Project 4 – Distributed Resource Management

Due: Fri. 11/16/2018 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]

System model. 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 contains 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 associated with that machine. Users, which are also running on the system nodes, will periodically request access to shared resources; the resource manager is responsible for managing access to its locally controlled resources, and for forwarding requests for remote resources to the relevant node's resource manager.

Resource model. What kinds of resources can be included in nodes? Rather than defining specific devices (printers, disk drives, etc.), this project keeps resources fairly abstract, and characterizes them by their names. ***As a result, no two resources can have the same name.***

What requests can a user make of a resource? In this project, there will be two sorts of requests: *access requests*, and *management requests*. Here are the allowed access requests.

- **Exclusive write.** If a user has exclusive-write access to a singleton resource, then s/he is permitted to perform any operation on the resource, including reading and changing the state of the resource. If any user has this level of access to a resource, then no other user is permitted access to the resource until the user has relinquished its access.
- **Concurrent read.** Users with this access to a resource may query the resource's state but not perform any operations that modify the resource's state in any way. Any number of users may simultaneously hold concurrent-read access; in this case, no user is allowed to have exclusive-write access, except in circumstances described 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.

[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.

- **Enable.** A user may re-enable a disabled resource, making it available again to other users.

Note that users do not need to have any access privileges to a resource in order to make management requests; a user may send these latter requests at any time. More about the specifics of both access and management requests is given later.

User model. From a resource-usage perspective, users make requests of resources. Users are assumed to be tied to a single computing node in a system; the purpose of the distributed-resource-management infrastructure that you will be implementing is to enable users to interact with resources without needing to know which node that the resource is attached to.

The usage model is the following: users issue requests to their local resource manager and wait for responses about whether the request can be granted. Access requests may take one of two forms.

- **Blocking.** When a user issues a blocking access request, s/he is indicating a desire to wait until the access is granted. In such a case, the user is awaiting a response from a resource manager indicating that the access request has been granted. (Such requests may also be denied, however, if the resource is disabled. More information about this possibility is given later.)
- **Non-blocking.** When a user issues a non-blocking access request, s/he is indicating an unwillingness to wait. In such a case, if the access cannot be granted immediately, the user should be told that the resource request has been denied.

If the request is an access request and is granted, then the user may access the resource. When the user is done, s/he then releases his / her access rights. In the case of management requests, the local resource manager processes them appropriately, either by taking direct action if the resource is local or by forwarding the request to the appropriate remote resource manager. More detail about request processing is given later.

Resource management model. Resource managers are responsible for processing requests for their local resources, and for forwarding requests to relevant resource managers when a user requests a remote resource. The following describes each functionality in turn.

Request forwarding. To support request forwarding, resource managers should each maintain a resource table mapping resources to the managers that control them; in other words, the table should match resources to the managers to which the resource is local. When a manager is given a request for a remote resource, it first consults this table to determine if the manager for that resource is known, and if so, it forwards the request to that manager. If the manager for the resource is not in the table, then the resource manager needs to locate its manager. It does so by sending messages to all remote managers asking if they manage the resource. 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, and the resource manager should send a "resource not found" access-denial response to the requestor.

Local request processing. The processing of requests for local resources should proceed as follows. For access requests, the resource manager should implement a *locking* scheme. Specifically, for each resource, the manager should record which users currently have what type of access to that resource. When an access request arrives, the resource manager should examine if it can be granted and send an appropriate reply directly to the requesting user, even if the user is remote. Recall that access requests will come in two forms: blocking, and non-blocking. For a blocking request (read or write), the resource manager should only send a response granting the quest when it is possible to do so; otherwise, the manager should insert the request into a local queue of pending requests that it maintains. For a non-blocking access request, the manager should send an access-denied notification if the requested access cannot be granted immediately.

The access policy to be implemented should also support *re-entrant* access permissions. Specifically, if a user requests an 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. In particular, if a user has exclusive-write access to a resource, then it may also be granted a concurrent-read access to the same resource. Similarly, if a user holds a concurrent-read access to a resource, and no other user holds a concurrent-read access to the resource, then the user may also be granted an exclusive-write access.

To release access rights they no longer need, users should send release messages to their local resource manager; these must be appropriately handled by the resource manager. If the resource is controlled locally, the manager must check that the user performing the release request indeed has been granted the access previously; in this case, the access right of the user is terminated (but of course may be re-granted later if the user requests access again). If the user has been granted multiple access rights to a resource, a release request only terminates one of those rights. Invalid release requests (i.e. release requests sent by a user that has not been granted access) should be ignored. If the resource is managed remotely, the resource manager should forward the request appropriately.

Resource managers should refuse disable requests from users that currently have access to a resource controlled by the manager, by sending an "access held by user" message. If the requesting user does not hold access rights to the resource, the resource manager should stop accepting access requests, responding to such requests with an indication that the resource is disabled, but should wait until all granted accesses have been released before changing the status of the resource to "disabled". Pending blocking access requests should be responded to with an indication that the resource is disabled. Disable requests should be responded to with confirmations once the device status has been set to "disabled". A resource may be disabled

multiple times; the second and subsequent such requests are ignored, except for the sending of the response message when the device status has been updated. Any access requests to a disabled resource should be denied, with to with a "resource disabled" message.

For enable requests, the resource manager should start accepting access requests again. A confirmation message should be sent to the user who sent the enable request. Multiple enabling is allowed; second and subsequent enabling requests are ignored, except that response messages are sent.

Project details. For this project you will be expected to implement a class `ResourceManagerActor` of actors. Each such actor will contain a list of local resources that it manages, and a list of other resource managers in the system, and will communicate with other resource-manager actors, and users, solely via message-passing. ***In your implementation of `ResourceManager`, you are not allowed to `Patterns.ask()`! You are also not allowed to share mutable objects among actors! All inter-actor communication must take place via `tell()`.***

Resources. Each resource will have a unique name (string). You should maintain a queue of blocking requests that are pending. All such requests will be handled in a FIFO (first-in first-out) manner; if a request cannot be granted immediately, it should be placed into the queue.[2]

Users. Users will also be actors. Each user, upon creation, will be assigned a single resource manager as its "local manager," and will be added to a list of local users by this manager. Users will in general generate a stream of requests to the manager and then terminate; 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.

Logging. In order for us to grade your program, we will ask you to log events by sending messages to a special Logger agent that is provided for you. The infrastructure for this, and more explanation, may be found in the code skeleton. Specifically, file `LogMsg.java` in the `messages` package contains a description of events that should be logged, and when.

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.

- `LoggerActor.java`. This class implements a class of actors that handle logging. **Do not modify this file!**

[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.

- `ResourceManagerActor.java`. **This is the class whose implementation you must complete.** In particular, your code must implement the resource-management policies described earlier in this document, and you must log every message you send and receive by sending appropriate messages to the logging actor included.
  Your code must also handle configuration messages appropriately. These messages are sent out at the beginning of a system run, and list the local resources and users for each resource manager. Your implementation should handle these messages by updating appropriate data structures within the resource manager and sending the appropriate logging messages.
- `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. You may assume that all resources in the initial node specifications have unique names. 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. **Do not modify this file!**
- `UserActor.java`. This class defines a class of user actors, which generate access and management requests and awaits appropriate responses. **Do not modify this file!**

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. **These must not be modified.** You may include other message types if you wish, however.

- `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! There some details you should be aware of.
    - The list of actors in the payload of a `AddRemoteManagersRequestMsg` contains all of the manager actors created, including the manager receiving the message; also all managers receive this same list. You should take care when processing this list; in particular, you should not modify it, although you should feel free to make copies of its contents if you wish.
    - The resources contained in an `AddInitialLocalResourcesRequestMsg` message all have their status set to "disabled". As part of processing these messages, you should be sure to enable all the resources.
- `LogMsg.java`, `LogResultMsg.java`. These messages are used for communication with `LoggerActor` actors. In particular, users or resource managers should use `LogMsg` objects to send events to be logged to the logger. This file also contains the definition of an enumerated type, `EventType`, listing the events that should be sent to the logger, and when. 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.

Your implementation. You should use the message types provided in your actor implementations, together with any new messages you wish to add that you think useful. However, 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.

Remember, you are not allowed to use `Patterns.ask()` in your code! You are also not allowed to share mutable objects among actors! All communication among actors should only take place via asynchronous message passing, and should be achieved using the `tell()` method alone.

Installing akka. For this project, you will need to install the akka 2.4.20 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 (lec18.pdf).

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.

Submission. Submit a .zip file containing your project files to the CS submit (submit.cs.umd.edu). You may use the same approach outlined in Project 0 to create this from inside Eclipse.