SPL 224 - Assignment 2

The GurionRock Pro Max Ultra Over 9000 Vacuum Robot Perception and Mapping System Simulation

Java Generics, Concurrency, and Synchronization

Teaching Assistants in Charge:

- Lotem Sakira
- Ahmed Massalha

Publication date: 12/12/2024
Deadline: 02/01/2025

Before you start

Purpose

The goal of this assignment is to practice concurrent programming in Java 8, focusing on Java Threads, Synchronization, Lambdas, and Callbacks. Ensure you have a good understanding of these topics by revisiting relevant lectures and practical sessions.

Preparation

Before coding, read the entire assignment thoroughly.

Environment

While you may develop your project in any environment, your submission will be tested and graded only on a CS Lab UNIX machine. Therefore, you must compile, link, and run your assignment on a lab UNIX machine before submitting it.

Communication

Q&A Forum

All questions and answers regarding this assignment will be handled in the assignment forum only.

Updates

Critical updates about the assignment will be published on the assignment page on the course website. It's your responsibility to stay informed.

Forum Guidelines

- **Read Before Asking:** Review previous Q&As to avoid duplicate questions.
- **Courtesy:** Be polite; the course staff is here to assist you.
- **No Code Sharing:** Do not post any part of your solution or source code as hints for other students. For such discussions, attend staff reception hours.

Extensions

Only Hedi can authorize extensions. Contact him directly if you require one.

General Guidelines

- **Documentation:**

    - Read the Javadocs of all provided interfaces carefully.
    - **Adherence:** You must strictly follow the Java documentation for each class and method.
        - **Classes:** Only add data members with the allowed access levels.
        - **Methods:** Do not change return types, parameters, or thrown exceptions.
    - **Exceptions:** Do not throw exceptions not specified in the method's documentation. If an exception is not mentioned, do not throw it or add it to the method's throws clause.
    - **Synchronization:** You may add the synchronized keyword to any method if necessary.
- **Concurrency and Efficiency:**
    - **Concurrent Data Structures:** Use Java's concurrent data structures to write efficient concurrent code. They are designed for minimal synchronization and blocking.
    - **Thread Safety:** Ensure that the data structures you use are thread-safe as required.
    - **Synchronization Minimization:** Synchronize only where unavoidable to enhance performance.
    - **Performance Impact:** The efficiency of your implementation can affect your grade, but correctness must not be compromised for efficiency.

Introduction

Welcome to BGRock, BGU's newest team of Robot Engineers! Congratulations on starting your career in robotics.

BGU aims to release a new vacuum-mopping combo robot—the **GurionRock Pro Max Ultra Over 9000**—to compete in the prestigious DreamUs Vacuum-Cleaner Robots Competition.

Modern vacuum cleaner robots are equipped with multiple sensors for perception, allowing them to map their environment and make informed decisions based on both the map and their internal state. Common sensors include cameras, LiDAR, GPS, and IMU.

Managing multiple sensors operating in parallel presents challenges. The robot's software must support concurrent data collection and computation while ensuring synchronization with a global clock. Some computations require integrating data from multiple sensors simultaneously.

In industry, robotic engineers often utilize systems like the Robot Operating System (ROS), which supports running multiple sensors and software components (nodes) in parallel and facilitates data sharing through messaging. This approach simplifies development by providing a robust framework for communication and synchronization.

However, the BGRock team has decided to develop a custom Java-based system that enables the creation of nodes in Java. Your task is to build a simple microservice framework as a prototype and implement the robot's perception and mapping system.

Perception and Mapping System Overview

The perception system integrates data from several sensors to comprehend the robot's surroundings:

Camera

A camera captures 2D projections of the 3D world by recording electromagnetic signals. This process inherently loses one dimension, resulting in flat images that represent the environment. The team will use an affordable camera capable of object detection but not effective at measuring distances. **In this assignment, the camera will provide simple descriptions of detected objects without their coordinates.**

LiDAR

LiDAR (Light Detection and Ranging) is a remote sensing method that determines distances by emitting laser pulses and measuring their reflections. By capturing the time, it takes for these pulses to return, LiDAR systems can create highly accurate, high-resolution 3D point cloud maps of their surroundings. This ability makes LiDAR particularly well-suited for tasks like navigation, collision avoidance, and environmental modeling.

In this assignment, the LiDAR sensor will provide 3D point cloud data representing the objects it detects. Because LiDAR processing can be computationally intensive, it often relies on dedicated hardware or parallel computation techniques. To simulate this, we will model the LiDAR's data processing pipeline using multiple threads, each acting as an independent "data worker" to handle the sensor's workload more efficiently. The LiDAR Data will be handled in a LiDAR Database.

IMU and GPS

An Inertial Measurement Unit (IMU) and Global Positioning System (GPS) are essential for determining the robot's position and movement. The IMU measures acceleration, orientation, and angular velocity using accelerometers and gyroscopes, providing data on the robot's motion. GPS offers location information by receiving signals from satellites, enabling global positioning. Together, they allow the robot to understand its pose relative to a coordinate system. **In this assignment, a simple service will provide the robot's pose at each time tick, expressed in the charging station's coordinate system.**

Fusion-SLAM

While data fusion is not a sensor, it is a critical algorithm that combines information from multiple sensors to create accurate environmental landmarks. The fusion algorithm integrates camera, LiDAR, IMU, and GPS data, providing a comprehensive understanding of the robot's surroundings that surpasses what each sensor could achieve individually.

The fusion process will also serve as a simple, naive implementation of Simultaneous Localization and Mapping (SLAM). SLAM is a fundamental technique in robotics that enables a robot to build a map of an unknown environment while simultaneously tracking its location within that map. It is essential for autonomous navigation, allowing robots to navigate and make decisions without prior knowledge of the environment. **In this assignment, objects are assumed to have already been tracked to simplify complexity. This means the IDs provided by the camera service output**

correspond directly to those from the LiDAR worker service output. The primary task is to convert these landmarks into the charging station's coordinate system. If the LiDAR sensor re-detects an object, the measurement will be updated by averaging it with the previous data, thereby refining the accuracy of the object's mapped position.

Part 1: Implementing a Future Class

In this preliminary part, you will implement a basic Future class, which we will use in the rest of the assignment.

Future Class Overview

- **Purpose:** Represents a promised result—an object that will eventually be resolved to hold the result of some operation.
- **Usage:** Allows retrieving the result once it is available. When a MicroService finishes processing an event, it will resolve the relevant Future.

Methods to Implement

Future has the following methods:

- *T get()*:
  - Retrieves the result of the operation.
  - Waits for the computation to complete in the case that it has not yet been completed.
- *resolve(T result)*:
  - Called upon the completion of the computation.
  - Sets the result of the operation to a new value.
- *Boolean isDone()*:
  - Returns true if this object has been resolved.
- *T get(long timeout, TimeUnit unit)*:
  - Retrieves the result of the operation if available.
  - If not, wait for at most the given time unit for the computation to complete.
  - Returns the result if available; returns null if the waiting time elapsed before completion.

Part 2: Synchronization MicroServices Framework

In this section, you will build a simple MicroService framework. A MicroService framework consists of two main components: a **MessageBus** and **MicroServices**. Each MicroService is a thread that can exchange messages with other MicroServices using a shared object referred to as the MessageBus.

Message Types

There are two different types of messages:

Events

Definition

An Event defines an action that needs to be processed (e.g., a camera detecting an object in space).

Handling

Each Micro-Service specializes in processing one or more types of events.

Upon receiving an event, the MessageBus assigns it to the messages queue of an appropriate Micro-Service which specializes in handling this type of event.

If multiple MicroServices specialize in the same event type, the MessageBus assigns the event to one of them in a **round-robin** manner (described below).

Result Handling

- Each event holds a result, represented by a Future object. This result should be resolved once the Micro-Service to which the event was assigned completes processing it. For example, for the event of `Detecting object in the space,' the Micro-Service in charge should return the coordinates of the object in the space in the case that the detection was successfully completed.
- The MicroService that processes the event must resolve the Future once processing is complete.
    - While a Microservice processes an event, it might create new events and send them to the Message Bus. The Message Bus will then assign the latest events to the queue of one of the appropriate Microservices. For example, the Camera sends a 'DetectObject' event, which is then added to the LiDar queue.

Broadcast

Definition

Represents a global announcement in the system.

Handling

Each Micro-Service can subscribe to the type of broadcast messages it is interested in receiving. The MessageBus sends broadcast messages to all MicroServices that have subscribed to receive that specific type of broadcast message. Broadcasts are different from events, which are sent to only one MicroService.

Round Robing Pattern

When multiple MicroServices are subscribed to the same event type, the MessageBus distributes events among them in a round-robin fashion.
Example:

- Event Type: DetectObject
- Subscribed MicroServices: LiDarA, LiDarB, LiDarC, LiDarD
- Events: Object1 to Object6

Assignment:

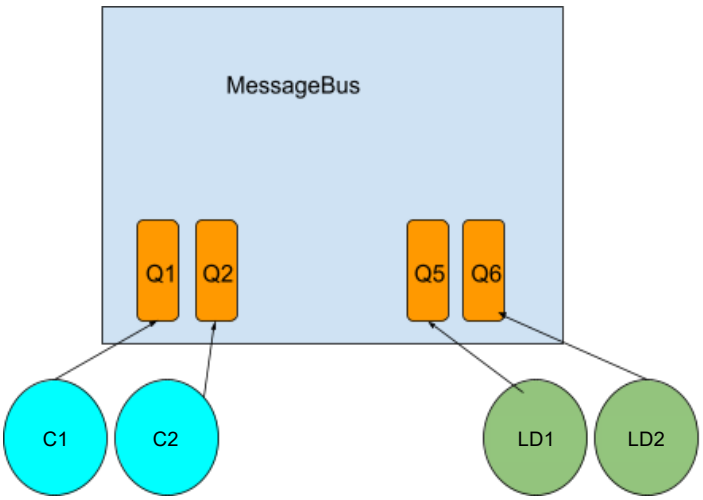
- Object1 → LiDarA
- Object2 → LiDarB
- Object3 → LiDarC
- Object4 → LiDarD
- Object5 → LiDarA
- Object6 → LiDarB
- ...and so on.

Example

In part 3 of the assignment, you will implement several such microservices, two of them will be the following (note that this is just a partial description; for a full description, see part 3):

- Camera: This creates the “DetectObject” event.
- LiDar: This creates the “tracked” event.

For simplicity, assume that there are only two instances of each of the above microservices. During initialization, each of the above microservices will register itself with the MessageBus and will have a queue instantiated for him in the MessageBus. See the next figure for illustration:



When a camera sends an event “DetectObject,” the MessageBus moves it to one of the LiDar queues, which processes the event.

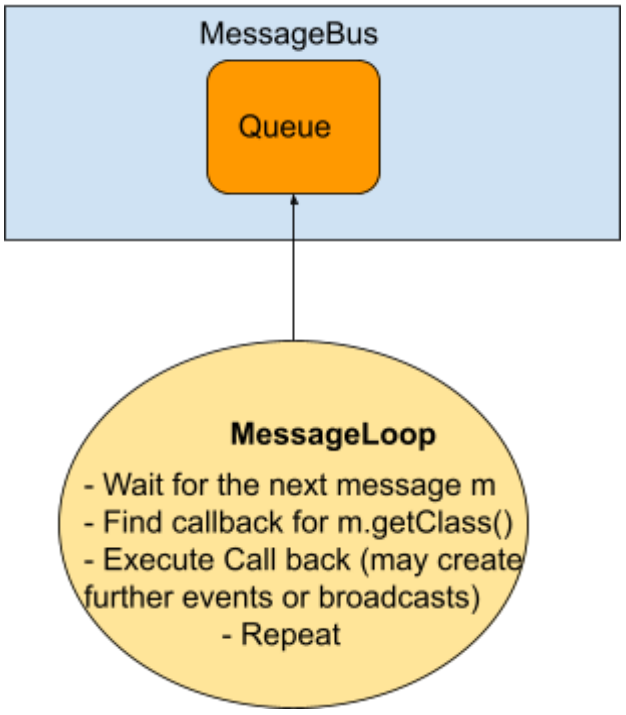
When an event is sent, a future is returned to the sender; for example, sending the “DetectObject” event will return a Future<Coordinates>. When the GPU is done handling the event, it will notify the MessageBus that the event is completed and set the Coordinates instance for the future. Thus, the camera will know that the detection has been completed.

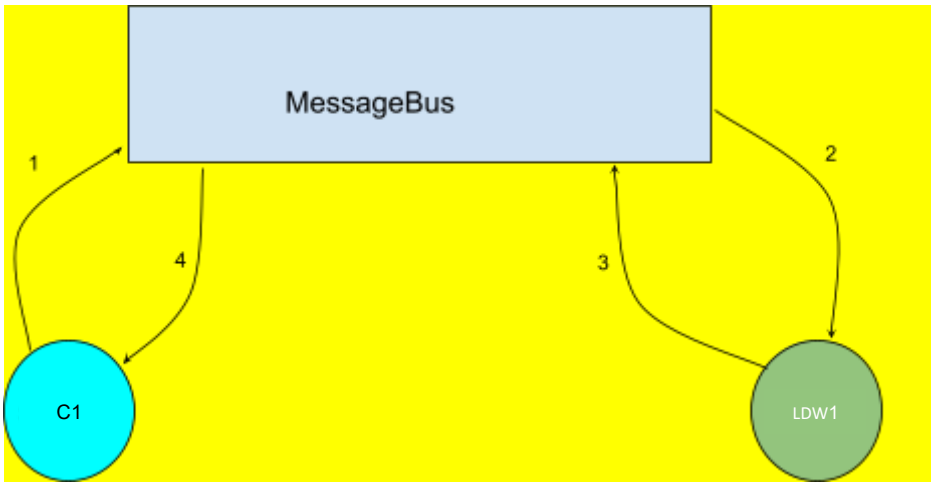
Message Loop Design Pattern

In this part, you have to implement the Message Loop design pattern. Each MicroService runs on its own thread and operates a message loop in such a pattern. In each iteration of the loop, the thread tries to fetch a message from its queue and process it. An important point in such a design pattern is not to block the thread for a long time unless there are no messages for it. The following figure describes the Message Loop in our system.

1. **Initialization:**
    a. Registers itself with the MessageBus.
    b. Subscribes to message types it is interested in.
    c. Performs any required initialization.
2. **Message Processing Loop:**
    a. Fetches messages from its message queue using awaitMessage.
    b. For each message, executes the corresponding callback.
    c. Continues until termination is signaled.

See the following figure for an example for a simple flow of events:
In this example, there is a single camera and a single LiDAR Worker.





1. The Time Service sends a tick broadcast. The Camera1 Service received it and detected that it needs to send a DetectObject event to the LiDAR.
2. Camera Sends the “DetectObject” event to the MessageBus, and the MessageBus inserts the “DetectObject” event to the LiDAR Worker 1 queue.
3. LiDAR Worker1 receives the message from the Queue and, according to its type (DetectObject), calls the appropriate CallBack; this processes the event.
4. LiDAR Worker 1 finishes processing the event and sends the result to the MessageBus, stating it is done with its results.
5. The MessageBus sets the future for the original event sent by Camera1 (e.g., the DetectObject), with the results of LiDAR Worker 1.
   Note: it’s optional. It returns an answer to the caller; you can use messages as void.

Implementation Details

To this end, each MicroService will run on its thread in our framework. The different Micro-Services will be able to communicate with each other using only a shared object: a Messagebus. The MessageBus supports the sending and receiving of two types of events: *broadcast messages*, which, upon being sent, will be delivered to every subscriber of the specific message type, and *Event messages*, which, upon being sent, will be delivered to only one of its subscribers (in a round-robin manner, described above).

The different Microservices will be able to subscribe for message types they would like to receive using the Message Bus. The different Microservices do not know each other's existence. All they know are messages that were received in their message queue, which is located in the Message Bus.

When building a framework, one should change the way one thinks. Instead of thinking like a programmer who writes software for end-users, they should now feel like a programmer writing software for other programmers. Other programmers will use this framework to build their applications. For this part of the assignment, you will create a framework (write code for other programmers); the programmer who will use your code to develop its application will be the future you while they work on the second part of this assignment. Attached to this assignment is a set of interfaces that define the framework you will implement. The interfaces are located at the *bgu.spl.mics* package. Read the JavaDoc of each interface carefully. You are only required to

implement this part's MessageBus and MicroService. The following is a summary and additional clarifications about implementing different framework parts.

- **Broadcast**: A Marker interface extending Message. When sending Broadcast messages using the MessageBus, it will be received by all the subscribers of this Broadcast message type (the message Class)
- **Event**: A marker interface extending Message. A Microservice that sends an Event message expects to be notified when the Microservice that processes the event has completed processing. The event has a generic type variable T, which indicates its expected result type (which should be passed back to the sending Microservice). The Microservice that has received the event must call the method 'Complete' of the Message-Bus once it has completed treating the event to resolve the event's result.
- **MessageBus**: The MessageBus is a shared object used for communication between MicroServices. It should be implemented as a threadsafe singleton, as it is shared between all the MicroServices in the system. Implementing the MessageBus interface should be inside the class MessageBusImpl (provided to you). There are several ways in which you can implement the message-bus methods; be creative and find a good, correct, and efficient solution. **Notice that fully synchronizing this class will affect all the micro-services in the system (and your grade!) - try to find good ways to avoid blocking threads as much as possible.** The MessageBus manages the queues for Micro-Services. It creates a queue for each MicroService using the 'register' method. When the Micro-Service calls the 'unregister' method of the Message-Bus, it should remove its queue and clean all references related to that Micro-Service. Once the queue is created, a Micro-Service can take the following Message in the queue using the 'awaitMessage' method. The 'awaitMessage' method is blocking; if no messages are available in the Micro-Service queue, it should wait until a message becomes available. Method includes:
    - *register*: A micro-service calls this method to register itself. This method should create a queue for the Micro-Service in the Message Bus.
    - *subscribeEvent*: A Micro-Service calls this method to subscribe itself to a specific class of event (the specific class type of the event is passed as a parameter).
    - *subscribeBroadcast*: A Micro-Service calls this method to subscribe itself to a broadcast message (the specific class type of the event is passed as a parameter).
    - *sendBroadcast*: A Micro-Service calls this method to add a broadcast message to the queues of all Micro-Services that subscribed to receive this specific message type.
    - *Future<T> sendEvent(Event<T> e)*: A Micro-Service calls this method to add the event e to the message queue of one of the Micro-Services that have subscribed to receive events of type e.getClass(). The messages are added in a round-robin fashion. This method returns a Future object - from this object, the sending Micro-Service can retrieve the result of processing the event once it is completed. If there is no suitable Micro-Service, it should return null.
    - *void complete(Event<T> e, T result)*: A Micro-Service calls this method to notify the Message-Bus that the event was handled and provides the result of handling the request. The future object associated with event e should be resolved according to the result given as a parameter.
    - *unregister*: A Micro-Service calls this method to unregister itself. It should remove the message queue allocated to it and clean all the references related to this Message Bus.

    - *awaitMessage(Microservice m)*: A Microservice calls this method to take a message from its allocated queue. This method is blocking (it waits until there is an available message and returns it).
- **MicroService**: The MicroService is an abstract class that any MicroService in the system must extend. The abstract MicroService class is responsible for getting and manipulating the singleton MessageBus instance. Derived classes of MicroService should never directly touch the Message-Bus. Instead, they can use a set of internal protected wrapping methods. The derived class also supplies a callback function when subscribing to message types. The MicroService stores this callback function together with the kind of Message it is related to. The Micro-Service is a Runnable (i.e., suitable to be executed in a thread). The run method implements a message loop. When a new message is taken from the queue, the Micro-Service will invoke the appropriate callback function. When the Micro-Service starts executing the run method, it registers itself with the Message Bus and then calls the abstract initialize method. The initialize method allows derived classes to perform any required initialization code (e.g., subscribe to messages). Once the initialization code is complete, the actual message loop should start. The Micro-Service should fetch messages from its message queue using the Message-Bus's awaitMessage method. Each Message should execute the corresponding callback. The MicroService class also contains a termination method that should signal the message loop that it should end. Each Micro-Service contains a name given to it in construction time (the name is not guaranteed to be unique).

Important:

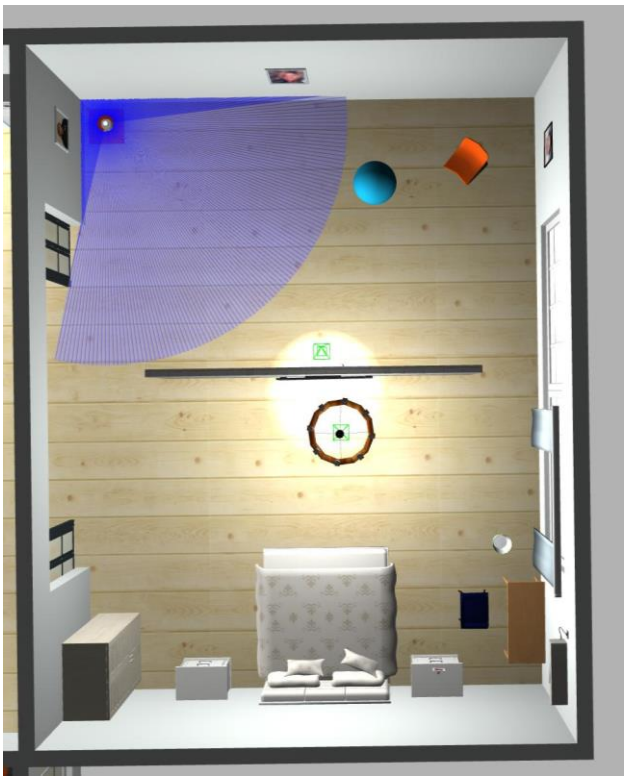
- **Callbacks Execution:** All callbacks must be executed inside the MicroService's own message loop.
- **Registration and Unregistration:** Must occur inside the run method of the MicroService.
- **Modifying Interfaces:**
    - You may add queries to the MessageBus interface, but they must be independent of your implementation.
    - For example, adding isMicroServiceRegistered is allowed; adding getQueue is not, as it forces a specific implementation.

Code Example:

You can find in the package bgu.spl.mics.example a usage example of the framework. This example is simple; it creates simple services, such as ExampleBroadcastListenerService, ExampleEventHandlerService, and ExampleMessageSenderService, and two messages: ExampleBroadcast and ExampleBroadcast.

Part 3: The GurionRock Perception-Mapping System

Before implementing this part, see Part 4.



In this part of the assignment, you will develop a perception and mapping system for the robot. This system uses the MicroService framework implemented in the previous section, along with data provided via JSON files that simulate sensor operations. The core idea is that as the robot moves through a house, its sensors (camera and LiDAR) collect information at specific time intervals. These observations are then integrated into a global map by the Fusion-SLAM service.

System Overview

- **Concepts:**
  - As time progresses, measured in discrete ticks, the robot moves through its environment.
  - Each sensor (camera, LiDAR) produces data at certain intervals based on their frequency. For example, if an object is detected at time T and the camera's frequency is F, the camera will make that data available at time T + F.
  - The camera detects objects but lacks precise distance measurements. It will send DetectObjectsEvent messages to LiDAR workers, which, in turn, will fetch the corresponding cloud points for those objects from a LiDARDataBase object. After obtaining the cloud points, the LiDAR worker provides detailed positional information for the detected objects.
  - The LiDAR workers, upon receiving these events and after factoring in their own frequency, send TrackedObjectsEvent messages to the Fusion-SLAM service.
  - The Fusion-SLAM service integrates all this information, converting coordinates into a global frame of reference (relative to the charging station) and updating a map of the environment with the objects' landmarks.
- **Workflow:**
  - **Time Broadcasts:**
    A TimeService broadcasts TickBroadcast messages at each tick. These ticks represent the passage of time in the simulation.

- **Pose Information:**
  The PoseService uses the current tick to determine the robot's position and orientation (pose) relative to the charging station. It sends PoseEvent messages to the Fusion-SLAM service, so it knows where the robot is at every time step.
- **Camera Data Processing:**
  When a new tick arrives, each CameraService checks if it has new detected objects at a time that aligns with its sending interval (time + camera_frequency). If yes, it sends DetectObjectsEvent messages to LiDAR workers. The camera provides the objects' IDs and descriptions, but not their exact coordinates.
- **LiDAR Data Processing (via LiDARDataBase):**
  A LiDAR worker receives a DetectObjectsEvent from the camera. When it reaches the appropriate tick (time + lidar_frequency), the LiDAR worker looks up the corresponding object's cloud points in a LiDARDataBase. After retrieving these coordinates, it sends a TrackedObjectsEvent to the Fusion-SLAM service. This step transforms raw detections into meaningful spatial data that can be used to update the map.
- **Fusion-SLAM Integration:**
  The Fusion-SLAM service, upon receiving TrackedObjectsEvent messages, transforms the object coordinates into the global coordinate system based on the current robot pose. It then updates or creates landmarks on the map. Be aware that the fusion-slam can only do the calculations after it has ALL the necessery data.
  - If it's a new object, Fusion-SLAM adds it as a new landmark.
  - If it's an already known object, it refines the object's location by averaging the new measurements with the previous data.

- **Event Chain Example:**
  - At time T, the camera detects objects and sends DetectObjectsEvent to a LiDAR worker.
  - The LiDAR worker, once it reaches the right time (T + idar_frequency), uses the LiDARDataBase to find the corresponding cloud points for those objects and then sends a TrackedObjectsEvent to Fusion-SLAM.
  - The Fusion-SLAM service uses the current pose and the tracked objects' data to update its global map.
- **Additional Notes:**
  - Camera services may send multiple DetectObjectsEvents over time, once previous ones are processed.
  - Frequencies ensure a delay between when objects are detected and when the full, coordinate-rich data becomes available.
  - Throughout the process, TickBroadcast messages from the TimeService synchronize all components with the global clock.

Objects and Data Structures

Camera

Represents a single camera.

- **Fields:**
  - id: int – The ID of the camera.
  - frequency: int – The time interval at which the camera sends new events (If the time in the Objects is 2 then the camera sends it at time 2 + Frequency).
  - status: enum – Up, Down, Error.
  - detectedObjectsList: List of Stamped DetectedObject – Time-stamped objects the camera detected.

StampedDetectedObjects

Represents objects the camera detected with a time stamp.

- **Fields:**
  - time: int – The time the objects were detected.
  - DetectedObjects : List<DetectedObject> – list of objects that were detected at time step `time`.

DetectedObject

Represents the object that was detected by the camera

- **Fields:**
  - id: string – The ID of the object.
  - description: String – Description of the object.

LiDarTrackerWorker

Represents a LiDar

- **Fields:**
  - id: int – The ID of the LiDar.
  - frequency: int – The time interval at which the LiDar sends new events (If the time in the Objects is 2 then the LiDarWorker sends it at time 2 + Frequency).
  - status: enum – Up, Down, Error.
  - lastTrackedObjects: List of TrackedObject – The last objects the LiDar tracked.

TrackedObject

Represents the object that was tracked by the LiDar

- **Fields:**
  - id: string – The ID of the object.
  - time: int – The time the object was tracked.
  - description: String – Description of the object.
  - coordinates: Array of CloudPoint – The coordinates of the object.

StampedCloudPoints

Represents a group of cloud points corresponding to an id with a time stamp

- **Fields:**
  - id: String – The ID of the object.
  - time: int – The time the object was tracked.
  - cloudPoints: List of lists of type double– List of cloud points.

CloudPoint

Represent the coordinates of a specific point of the object according to the LiDar.

- **Fields:**
  - x: int
  - y: int

FusionSLAM

- **Fields:**
  - landmarks: Array of Landmark – Represents the map of the environment.
  - Poses: List of type Pose – Represents previous Poses needed for calculations.

Landmark

Represents a landmark in the environment map.

- **Fields:**
  - Id: String – the internal ID of the object.
  - Description: String – the description of the landmark.
  - Coordinates: List of CloudPoints – list of coordinates of the object according to the charging station's coordinate system.

GPSIMU

Represents the robot's GPS and IMU. It will have the following fields

- **Fields:**
  - currentTick: int – the current time.
  - status: enum – Up, Down, Error.
  - PoseList: list of type Pose – represents a list of time-stamped poses.

Pose

The robot's current pose (x: int,y: int, yaw: int) according to the charging station's coordinate system.

- **Fields:**
  - x: float
  - y: float
  - yaw: float – The orientation angle relative to the charging station's coordinate system.
  - Time: int – the time when the robot reaches the pose.

StatisticalFolder

The StatisticalFolder aggregates key metrics reflecting the system's performance. Both cameras and LiDAR workers update this folder each time they send their observations, regardless of whether the detections are new or re-detections. This ensures that the statistics accurately represent all detected and tracked objects.

- **Fields:**
  - systemRuntime: int – The total runtime of the system, measured in ticks.
  - numDetectedObjects: int – The cumulative count of objects detected by all cameras. This includes both initial detections and subsequent re-detections.
  - numTrackedObjects: int – The cumulative count of objects tracked by all LiDAR workers, encompassing both new tracks and ongoing tracking of previously detected objects.
  - numLandmarks: int – The total number of unique landmarks identified and mapped within the environment. This count is updated only when new landmarks are added to the map.

LiDARDataBase

Holds the LiDar's data.

- **Fields:**
  - cloudPoints: List<StampedCloudPoints> - The coordinates of that we have for every object per time.

Messages:

The following message classes are mandatory, the fields that these messages hold are omitted; you need to think about them yourself:

DetectObjectsEvent

- **Sent by:** Camera
- **Handled by:** a LiDar Worker
- **Details:**
  - Includes DetectedObjects.
  - The Camera send DetectedObjectsEvent of the Objects with time T for all the subscribed Lidar workers to this event at time T, and one of them deals with a single event.
  - The LiDar gets the X's,Y's coordinates from the DataBase of them and sends a new TrackedObjectsEvent to the Fusion.
  - After the LiDar Worker completes the event, it saves the coordinates in the lastObjects variable in DataBase and sends True value to the Camera.

TrackedObjectsEvent

- **Sent by:** a LiDar worker
- **Handled by:** Fusion-SLAM
- **Details:**
  - Includes a list of TrackedObjects.

- Upon receiving this event, Fusion:
  - Transforms the cloud points to the charging station's coordinate system using the current pose.
  - Checks if the object is new or previously detected:
    - If new, add it to the map.
    - If previously detected, updates measurements by averaging with previous data.

PoseEvent

- **Sent by:** PoseService
- **Handled by:** Fusion-SLAM
- **Details:**
  - Provides the robot's current pose.
  - Used by Fusion-SLAM for calculations based on received TrackedObjectEvents.

TickBroadcast

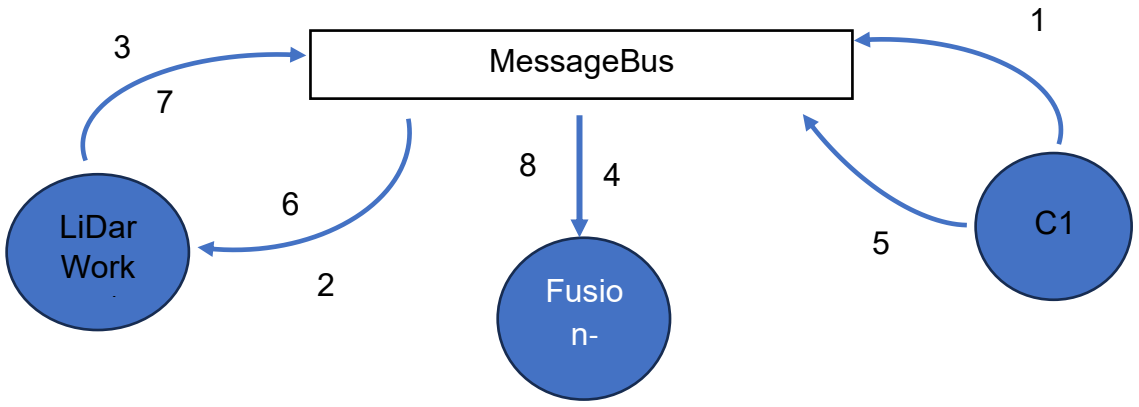
- **Sent by:** TimeService
- **Used for:** Timing message publications and processing.

TerminatedBroadcast

- **Sent by** all the sensors
- **Used for:** notifying all other services that the service sending the broadcast will terminate.

CrashedBroadcast

- **Sent by** all the sensors
  - **Used for:** notifying all other services that the sender service has crashed.



1) DetecionObjectsEvent
2) DetectoinObjectsEvent
3) TrackedObjectsEvent
4) TrackedObjectsEvent
5) DetectionObjectsEvent
6) DetecnObjectsEvent
7) TrackedObjectsEvent
8) TrackedObjectsEvent

MicroServices

TimeService

- **Purpose:** Global system timer handling clock ticks.
- **Responsibilities:**
  - Counts clock ticks since initialization.
  - Sends TickBroadcast messages at each tick.
  - Receives speed (tick duration in milliseconds) and duration (number of ticks before termination) as constructor arguments.
  - Stops sending TickBroadcast messages after duration ticks.
  - Signals termination of the process; do not wait for all events to finish.
  - **Note:** Ensure the event loop is not blocked.

CameraService

- **Responsibilities:**
  - Sends DetectObjectsEvents only.
  - Subscribe to TickBroadcast.

LiDarWorkerService

- **Responsibilities:**
  - Sends TrackedObjectsEvents (can be multiple events).
  - Subscribes to TickBroadcast.

FusionSlamService

- **Responsibilities:**
  - Does not send events.
  - Subscribes to TickBroadcast, TrackedObjectsEvent, and PoseEvent.
  - Manages the environmental map by processing tracked objects.

PoseService

- **Responsibilities:**
  - Holds the robot's coordinates at each tick.
  - Sends PoseEvents.
  - Subscribes to TickBroadcast.

Error Handling

- **Sensor Disconnection:**
  - If a sensor detects an error (e.g., cable disconnection indicated in the JSON files), it interrupts all other sensors, causing the system to stop.

- Affected MicroServices terminate and write an output file detailing the system's state before the error and indicating which sensor(s) caused the error.

Notes

- **Processing with Objects:**
  - MicroServices perform processing using the objects (e.g., CameraService uses Camera), so you may need to add necessary methods to those objects.
- **Flexibility:**
  - You may add additional MicroServices, messages, or objects, and add fields or functions to existing objects.
  - You may change members from private to public if justified.

Input Files

The system's input will be in JSON format. You can use any library to parse JSON, but GSON is recommended.
The input includes four JSON files:

1. Configuration JSON File

This file defines the overall configuration of the system and includes:

- **Cameras:** An array of cameras, where each entry contains:
  - id (integer): The unique identifier for the camera.
  - frequency (integer): The time interval (in ticks) at which the camera operates.
  - camera_datas_path (string): Path to the camera data JSON file
  - camera_key (string): The key in the camera data JSON file corresponding to this camera.
- LiDars:
  a. **LidarConfigurations:** An array of LiDAR workers, where each entry contains:
     i. id (integer): The unique identifier for the LiDAR worker.
     ii. frequency (integer): The time interval (in ticks) at which the LiDAR worker operates.
     iii. lidars_data_path (string): Path to the LiDAR data JSON file.
2. Pose JSON File:
   - poseJsonFile (string): Path to the pose data JSON file.
- Simulation Timing:
  - TickTime (integer): The duration of each tick in milliseconds.
  - Duration (integer): The total number of ticks the simulation will run.

2. Pose Data JSON File:

Describes the robot's current pose at each tick. Each entry includes:

- time (integer): The tick number.
- pose (array of three values):

    - x (double): The robot's x-coordinate.
    - y (double): The robot's y-coordinate.
    - yaw (double): The robot's orientation angle in degrees.

3. LiDar Data JSON File:

Contains the detected cloud points captured by each LiDAR worker at specific ticks. Each entry includes:

- time (integer): The tick number.
- cloudPoints (array of arrays): A collection of points, where each point is represented as [x, y, z].

4. Camera Data JSON File:

Contains the objects detected by each camera at specific ticks. Organized by camera ID, the file includes:

- **camera1**, **camera2**, etc.:
    - time (integer): The tick number.
    - detectedObjects (array of objects): Each detected object includes:
        - id (string): The unique identifier for the object.
        - description (string): A textual description of the object.

Main Function Argument

The path to the **Configuration JSON File** must be provided as an argument to the main function.

Output File

The output will be a single JSON file named **output_file.json**, located in the same directory as the input configuration file. It contains:

1. Statistics

- systemRuntime (integer): The total runtime of the system.
- numDetectedObjects (integer): The total number of objects detected by cameras.
- numTrackedObjects (integer): The total number of objects tracked by LiDAR workers.
- numLandmarks (integer): The total number of unique landmarks identified on the map.

2. World Map

- landMarks (array): The final map of the environment generated by the Fusion-SLAM service. Each landmark entry includes:
    - id (string): The unique identifier for the landmark.
    - description (string): A description of the landmark.
    - coordinates (array of points): The global coordinates of the landmark.

3. In Case of Error

If an error occurs during the simulation:

- Error (string): A description of the source of the error. Examples:
    - For camera-related errors, the description will specify the issue (e.g., "Camera disconnected").
    - For other sensors, it will specify the type of sensor and error (e.g., "Sensor X disconnected").
- faultySensor (array): A list of sensors (cameras or LiDAR workers) that caused the error.
- lastFrames (object): The last frames detected by:
    - cameras: Includes the most recent detectedObjects for each camera.
    - lidar: Includes the most recent cloudPoints for each LiDAR worker.
- poses (array): All the robot poses up to the tick where the error occurred.
- statistics: As described above.

Program Execution

- **Initialization:**
    - The program starts by parsing the input files and constructing the system.
    - Each MicroService subscribes to appropriate events and broadcasts.
    - The TimeService starts the clock.
- **Processing:**
    - Cameras send their data via DetectObjectEvents.
    - LiDAR Workers send their tracked objects.
    - Fusion-SLAM calculates the world map.
- **Termination Conditions:**
    - **Normal Completion:**
        - All sensors have no more data to read.
        - All sensors send TerminationBroadcast which the Fusion-SLAM recieves.
        - The Fusion-SLAM creates an output JSON file with statistics and the world map and then terminates. By then the system should terminate succefully.
    - **Sensor Error:**
        - An error occurs in one or more sensors, and it happened in the following scenarios:
            - In a camera's data there is an object of ID `ERROR` and the description will describe what happaned.
            - In the LiDar's data, there will be an object of ID `ERROR`.
        - A sensor which detected an error will send a CrashedBroadCast and when each other service receives the broadcast stops immediately.
        - When all the services are terminated, the system will output a JSON file as described above.

Part 4: Test-Driven Development (TDD)

Testing is an integral part of development. In this assignment, you are required to write unit tests for your classes using JUnit. This must be done for (at least) the following classes:

- Fusion-Slam – the method that transforms tracked objects to landmarks.
- Camera or Lidar - the method which prepares data before sending.
- MessageBus – the entire class.

Instructions

- **Location:** In Maven projects, unit tests are placed in src/test/java.
- **Steps:**
  - Download the interfaces.
  - Extract them.
  - Import the Maven project into your IDE.
  - Add tests (should be placed at src/test/java).
  - Submit your package with all classes.
    - Ensure your project is compiled with Maven.

Part 5: Submission Guidelines

Attached to this assignment is a project you can use to start working. To ensure your implementation is properly testable, you must conform to the package structure as it appears in the supplied project.

Submission Requirements

- **Files to Submit:**
  - All packages, including unit tests.
  - Your pom.xml file should include a dependency for JUnit.
- **Group Work:**
  - Submission is allowed only in pairs.
  - If you do not have a partner, find one.
  - You need explicit course staff authorization to submit individually.
  - You cannot submit in a group larger than two.
- **File Format:**
  - Submit a single .tar.gz file named assignment2.tar.gz.
  - Do not use other compression formats (e.g., .zip, .rar, .bz).
  - The compressed file should contain the src folder and the pom.xml file only (not inside an additional folder).
- **Extensions:**
  - Extension requests are handled by Hedi.

Building the Application: Maven

In this assignment, you will use Maven as your build tool.

- **Overview:**
  - Maven is the de-facto Java build tool and project management system.
  - IDEs like Eclipse, NetBeans, and IntelliJ have native support for Maven.
- **Instructions:**
  - Use the provided pom.xml file to get started.
  - Learn how to add dependencies to it (e.g., for JUnit).

Grading

Although you are free to work wherever you please, assignments will be checked and graded on CS Department Lab Computers. Ensure your program compiles and runs correctly on these machines.

Evaluation Criteria

- **Design and Implementation:**
  - Quality of your application design and code implementation.
- **Correctness:**
  - The application must be completed successfully and within a reasonable time.
- **Concurrency Issues:**
  - **Liveness and Deadlock:**
    - Understanding causes of deadlock.
    - Identifying potential deadlock points in your application.
  - **Synchronization:**
    - Appropriate use of synchronization.
    - Justification for your synchronization decisions.
    - Points may be deducted for overusing synchronization or unnecessary interference with program liveness.
- **Documentation and Coding Style:**
  - Adherence to guidelines detailed in "General Guidelines."

Note: There will be no automated tests. Your code will be tested with our input, and the output will be manually examined by the grader. You may assume all input will be valid.

GOOD LUCK!!!