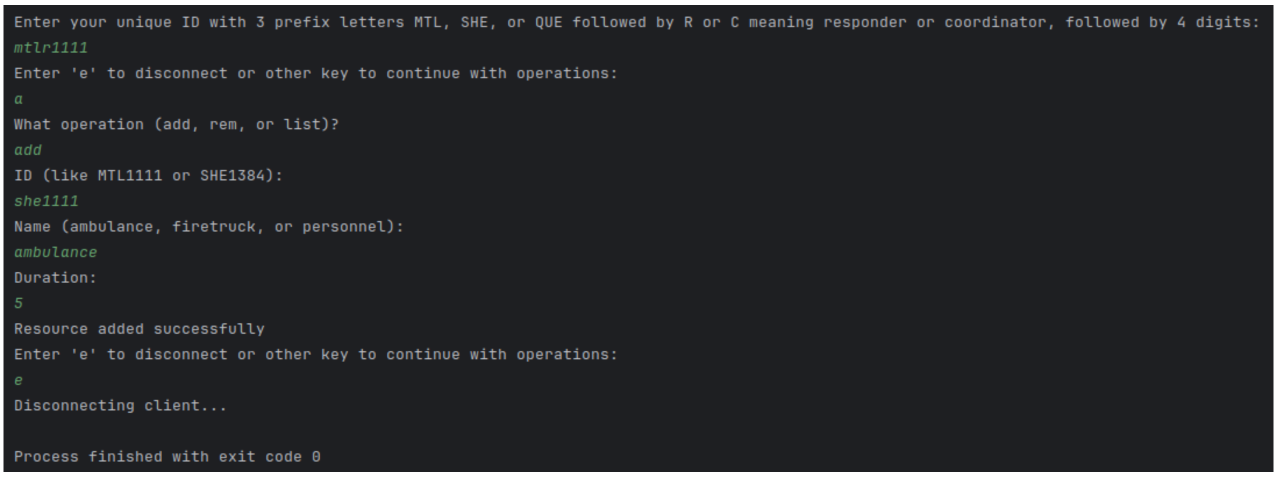
1. clientResources (ConcurrentHashMap<String, Map<String, Map<String, Integer>>>)
2. logfilesServer (List<List<String>>)

3. logfilesClient (List<List<String>>)
4. resourcesServer (Map<String, Map<String, Integer>>)
5. resourceWaitQueue (Map<String, Queue<String>>)

Test cases and edge cases per method:

AddResource:

Test case:

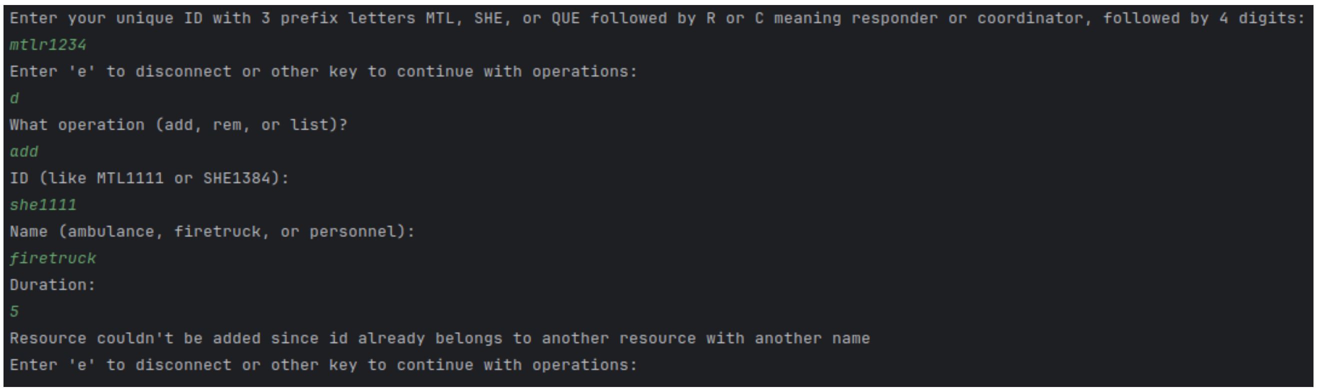


```
Enter your unique ID with 3 prefix letters MTL, SHE, or QUE followed by R or C meaning responder or coordinator, followed by 4 digits:
mtlr1111
Enter 'e' to disconnect or other key to continue with operations:
a
What operation (add, rem, or list)?
add
ID (like MTL1111 or SHE1384):
she1111
Name (ambulance, firetruck, or personnel):
ambulance
Duration:
5
Resource added successfully
Enter 'e' to disconnect or other key to continue with operations:
e
Disconnecting client...

Process finished with exit code 0
```

Above is an addition of a resource to a server by a client from another area.

Edge case:

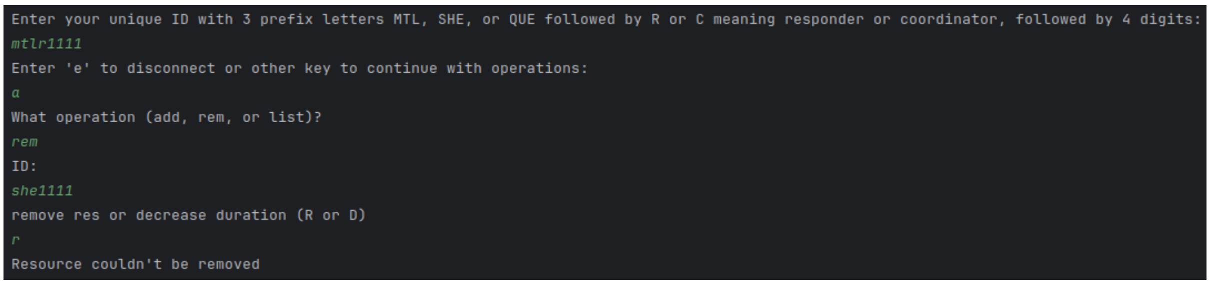


```
Enter your unique ID with 3 prefix letters MTL, SHE, or QUE followed by R or C meaning responder or coordinator, followed by 4 digits:
mtlr1234
Enter 'e' to disconnect or other key to continue with operations:
d
What operation (add, rem, or list)?
add
ID (like MTL1111 or SHE1384):
she1111
Name (ambulance, firetruck, or personnel):
firetruck
Duration:
5
Resource couldn't be added since id already belongs to another resource with another name
Enter 'e' to disconnect or other key to continue with operations:
```

Above shows what happens if another client tries to add a resource with the same id but another name to the server. This shouldn’t be possible to keep the id unique.

RemoveResource:

Test case:

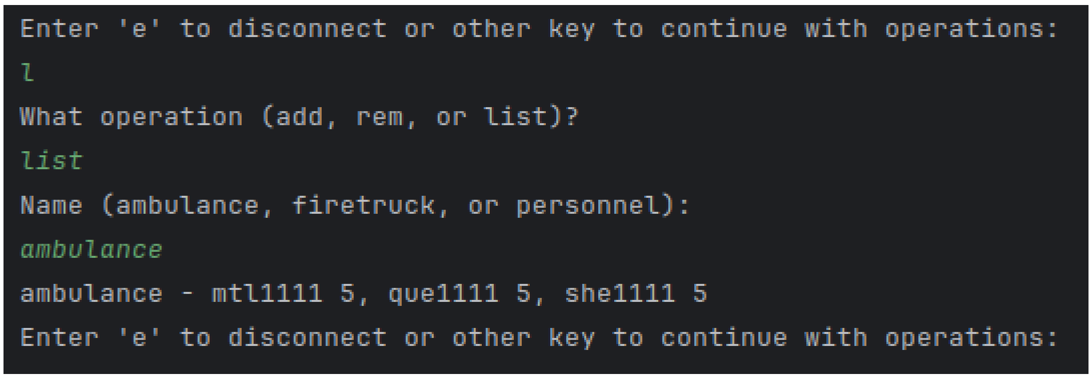


```
Enter your unique ID with 3 prefix letters MTL, SHE, or QUE followed by R or C meaning responder or coordinator, followed by 4 digits:
mtlr1111
Enter 'e' to disconnect or other key to continue with operations:
a
What operation (add, rem, or list)?
rem
ID:
she1111
remove res or decrease duration (R or D)
r
Resource couldn't be removed
```

Above shows what happens if client tries to remove non-existing resource.

ListResourceAvailability:

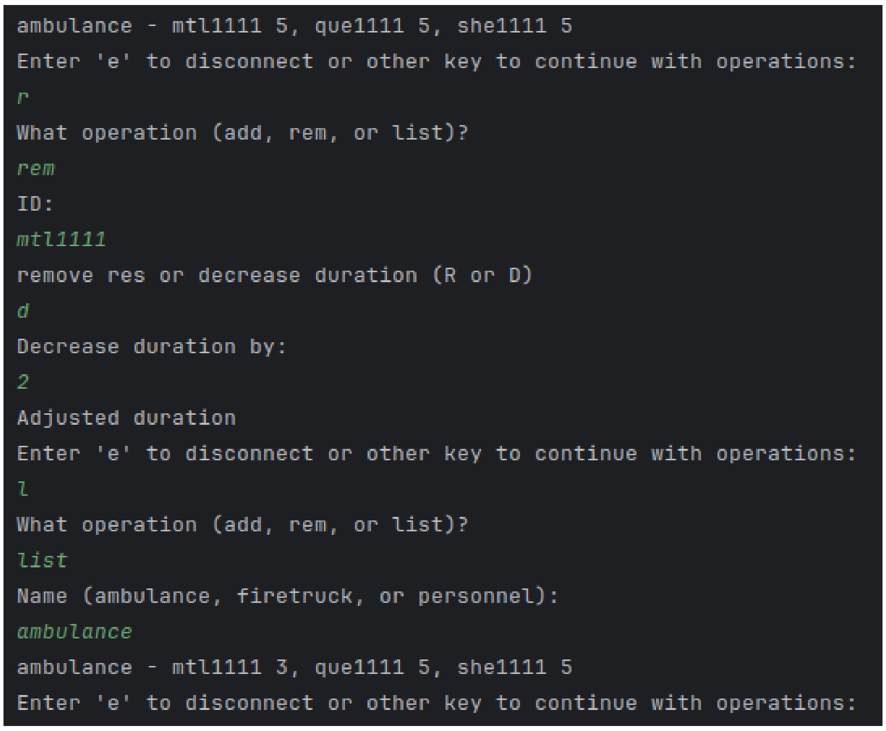
Test case:



```
Enter 'e' to disconnect or other key to continue with operations:
l
What operation (add, rem, or list)?
list
Name (ambulance, firetruck, or personnel):
ambulance
ambulance - mtl1111 5, que1111 5, she1111 5
Enter 'e' to disconnect or other key to continue with operations:
```

Above is the result after first adding 3 resources of type ambulance and then calling list on this type.

Edge case:

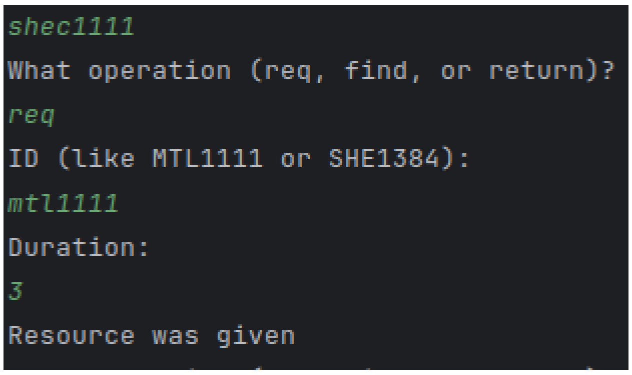


```
ambulance - mtl1111 5, que1111 5, she1111 5
Enter 'e' to disconnect or other key to continue with operations:
r
What operation (add, rem, or list)?
rem
ID:
mtl1111
remove res or decrease duration (R or D)
d
Decrease duration by:
2
Adjusted duration
Enter 'e' to disconnect or other key to continue with operations:
l
What operation (add, rem, or list)?
list
Name (ambulance, firetruck, or personnel):
ambulance
ambulance - mtl1111 3, que1111 5, she1111 5
Enter 'e' to disconnect or other key to continue with operations:
```

Above shows result of decreasing duration of an earlier added resource and then calling list on the resource type.

RequestResource:

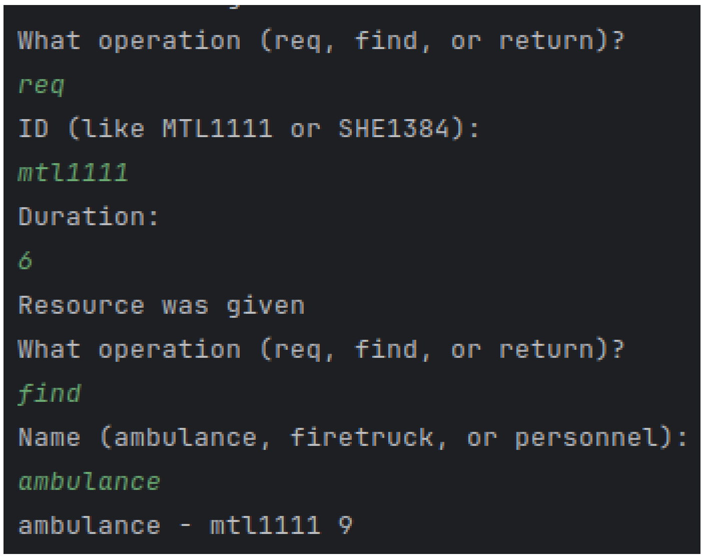
Test cases:



```
shec1111
What operation (req, find, or return)?
req
ID (like MTL1111 or SHE1384):
mtl1111
Duration:
3
Resource was given
```

Above shows request of resource from another server than the one directly connected to the client.

Edge case:

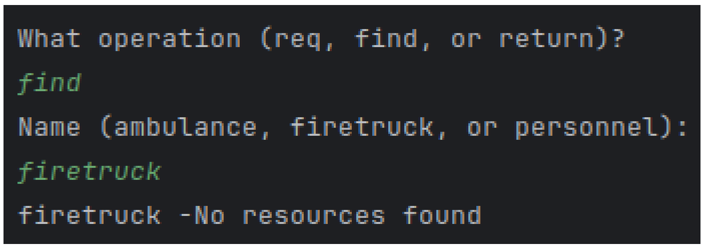


```
What operation (req, find, or return)?
req
ID (like MTL1111 or SHE1384):
mtl1111
Duration:
6
Resource was given
What operation (req, find, or return)?
find
Name (ambulance, firetruck, or personnel):
ambulance
ambulance - mtl1111 9
```

Above shows what happens if the client requests the same resource for a second time. Correctness is proven by using find to display that the client is holding the resource for duration 9 (3 from first + 6 from second request).

FindResource:

Test case:

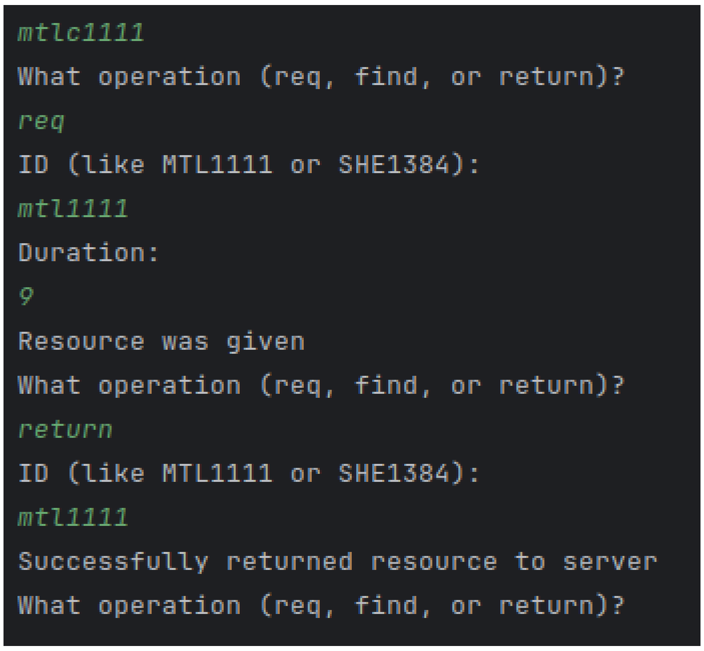


```
What operation (req, find, or return)?
find
Name (ambulance, firetruck, or personnel):
firetruck
firetruck -No resources found
```

Above shows what happens if client tries to find resource which they don't hold.

ReturnResource:

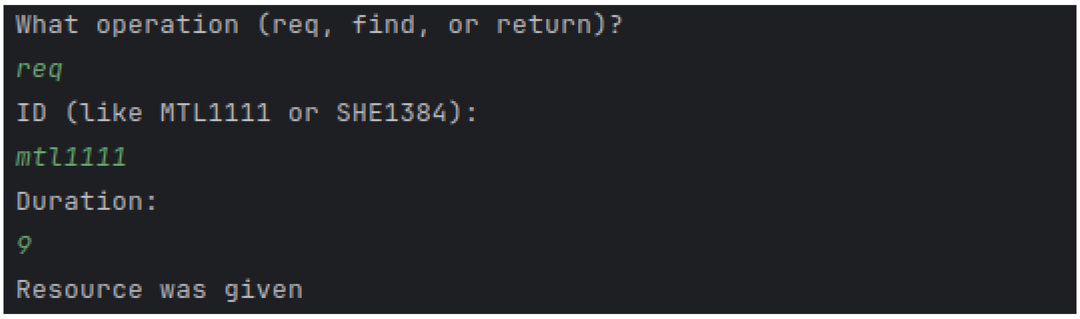
Test case:



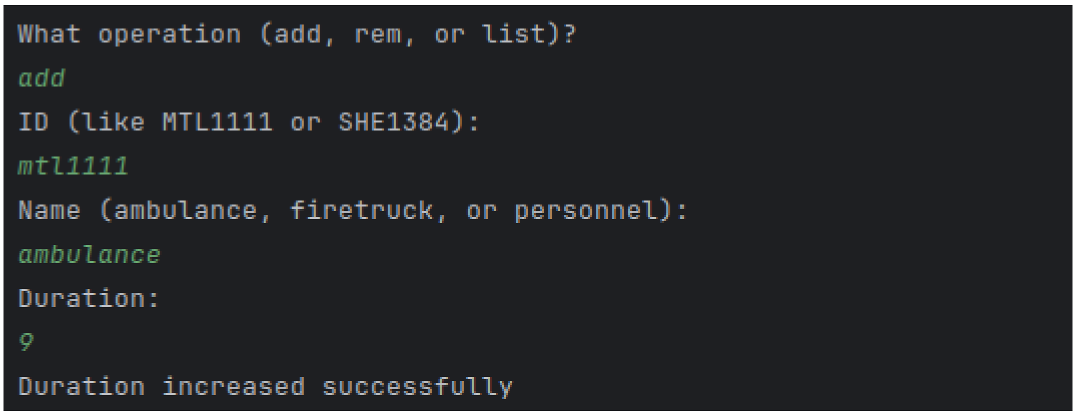
```
mtlc1111
What operation (req, find, or return)?
req
ID (like MTL1111 or SHE1384):
mtl1111
Duration:
9
Resource was given
What operation (req, find, or return)?
return
ID (like MTL1111 or SHE1384):
mtl1111
Successfully returned resource to server
What operation (req, find, or return)?
```

Above shows normal return case.

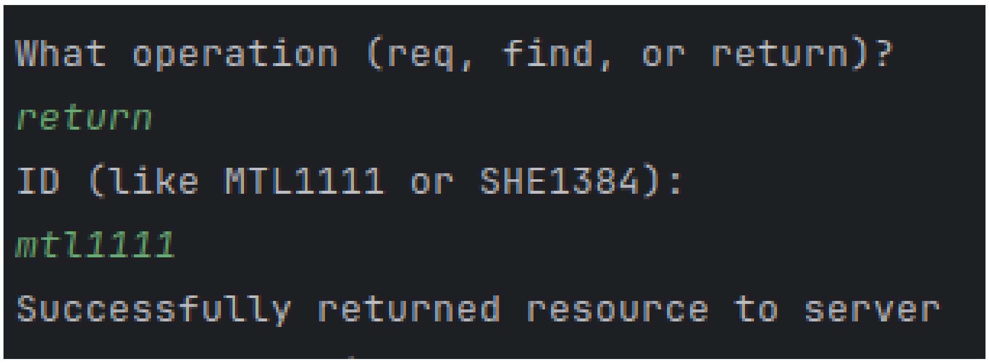
Edge case:



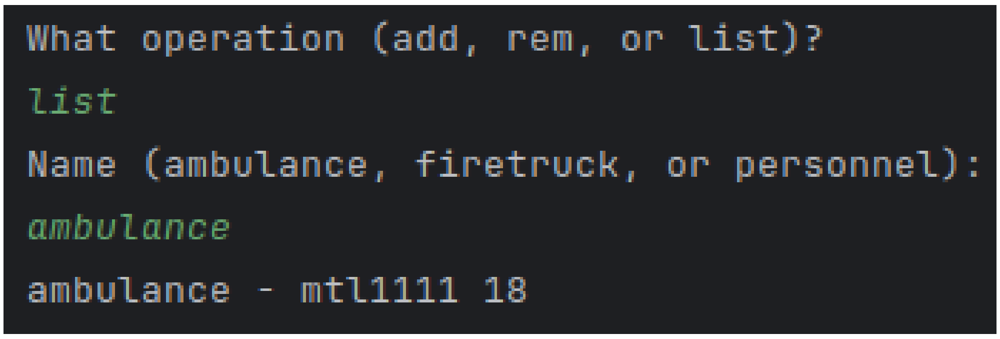
```
What operation (req, find, or return)?
req
ID (like MTL1111 or SHE1384):
mtl1111
Duration:
9
Resource was given
```



```
What operation (add, rem, or list)?
add
ID (like MTL1111 or SHE1384):
mtl1111
Name (ambulance, firetruck, or personnel):
ambulance
Duration:
9
Duration increased successfully
```



```
What operation (req, find, or return)?
return
ID (like MTL1111 or SHE1384):
mtl1111
Successfully returned resource to server
```



```
What operation (add, rem, or list)?
list
Name (ambulance, firetruck, or personnel):
ambulance
ambulance - mtl1111 18
```

Above shows what happens when a client returns a resource which the server was already holding.

Test cases for concurrency:

For testing concurrency I have added a file in the Moodle submission called ConcurrencyTest.java.

Issues:

Issue 1: make a server send a message back to client, after which the server would check what the client said and had to act upon that. This was the case for adding a client to the waiting queue for a resource if it wasn't available at the time of the request.

Solution 1: have the server send back the question, then let the client check if the question was asked. If it was and client responded yes: let the server add the client to the waiting queue. This is done by calling addToQueue with clientID and resourceID as arguments.

Issue 2: deciding when the resource would be given to a waiting coordinator. For example, I had the case where client1 requested mtl1111 with duration 8. However, client2 held that resource but with duration 1. Now if client2 decided to return this resource, would the resource be allocated to client1 but with less duration, or would another waiting client who joined after client1 joined but only requested duration 1 get the resource?

Solution 2: for simplicity I have decided to just allocate the resource to the waiting client at the head of the queue to avoid extra complexity.