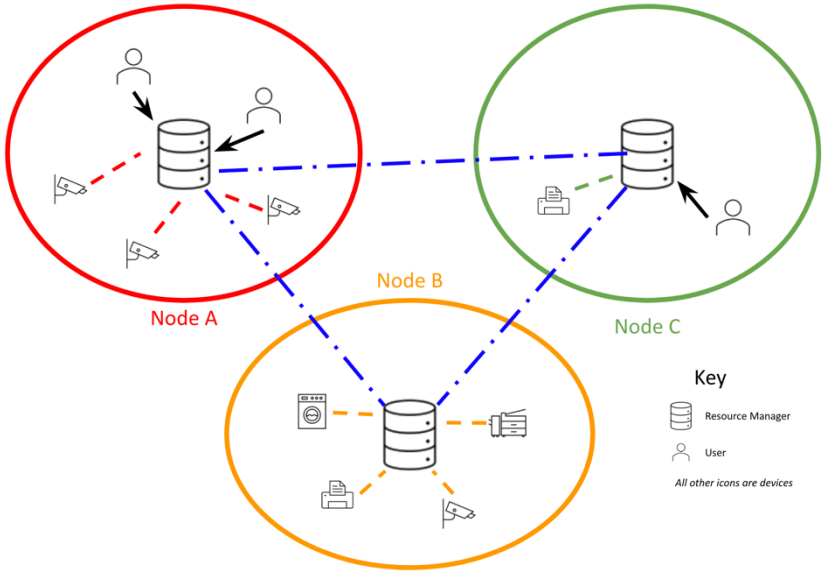
Project 4 – Distributed Resource Management

Due: 11/08/2021 11:59pm

Overview. In this project, you will implement a distributed resource-management scheme using Akka actors. Generally speaking, *resources* are anything that a system may use to fulfill the tasks that its users ask of it. Disk drives are resources, as are printers and scanners; memory and cores can also be viewed as resources, as can database controllers and more exotic devices such as motion sensors, bar code readers, and webcams. In a distributed system, one often wants to share these resources with all nodes in the network.[1]



Conceptually, the system you will be building consists of a collection of computing nodes (think "machines on a network") that can communicate via message passing. Each node owns a collection of local resources that it is willing to share with other nodes in the network.

To manage this sharing, each node includes a *Resource Manager*, which is responsible for controlling access to the resources it owns. *Users*, which are also running on the system nodes, will periodically request access to shared resources; *the Resource Manager* processes requests for its locally controlled resources, and forwards requests for remote resources to the relevant node's *Resource Manager*.

For this project, it is your responsibility to implement the full logic of the *Resource Manager.*

[1] In an actual system, one would also want to control access to these resources using authentication schemes and the like to ensure system safety and security. This project will not be concerned with these aspects, although they are of course a major issue in real system design.

Background. Before diving into the *Resource Manager*'s specifications, let's first gain a deeper understanding of users and resources.

*Resources***.** Rather than defining specific devices (printers, disk drives, etc.), this project keeps resources fairly abstract, and represents them as objects: `Resource.java`. ***No two resources can have the same name.***

There are two types of resource requests: *access requests* and *management requests*.

The access requests that a user make are as follows.

- **Exclusive write.**
  - A user may only be granted exclusive-write access to a resource if it is the sole user on said resource.
  - A user with exclusive-write access to a resource is permitted to request additional write and read requests
  - If a user holds exclusive-write access, no other user is permitted to access said resource until the user has relinquished its access.
- **Concurrent read.**
  - A user may only be granted concurrent-read access to a resource as long as no other users holds exclusive write access
  - Any number of users may simultaneously hold concurrent-read access; in this case, no user is allowed to have exclusive-write access

After an access request is granted, a user will submit an *access release* to relinquish its permissions to a resource. More on this later.

The management requests that users may make are as follows.

- **Disable.** A user may disable a resource, making it unavailable for users to access.
- **Enable.** A user may enable a resource, making it available again to other users.

Note that users do not need to have any access request privileges to a resource in order to make management requests; a user may send management requests at any time.

Users. In this project, users are implemented as `UserActor`. At initialization, each user will be assigned to a resource manager. Through this manager, users will issue a stream of requests and await responses for whether they were granted or denied. The specific stream of requests is given as a "script" when the user is created. When a user finishes executing its script, that user terminates. You may assume users are provided valid input scripts.

Users are assumed to be tied to a single computing node in a system; the purpose of the *Resource Manager* is to enable users to interact with resources without needing to know which node the resource belongs to.

Access requests may take one of two forms.

- **Blocking Access Request.**
  - The user indicates a desire to wait until the read/write access is granted.
  - The user will await a response from its local resource manager before continuing its execution
- **Non-blocking Access Request.**
  - The user indicates an unwillingness to wait.
  - If access cannot be granted immediately, the user is told that the resource request was denied and continues its execution

If the access request is granted, then the user may access the resource. When the user is done, they will send an access release message to remove permissions from the resource.

Management requests are processed immediately.

Project Details. For this project you will be expected to implement the class `ResourceManagerActor`. Each such actor will communicate with other resource-manager actors and users solely via message-passing. In your implementation of `ResourceManager`, **you are NOT allowed to use `Patterns.ask()`! You are also NOT allowed to share mutable objects among actors! All inter-actor communication must take place using via `tell()`.**

Take a look at `SimulationManagerActor` and actors from the CMSC433 Coding Examples repository for examples on how to communicate via `tell()`.

Specification in Detail. Resource managers process requests for their local resources and forward remote resource requests to remote resource managers. You will implement the resource management, message processing, and logging of the `ResourceManagerActor` through `createReceive()` and/or `onReceive()`

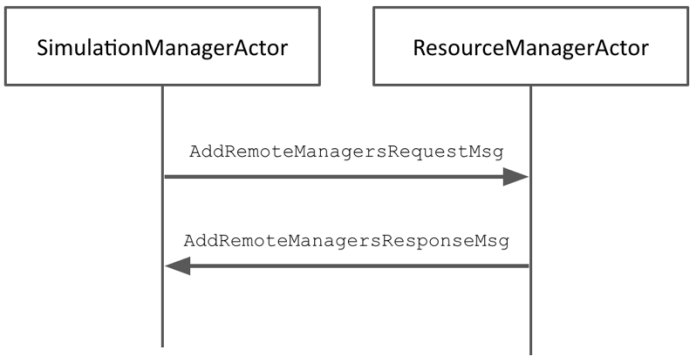
1. ***Resource Management.*** Resources are identified by their unique names (string).
   - At initialization, each `ResourceManagerActor` will be assigned a list of local `Resource` objects via an `AddInitialLocalResourcesRequestMsg`
   - For each local `Resource`, a `ResourceManagerActor` must maintain:
     - a separate queue for blocking access requests. If a blocking access request cannot be granted immediately, it should be placed into the queue.[2] Pending requests will be handled in FIFO order *per resource*;
     - a record of which users currently have read and write access
     - enable/disable status
     - a list of users awaiting confirmation on a resource's pending disablement

[2] Real systems will often use other schemes for processing requests that are based on priorities assigned to different types of requests. For example, reads might be given preference over writes, or vice versa. This project will only use the simpler FIFO mechanism, however.

2. ***Message Processing.*** There are a total of 8 messages that `ResourceManagerActor` will need to process. Each message comes from an actor that is waiting for a response. Make sure you reply with the appropriate response message, or the program will wait for the response and time-out. This response message can be sent by any `ResourceManagerActor`.

These messages are divided into the following categories. Example sequence diagrams for each section are included to assist in visualizing actor communication.

Initialization: These messages are sent out at the beginning of the simulation. Your implementation should handle these messages by updating appropriate data structures within the resource manager and sending the appropriate logging messages.



`AddRemoteManagersRequestMsg msg`

- This message contains a list of all `ResourceManagerActor` in the simulation, including the manager receiving the message.
- Take care when processing this list; in particular, you should NOT modify it. Instead, make copies of its contents
- After processing, reply to `getSender()` with `AddRemoteManagersResponseMsg`

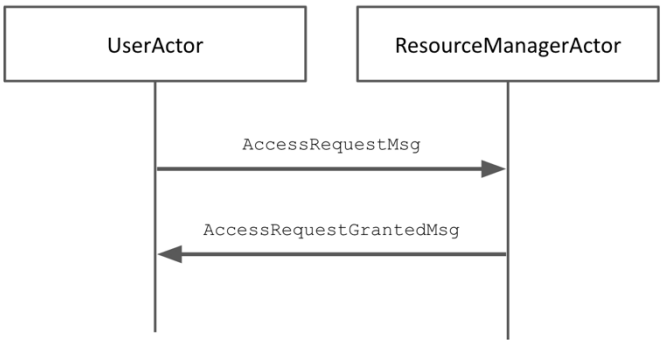
`AddLocalUsersRequestMsg msg`

- This message contains a list of local users who will issue requests to a `ResourceManagerActor`
- After processing, reply to `getSender()` with `AddLocalUsersResponseMsg`

`AddInitialLocalResourcesRequestMsg msg`

- This message contains a list of local `Resource` under a `ResourceManagerActor`
- The resources contained in this message all have their status set to “disabled”. As part of processing these messages, you should be sure to enable all the resources.
- After processing, reply to `getSender()` with `AddInitialLocalResourcesResponseMsg`

Local Request Processing. If a resource request belongs to a local resource, the `ResourceManagerActor` should take care of the request internally and update appropriate data structures.



`AccessRequestMsg msg`

The requesting user can be reached using `msg.getReplyTo()`. The exact type of each access request can be found using `msg.getAccessRequest().getType()`

When an access request arrives, `ResourceManagerActor` should examine if it can be granted and send an appropriate reply **directly** to the user.

- *Non-blocking Write Requests:*
  - If the resource is enabled and not occupied by another user, update data structures and send back an `AccessRequestGrantedMsg`
  - Otherwise, send back an `AccessRequestDeniedMsg` with the appropriate denial reason.
- *Non-blocking Read Requests:*
  - If the resource is enabled and not being written by other users, update data structures and send back an `AccessRequestGrantedMsg`
  - Otherwise, send back an `AccessRequestDeniedMsg` with the appropriate denial reason.

- *Blocking Write Requests:*
  - If the resource is enabled and not occupied by another user, update data structures and send back an `AccessRequestGrantedMsg`
  - If the resource is disabled, send back an `AccessRequestDeniedMsg`
  - Otherwise, insert the request into the resource's pending requests queue

- *Blocking Read Requests:*
  - If the resource is enabled and not being written by other users, update data structures and send back a `AccessRequestGrantedMsg`
  - If the resource is disabled, send back an `AccessRequestDeniedMsg`
  - Otherwise, insert the request into the resource's pending requests queue

`ResourceManagerActor` should also support *re-entrant* access permissions.

- If a user requests access to a resource that it already holds, then it is automatically granted.
- To release a resource completely, a user must release as many accesses as it has been granted.
  - For example, if a user has two concurrent-read access to a resource given to it, then it must release access to that resource twice.

Users can also obtain different types of access to a resource.

- If a user has exclusive-write access to a resource, then it may also be granted a concurrent-read access to the same resource.
- If a user holds a concurrent-read access to a resource, and no other users occupy the resource, then they may be granted an exclusive-write access

`ManagementRequestMsg msg`

The requesting user can be reached using `msg.getReplyTo()`. The exact type of each management request can be found using `msg.getRequest().getType()`

Disable Requests

- If the requesting user currently holds access rights to the resource, send back a `ManagementRequestDeniedMsg` with the appropriate denial reason
- Otherwise, the resource is now pending disablement:
  - **Immediately** stop accepting access requests. All future access requests will be send an `AccessRequestDeniedMsg`
  - **Immediately** clear the resource's pending queue and respond to pending blocking access requests with an `AccessRequestDeniedMsg` with the appropriate denial reason
  - Once **no users occupy the resource,** set the resource status to DISABLE and send the user a `ManagementRequestGrantedMsg`

Note that the users currently accessing a resource are NOT kicked. Instead, the resource status is not actually disabled until users release their access and **there are no users occupying the resource**.

A resource may be disabled multiple times; the second and subsequent such requests are ignored, except for the sending of a `ManagementRequestGrantedMsg` to the user(s) when the resource status is updated.

Enable requests

- Set the resource status to ENABLE and send the user a `ManagementRequestGrantedMsg`

Multiple enabling is allowed; second and subsequent enabling requests are ignored, except that `ManagementRequestGrantedMsg` messages are sent to the user(s)

You do not need to handle the case where an **enable request comes in while disable is pending**.

`AccessReleaseMsg msg`

To release access rights, users send release messages to the resource manager. The exact type of each access release can be found using `msg.getAccessRelease().getType()`

- If the user was previously granted access of this type to the resource, remove the user's access right. Otherwise, ignore the release request
- If the user has been granted multiple access rights to a resource, a release request only terminates one of those rights.

Note that **no message is sent back to the user**

Processing `AccessReleaseMsg` is complicated by resources pending disablement. After releasing access:

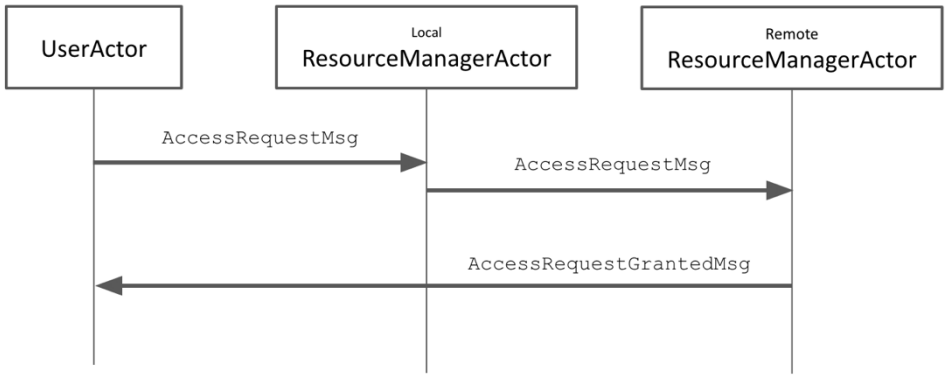
- If the resource is pending disablement and is not occupied by any users, set the resource status to DISABLE and send users who requested to disable the resource a `ManagementRequestGrantedMsg`

Note, multiple users could have requested the disable, so you need to notify all of them.

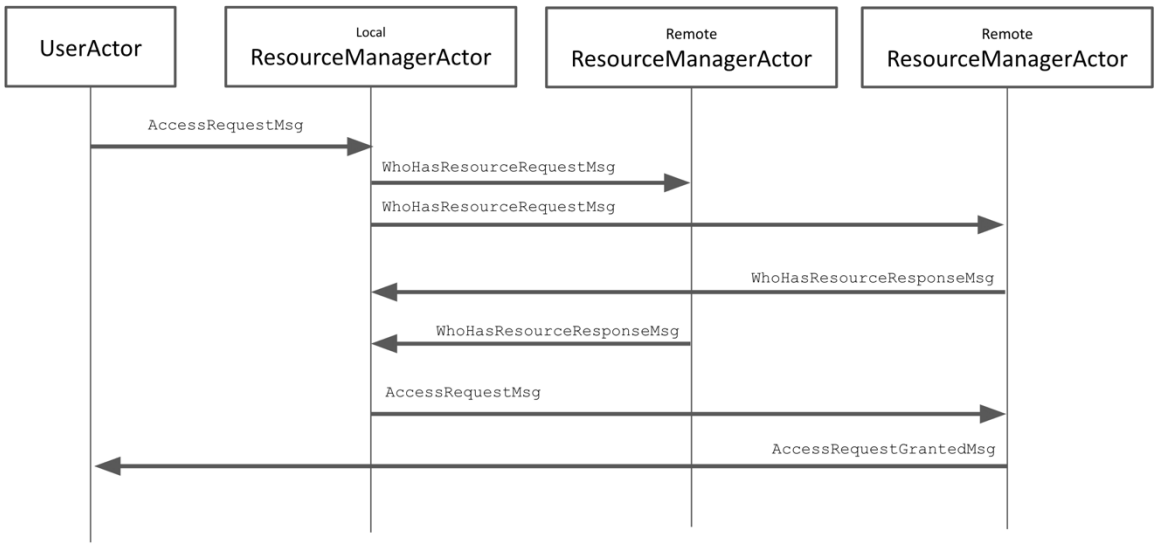
Remote Request Forwarding. For resources that are not local to a `ResourceManagerActor`, the actor must forward the request.

To support request forwarding, each `ResourceManagerActor` should maintain a table mapping resources to the resource manager in which the resource is local. When a manager is given a request for a remote resource, it consults this table to determine if the resource's manager is known.

Known Managers. If the manager is known, `ResourceManagerActor` forwards the request to that manager.



Unknown Manager. If the manager for the resource is not in the table, then the resource manager needs to locate its manager.



To do this, the `ResourceManagerActor` should send a `WhoHasResourceRequestMsg` to all remote managers asking if they manage the resource. Remote resource managers will send back a `WhoHasResourceResponseMsg` indicating their response.

- If a remote manager replies affirmatively, then the local manager updates its table and forwards the request
- If no remote manager replies affirmatively, then the request is for a non-existent resource. The resource manager should send a `<Access/Management>RequestDeniedMsg` to the user with the appropriate denial reason ("reason not found").

`ResourceManagerActor` to `ResourceManagerActor` communication is up to your design. You may modify `WhoHasResource<Request/Response>Msg` however you like.

3. ***Logging.*** In order for us to grade your program, we will ask you to log events by sending `LogMsg` objects to the `LoggerActor` via the `log()` method found inside `ResourceManagerActor.`

`LogMsg.java` also contains the definition of an enumerated type, `EventType`, listing the events that should be sent to the logger, and when. You will need to log each `EventType` that the `LogMsg` specifies.

The logging methods in `LogMsg` are between **Line 93 and Line 270.** It is recommended you use these static methods to construct `LogMsg` objects.

Note, logging does not necessarily take place in one specific method or code block for different events. The same log message can be logged in several places, depending on your implementations.

The final message log is not printed until all `ResourceManagerActor` terminate. If your code crashes or times out, it will not print anything out.

Code skeleton. We are providing you with the following classes and other infrastructure. Details about them can be found in the comments in the files you are being provided.

Actors. The following actor classes are provided. **DO NOT MODIFY ANY FILES EXCEPT FOR ResourceManagerActor.**

- `LoggerActor.java`. This class implements a class of actors that handle logging.
- `ResourceManagerActor.java`. **This is the class whose implementation you must complete.**
- `SimulationManagerActor.java`. Actors in this class construct and run simulations of resource managers. The specification of the system to simulate is given as a list of so-called *node specifications*, which include a list of scripts that users associated with the node should run and a list of local resources to manage. A `SimulationManager` actor then constructs a `ResourceManager` for each node, including a `UserActor` for each script, and launches the simulation by sending start messages to all user actors it creates.
- `UserActor.java`. This class defines a class of user actors, which generate access and management requests and awaits appropriate responses.

Enums. Several enumerated types are provided for you. **These must not be modified.** You may include other such types if you wish, however.

- `AccessRequestDenialReason.java`. Reasons an access request can be denied.
- `AccessRequestType.java`. Types of access requests a user can make made.
- `AccessType.java`. Types of possible access rights a user can hold.
- `ManagementRequestDenialReason.java`. Reasons a management request can be denied.
- `ManagementRequestType.java`. Types of management requests that a user can make.
- `ResourceStatus.java`. Status of a resource (enabled or disabled).

Messages. A number of message classes are provided for you. **DO NOT MODIFY ANY FILES EXCEPT WhoHasResourceRequestMsg and WhoHasResourceResponseMsg.**

- `AccessReleaseMsg.java`, `AccessRequestDeniedMsg.java`, `AccessRequestGrantedMsg.java`, `AccessRequestMsg.java`. These classes define messages used to request, and grant or deny, access rights.
- `AddInitialLocalResourcesRequestMsg.java`, `AddInitialLocalResourcesResponseMsg.java`, `AddLocalUsersRequestMsg.java`, `AddLocalUsersResponseMsg.java`, `AddRemoteManagersRequestMsg.java`,

`AddRemoteManagersResponse.java`. These messages are used during the configuration phase of system construction. **Your implementation of `ResourceManagerActor` must use these appropriately!**

- `LogMsg.java`, `LogResultMsg.java`. These messages are used for communication with `LoggerActor` actors. The logger uses `LogResultMsg` messages to send completed logs to simulation managers.
- `ManagementRequestDeniedMsg.java`, `ManagementRequestGrantedMsg.java`, `ManagementRequestMsg.java`. These are message types that users and resource managers use to handle management requests.
- `SimulationFinishMsg.java`, `SimulationStartMsg.java`. These messages are used to communicate between "Java-world" and simulation managers.
- `UserStartMsg.java`. These messages are used by simulation managers to start user actors at the beginning of a simulation.

Utilities. These classes contain implementations of various useful auxiliary data structures.

- `AccessRelease.java`, `AccessRequest.java`. Type of access releases and requests that users can make. These are used in user scripts.
- `Main.java`. Sample launching of a simulation.
- `ManagementRequest.java`. Type of management requests that users can make. These are used in user scripts.
- `NodeSpecification.java`. Specification of a node in a resource-management system. The specification includes a list of scripts (so a node should include a user for each script), and a list of resources (these are the local resources that the resource manager of the node should control access to).
- `Resource.java`. Type of resources.
- `SleepStep.java`. Encapsulates sleep step used in used in user scripts.
- `SystemActors.java`. Objects in this type contain a list of resource-manager actors and a list of user actors that have been created when constructing a resource-management system.
- `Systems.java`. Static methods for building up resource-management systems.
- `UserScript.java`. Type of scripts that users execute. A script consists of a sequence of steps, where each step consists of a list of access and management requests, and access releases, or a request to sleep for a given period of time. To execute a script, a user executes one step at a time, in the order specified. To execute a step, the user sends messages for each request, then awaits the responses (with the exception of release request, which do not result in responses being sent). When all responses are received, the user moves on to the next step.

We will only interact with your code using the messages that we provide to you, so you must ensure that you correctly respond to these messages.

Installing akka. For this project, you will need to install the akka 2.5 actor libraries for Java 8 and ensure they are on the build path for your implementation. Directions for doing this may be found in the lecture notes.

Testing. You are responsible for testing your code as you implement your solution to this project. Collaboration on the creation of test cases is encouraged, as is their sharing. Of course, you should not share code that you intend to submit.

Due to the nature of this project, you will have to build the general structure of your code even before you can test it. In other words, debugging can be extremely hard. It is recommended you start your project early and utilize print statement debugging as you build up actor's logic.

Due to the nature of multithreading, the exact order of the log messages can fluctuate. This is expected behavior, as long as the relative order of events remains the same.

Submission. Submit ONLY the `ResourceManagerActor.java`, `WhoHasResourceRequestMsg.java`, and `WhoHasResourceResponseMsg.java` file on Gradescope. You may submit an unlimited number of times.

Grading. Project 4 is 100% semi-public tests. You will be able to see each test's name, output, and result. You will not have access to the test's internal logic.