SPL 221 - Assignment 2

Java Generics, Concurrency, and Synchronization

TA's in charge:
Or Dinari
Doron Cohen

Publication date: 25/11/2021
Deadline: 16/12/2021
Unit Test Submission Deadline: 5/12/2021

Before you start:

- The goal of the following assignment is to practice concurrent programming on the Java 8 environment. This assignment requires a good understanding of Java Threads, Java Synchronization, Lambdas, and Callbacks. Make sure you revise the lectures and practical sessions which cover these topics.
- **Before you write a single line of code, read the entire assignment.**
- While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine before submitting it.
- The Q&A of this assignment will take place at the assignment forum only. Critical updates about the assignment will be published on the assignment page on the course website. These updates are mandatory, and it is within your own responsibility to be updated.
- A number of guidelines for using the forum:
    - Read previous Q&A carefully before asking a new question; repeated questions will most probably go unanswered.
    - Be polite, remember that the course staff does this as a service for the students.
    - You are NOT allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss any such matters, please use the staff reception hours.
- Yair is the only staff member who can authorize extensions. In case you require an extension, please contact him directly.

Good Luck.

1) General Guidelines:

- Read the javadocs of all the interfaces we provided to you.
    - You must stick to the java documentation of each class and each method. For classes, you must add data members only with the allowed access levels. For methods, you must NOT change its return value type, parameters it receives, and the Exceptions types it throws.
    - You cannot throw exceptions that are not specified in the java documentation of the method. For example- if it is not stated in the documentation that a method throws an exception, then you must not throw an exception (in this case, DO NOT add the keyword throws in the header of the method).
    - You can add the word “synchronized” to any method if needed.
- You should try to make your code as concurrent and as efficient as possible.
    - Java introduces a collection of concurrent data structures that can help you in writing concurrent code. In this collection, the different data structures are implemented in such a way that different threads can use the data structure with as little synchronization and blocking as possible –(it is much more efficient than the naïve solution of synchronizing the methods of the data structure).
    - Read the Javadoc of each data structure you use – try to understand the level of concurrency each data structure allows (and if it is thread-safe at all) and the different features of it (.e.g, blocking data structure).
    - Try to synchronize as little as possible – you still need to use synchronization where it is not avoidable.
    - The performance of your implementation can affect your grade. However, efficiency must not come at the cost of the correctness of the required implementation.

2) Introduction:

In the following assignment, you are required to implement a simple Micro-Service framework, which you will then use to implement a system for managing the University compute resources.
The Micro-Services architecture has become quite popular in recent years. In the Micro-Services architecture, complex applications are composed of small and independent services which are able to communicate with each other using messages. The Micro-Service architecture allows us to compose a large program from a collection of smaller independent parts.

This assignment is composed of two main sections:

1. Building a simple Micro-Service framework.
2. Implementing a system for managing the university computer resources on top of this framework

3) Preliminary:

In this section, you will implement a basic Future class which we will use during the rest of the assignment. A Future object represents a promised result - an object that will eventually be resolved to hold a result of some operation. The class allows retrieving the result once it is available. When a MicroService finishes processing an event, it will resolve the relevant Future (see below).

Future has the following methods:

- *T get()*: Retrieves the result of the operation. This method 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, this method sets the result of the operation to a new value.
- *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, and then retrieve its result, if available. If the waiting time in longer than the timeout wil return null.

4) Part 1- Synchronization MicroServices Framework:

In this section, we will build a simple Micro-Service framework. A Micro-Service framework consists of two main parts: A Message-Bus and Micro-Services. Each Micro-Service is a thread that can exchange messages with other Micro-Services using a shared object referred to as the Message-Bus. There are two different types of messages:

Events:

- An Event defines an action that needs to be processed, e.g., training a Deep Learning model. Each Micro-Service specializes in processing one or more types of events. Upon receiving an event, the Message-Bus assigns it to the messages queue of an appropriate Micro-Service which specializes in handling this type of event. It is possible that there are several Micro-Services that specialize in the same events, (e.g., two or more Micro-Services that can handle an event of training a deep learning model). In this case, the Message-Bus assigns the event to a single Micro-Service of those which specialize in this event in a round-robin manner (described below).
- Each event holds a result. This result should be resolved once the Micro-Service to which the event was assigned completes processing it. For example- for the event of `Training a Deep Learning model' the Micro-Service in charge should return a trained model in the case that the training was successfully completed. The result is represented in the template Future object (T defines the type of the results).
- While a Micro-Service processes an event, it might create new events and send them to the Message-Bus. The Message-Bus will then assign the new events to the queue of one of the appropriate Micro-Services. For example: while processing a 'Training a Deep Learning model' event, the Micro-Service processing the event will need additional Data, Therefore, it must create an event for acquiring this data. The new event will be processed by another Micro-Service which can perform data

preprocessing; In this case the Micro-Service which generated the new event is interested in receiving the result of the new event in order to proceed, therefore it should wait until the new event is resolved (the computation completed) and retrieve its result from the Future object.

Broadcast:

Broadcast messages represent a global announcement in the system. Each Micro-Service can subscribe to the type of broadcast messages it is interested to receive. The Message-Bus sends the broadcast messages that are passed to it to all the Micro-Services which are interested in receiving them (this is in contrast to events that are sent to only one of the Micro-Services which are interested in them).

Round Robing Pattern:

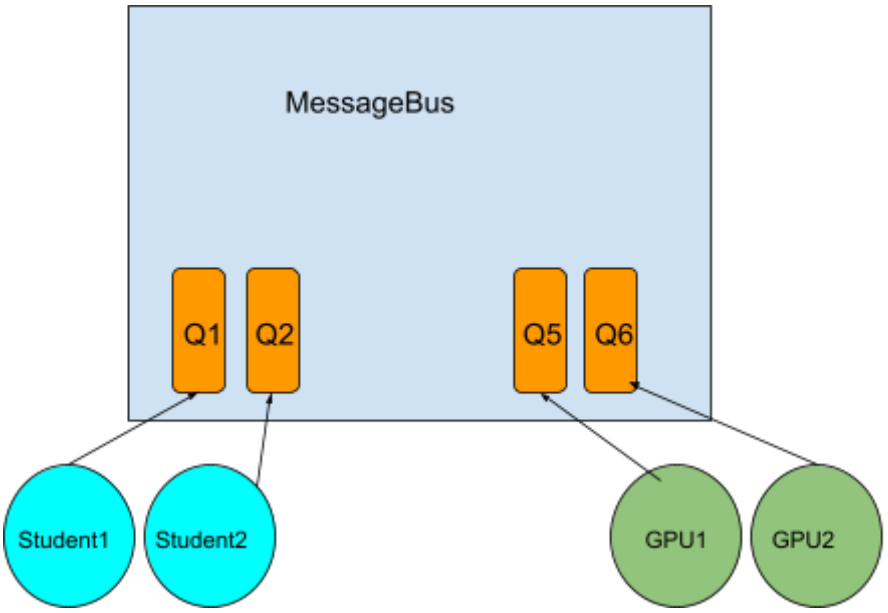
In a round-robin manner, we insert the event to one of the micro-services subscribed to receive it in round order. For example: if the event of type TrainModel has 4 services subscribed to receive it: ServiceA, ServiceB, ServiceC, ServiceD, and there are 6 models – TrainModel1, TrainModel2, TrainModel3, TrainModel4, TrainModel5, TrainModel6; Then: TrainModel1 will be inserted to ServiceA, TrainModel2 to ServiceB, TrainModel3 to ServiceC and TrainModel4 to ServiceD. After that we get back to ServiceA, so TrainModel5 will be inserted to ServiceA and TrainModel6 to ServiceB, and so on.

Example:

In part 2 of the assignment, you will implement several such microservices, three of them will be the following (note that this is just a partial description, for full description, see part 2):

- Student: Which creates the “TrainModel” event.
- GPU-machine: Which can process the “TrainModel” event.

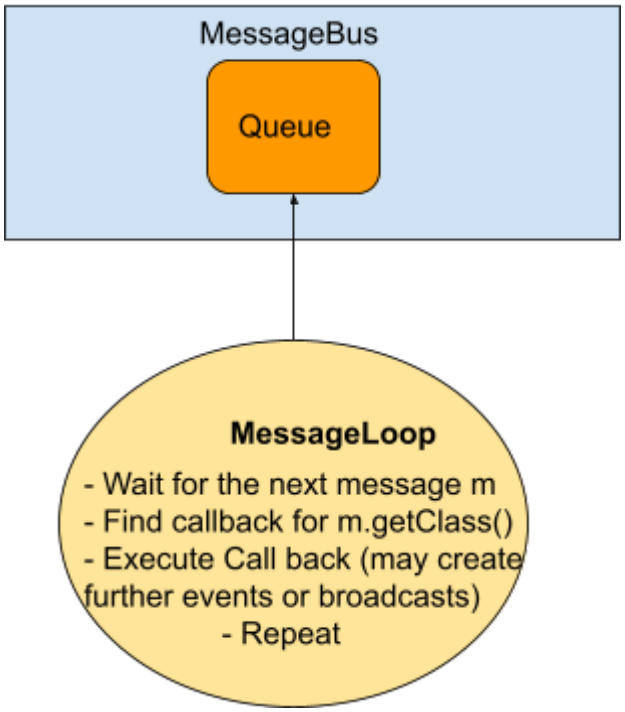
For simplicity, assume that there are only 2 instances of each of the above microservices. During initialization, each of the above microservices will register itself with the MessageBus, and will have a queue instantiated for him in the MessageBus, see the next figure for illustration:



When a student will send an event to “TrainModel”, the MessageBus will move it to one of the GPU queue, which will process the event.
When an event is sent, a future is returned to the sender, for example, sending the “TrainModel” event will return a Future<Model>, when the GPU is done handling the event, it will notify the MessageBus the event is completed, and set the Model instance in the future. Thus the student will know the training has been completed.

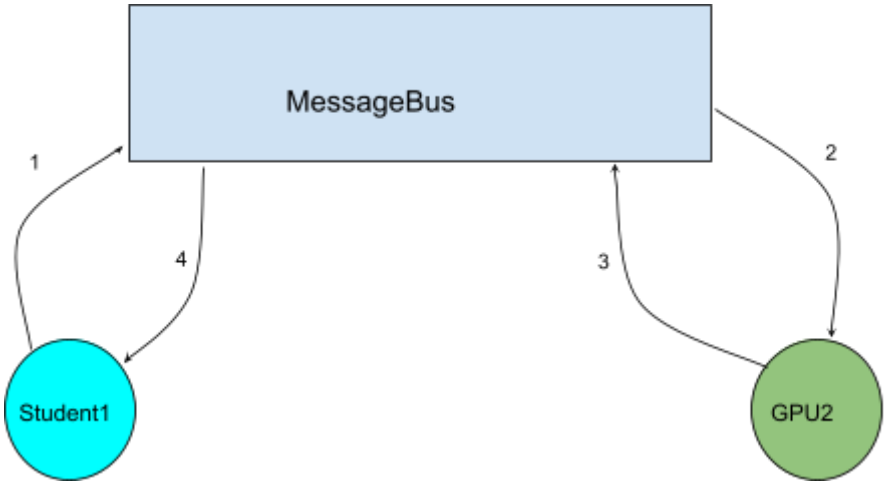
Message Loop:

In this part, you have to implement the Message Loop design pattern. In such a pattern, each micro-service is a thread that runs a loop. 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 next figure describes the Message Loop in our system.



Processing Events:

See the following figure for a simple flow of events:



1) Student Send the “TrainModel” event to the MessageBus, the MessageBus inserts the “TrainModel” event to GPU2 queue.
2) GPU2 receives the message from the Queue, and according to its type (TrainModel) calls the appropriate CallBack, this processes the event.
3) The GPU finishes processing the event and send the MessageBus that it is done, with the results of the event.
4) The MessageBus sets the future for the original event sent by Student1 (e.g. the TrainModel), with the results of GPU2.

Implementation Details:

To this end, in our framework, each Micro-Service will run on its own thread. The different MicroServices will be able to communicate with each other using only a shared object: A *Message-Bus*. The Message-Bus 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 Micro-Services will be able to subscribe for message types they would like to receive using the Message-Bus. The different Micro-Services do not know of each other’s existence. All they know of is 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 which writes software for end-users, they should now think like a programmer writing software for other programmers. Those other programmers will use this framework in order to build their own applications. For this part of the assignment, you will build a framework (write code for other programmers), the programmer which will use your code in order 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 are going to implement. The interfaces are located at the *bgu.spl.mics* package. Read the JavaDoc of each interface carefully. You are only required to implement the MessageBus and the MicroService for this part. The following is a summary and additional clarifications about the implementation of different parts of the framework.

- **Broadcast**: A Marker interface extending Message. When sending Broadcast messages using the Message-Bus it will be received by all the subscribers of this Broadcast-message type (the message Class)
- **Event**: A marker interface extending Message. A Micro-Service that sends an Event message expects to be notified when the Micro-Service that processes the event has completed processing it. The event has a generic type variable T, which indicates its expected result type (should be passed back to the sending Micro-Service). The Micro-Service that has received the event must call the method 'Complete' of the Message-Bus once it has completed treating the event, in order to resolve the result of the event.
- **MessageBus**: The Message-Bus is a shared object used for communication between MicroServices. It should be implemented as a thread-safe singleton, as it is shared between all the MicroServices in the system. The implementation of 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, 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 Message-Bus manages the queues of the 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, the Message-Bus should remove its queue and clean all references related to that Micro-Service. Once the queue is created, a Micro-Service can take the next message in the queue using the 'awaitMessage' method. The 'awaitMessage' method is blocking, that is, if there are no messages available in the Micro-Service queue, it should wait until a message becomes available.
    - *register*: a Micro-Service calls this method in order to register itself. This method should create a queue for the Micro-Service in the Message-Bus.
    - *subscribeEvent*: A Micro-Service calls this method in order to subscribe itself for some type of event (the specific class type of the event is passed as a parameter).
    - *subscribeBroadcast*: A Micro-Service calls this method in order to subscribe itself for some type of broadcast message (The specific class type of the event is passed as a parameter).
    - *sendBroadcast*: A Micro-Service calls this method in order to add a broadcast message to the queues of all Micro-Services which subscribed to receive this specific message type.

    - *Future<T> sendEvent(Event<T> e)*: A Micro-Service calls this method in order to add the event e to the message queue of one of the Micro-Services which 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 in order 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 to the result given as a parameter.
    - *unregister*: A Micro-Service calls this method in order to unregister itself. Should remove the message queue allocated to the Micro-Service and clean all the references related to this Message-Bus.
    - *awaitMessage(Microservice m)*: A Micro-Service calls this method in order to take a message from its allocated queue. This method is blocking (waits until there is an available message and returns it).
- **MicroService**: The MicroService is an abstract class that any Micro-Service 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 have a set of internal protected wrapping methods they can use. When subscribing to message types, the derived class also supplies a callback function. The MicroService stores this callback function together with the type 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 completes, the actual message-loop should start. The Micro-Service should fetch messages from its message queue using the Message-Bus's awaitMessage method. For each message, it should execute the corresponding callback. The MicroService class also contains a terminate 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:

- *All the callbacks that belong to the micro-service must be executed inside its own message-loop.*
- *Registration, Initialization, and Unregistration of the Micro-Service must be executed inside its run method.*

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: ExampleBroadcastListenerService, ExampleEventHandlerService, and ExampleMessageSenderService, and two messages: ExampleBroadcast and ExampleBroadcast.

5) Part 2- Compute Resources Management System:

Before implementing this part, see Part 3.

Overview of our system:

In this part, you build a system for managing the university compute resources. You will utilize the Micro-Service framework implemented in the previous section.
The description of the system is simple -
Students can create the “TrainModel” event in order to start training a model.
Once such an event is sent, a chain of events starts (described above, see ‘Processing Events’ in the previous section).
In addition, when a student finishes training a model, he can create another event “TestModel”, which will be handled by the GPU, and will return results.
if the results are good (defined randomly, see below) he can publish its results.
Results are submitted to a new type of Micro-Service, *`Conference’*, each conference will aggregate results from students, and on a set predefined time, will Broadcast the aggregated results to all the students.

Objects:

- **Student:** Represents a single student. Will have the following fields:
  - name: string - the name of the student.
  - department: string - the student department.
  - status: enum - for either MSc or PhD.
  - publications: int - number of published results (increases when the conference publishes the student results, not when the student sends the event).
  - papersRead: int - number of publications received from conferences, excluding his own.
- **Data:** Represents data used by the model:
  - type: enum - type of data, can be Images, Text, Tabular.
  - processed: int - Number of samples which the GPU has processed for training (see below).
  - size: int - number of samples in the data.
- **DataBatch:** Represents a Batch of data
  - data: Data - the Data the batch belongs to
  - start_index: int - The index of the first sample in the batch.
- **Model:** Represent a Deep Learning model.
  - name: string - name of the model.
  - data: Data - the data the model should train on.
  - student: Student - The student which created the model.
  - status: enum - can be “PreTrained”, “Training”, “Trained”, “Tested”.
  - results: enum - can be “None” (for a model not in status tested), “Good” or “Bad”.
- **ConferenceInformation:** Represent the information on a conference.
  - name: string - the name of the conference.
  - date: int - the time of the conference.
- **GPU:** Represent the a single GPU
  - type: enum - can be “RTX3090”, “RTX2080”, ”GTX1080”.
  - model: Model - the model the GPU is currently working on. (null for none)
  - cluster: Cluster - The compute cluster (see below)
- **CPU**: Represent a single CPU.
  - cores: int - number of cores.
  - data: Container<DataBatch> - the data the cpu currently processing. You can choose a container of your choice.
  - cluster: Cluster - The compute cluster (see below)
- **Cluster:** Represents the entire computer cluster, will be used for logging, statistics, and communication between GPUS and CPUS. A thread safe singleton.
  - GPUS: A collection of GPU - All the gpus in the system.
  - CPUS: A collection of CPU - All the cpus in the system.

- Statistics: You are free to choose how to implement this - It needs to hold the following information: Names of all the models trained, Total number of data batches processed by the CPUs, Number of CPU time units used, number of GPU time units used.

Messages:

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

- **TrainModelEvent:** Sent by the Student, this event is at the core of the system. It will be processed by one of the GPU microservices. During its process, it will utilize both the CPUS and the relevant GPU.
  As soon as the process starts, the GPU will send chunks of unprocessed data, in batches of 1000 samples, using the object *DataBatch* (which contains 1000 samples each), to the cluster, the CPUS in the system will take the data from the cluster, process it, and send back the processed data to the cluster.
  The CPU *DataBatch* processing time is determined by the CPU specifications:
  - Images - (32 / number of cores) * 4 ticks.
  - Text - (32 / number of cores) * 2 ticks.
  - Tabular - (32 / number of cores) * 1 ticks.

  When a CPU is done handling a batch of data, it will send it back to the cluster, the cluster will determine the relevant GPU, and send the processed data to the GPU.
  The GPU can then use the processed data to train the model.
  The time it takes to train each batch of data in the model is determined by the GPU type:
  - 3090 - 1 ticks.
  - 2080 - 2 ticks.
  - 1080 - 4 ticks.

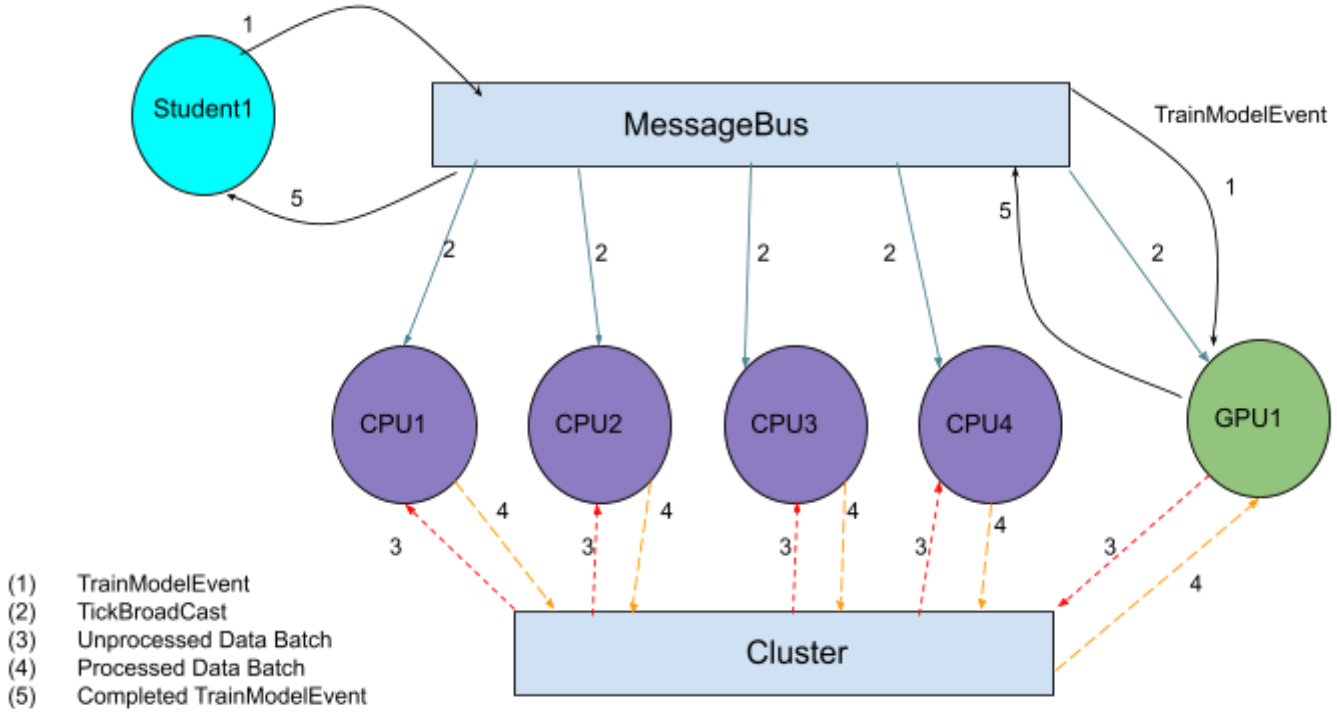
  When the GPU is done training with all the data, it will finish processing the event.
  The communication between the GPUs and the CPUs is done through the Cluster, there is no need to use the MessageBus for this high-paced communication.
  Note however the following limitation:
  Each GPU has limited memory (VRAM), and can only store a limited number of processed batches at a time, determined by its type:
  - 3090 - 32 batches.
  - 2080 - 16 batches.
  - 1080 - 8 batches.

  This limitation makes synchronization a challenging problem, specifically, if a GPU will send the entire data at once, it might clog the CPUs, preventing them from processing data from other GPUS.

As unprocessed batches are held at the Disk, and not in the more expensive VRAM, there is no limitation on the number of unprocessed batches a GPU can store, nor the number of batches it can send at a time.

You are free to implement the synchronization between the GPUS-CPUS-Cluster however you like, but you must have efficiency in mind. if GPUS will stand idle waiting for data, the processing will take longer. The grade will be deduced for solutions that are inefficient, e.g. a solution where the GPU does not utilize their time well.
Extremely efficient solutions will have their grades increased (to the limit of 100).

Example for such event:



Student1 sends the TrainModel event, the **Data** for the model is 10^6 Images.
GPU1 receives this event and starts processing it. GPU1 is of type `RTX3090`, it divides the data into batches of 1000 samples, each represented by **DataBatch** object, and stores them in its Disk.
To make sure it does not clog the CPUs, GPU1 will only send data if it has room to store it when it returns (e.g. after the CPUs processed it).
When a CPU finishes processing a batch, it sends the Cluster the processed batch, and takes a new batch for processing.

The cluster will send the processed batch to the GPU.

When this process is complete, e.g. the GPU sent all 1000 batches of samples to the CPUs, which have processed them, and returned them for training, and the GPU finished training on all those 1000 batches - it will set the future of TrainModelEvent as complete.

Note that the GPUs and CPUs communicate via the cluster, they are not allowed to directly call each other methods.

- **TestModelEvent:** Sent by the student, handled by the GPU microservice, this type of event is processed instantly, and will return results on the model, will return 'Good' results with a probability of 0.1 for MSc student, and 0.2 for PhD student. (yes this is random), when the GPU finish handling the event it will update the object, and set the future via the MessageBus, so the Student can see the change.
- **PublishResultsEvent:** Sent by the student, handled by the Conference micro service, the conference simply aggregates the model names of successful models from the students (until it publishes everything).
- **PublishConferenceBroadcast:** Sent by the conference at a set time (according to time ticks, see below), will broadcast all the aggregated results to all the students. After this event is sent, the conference unregisters from the system.
- **TickBroadcast:** Send on each tick (second, as calculated by the TimeService), used for timing the conferences publications and processing (by GPUs and CPUs).

MicroServices:

- **TimeService:** This Micro-Service is our global system timer (handles the clock ticks in the system). It is responsible for counting how many clock ticks passed since its initial execution and notify every other Micro-Service (that is interested) about it using the TickBroadcast. The TimeService receives the number of milliseconds each clock tick takes (speed:int) together with the number of ticks before termination (duration:int) as constructor arguments. The TimeService stops sending TickBroadcast messages after the duration ends, this signals the termination of the process. When terminating the process do not wait for all the events to finish. Be careful that you are not blocking the event loop of the timer Micro-Service. You can use the Timer class in java to help you with that. The current time always starts from 1.
- **StudentService:** Each student will have its own service, it is responsible for sending the TrainModelEvent, TestModelEvent and PublishResultsEvent. In addition, it must sign up for the conference publication broadcasts.
- **GPUService:** Each GPU will have this service, responsible for handling the “TrainModelEvent” and “TestModelEvent”.
- **CPUService:** Each CPU will have this service, its sole responsibility is to update the time for the CPU.
- **ConferenceService:** Each conference will have this service, responsible for aggregating good results and publishing them via the PublishConferenceBroadcast, after publishing results the conference will unregister from the system.

Note that the microservices will perform the processing using the objects, not by themself, e.g. a GPUService will use the GPU to do the actual processing, you will be required to add the necessary methods to those objects.

You are free to add additional MicroServices, Messages, Objects, or add fields/functions to Objects. You may change members from private to public if it is justified.

Input File:

The input will be in JSON format, you can read about its syntax here, you are free to handle it with any library you wish, however we recommend using the GSON library.
The input file json will contain the following fields:

- students - Array of students (with the fields described above), each will have an array containing the models the student intends to train and publish.

- GPUS - Array of GPU types in the systems.
- CPUS - Array of CPUS in the system, each with its cores.
- Conferences - Array conferences, each with its name and time of publication.
- TickTime - will include the time each tick will take (milliseconds).
- Duration - The duration of running the process.

Output File:

Your output will be a text file, containing:

- Each student name, with the details of the the models he trained, and which one of them has been published, in addition to the number of papers he has read.
- Each conference and its publications (e.g. name of models it published).
- GPU time used.
- CPU time used.
- Amount of batches processed by the CPUs.

Program Execution:

The program will start by parsing the input and constructing the system.
Each microservice will subscribe to its appropriate events/broadcasts, and the TimeService will start the clock. The students will send their models for training, and will only send the next model when the previous one finishes the entire process - Train, Test and Publish (if needed).
Conferences will aggregate publications from students, and at a set time (according to the input file) will publish all the results.
The program will finish execution according to the duration set by the input file, and will create the output file.

6) Part 3- TDD:

Testing is an important part of developing. It helps us make sure our project behaves as we expect it to. This is why in this assignment, you will use Junit for testing your project. You are required to write unit tests for the classes in your project.
This must be done for (at least) the following classes:

- MessageBus
- GPU
- CPU
- Future

You need to submit the unit tests by 5/12/2021 (before the deadline of the assignment).
Here are some instructions.
Notice: In Maven the unit tests are placed in src/test/java.
These are the steps:
1. Download the interfaces.

2. Extract them.
3. Import Maven Project in your IDE.
4. Add tests (may vary with different IDEs), should be placed at src/test/java.
5. Submit your package, with all the classes.
    a. Make sure you compile with Maven.

7) Part 3- Submission:

Attached to this assignment is a project. You can use to start working on your project. In order for your implementation to be properly testable, you must conform to the package structure as it appears in the supplied project.

- Submit all the packages (including the unit tests).
- You need to submit the Unit Tests as well, therefore your POM should include a dependency for the JUNIT.
- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff in order to submit without a pair. You cannot submit in a group larger than two.
- You must submit one '.tar.gz' file with all your code. The file should be named "assignment2.tar.gz". Note: We require you to use a '.tar.gz' file. Files such as '.rar', '.zip', '.bz', or anything else which is not a '.tar.gz' file will not be accepted and your grade will suffer.
- The submitted compressed file should contain the 'src' folder and the 'pom.xml' file only! no other files are needed. After we extract your compressed file, we should get one src folder and one pom file (not inside a folder).
- Extension requests are handled by Yair Vaknin.

Building the application: Maven

In this assignment, you are going to be using maven as your build tool. Maven is the de-facto java build tool. In fact, maven is much more than a simple build tool, it is described by its authors as a software project management and comprehension tool. You should read a little about how to use it properly. IDEs such as Eclipse, Netbeans and Intellij all have native support for maven and may simplify interaction with it - but you should still learn yourself how to use it. Maven is a large tool. In order to make your life easier, you have (in the code attached to this assignment) a maven pom.xml file that you can use to get started - you should learn what this file is and how you can add dependencies to it (and what are dependencies).

Grading:

Although you are free to work wherever you please, assignments will be checked and graded on Computer Science Department Lab Computers - so be sure to test your program thoroughly on them before your final submission. "But it worked fine on my windows-based

home computer" will not earn you any mercy points from the grading staff if your code fails to execute at the lab.
Grading will tackle the following points:

- Your application design and implementation.
- Your application must complete successfully and in a reasonable time.
- Liveness and Deadlock: causes of deadlock, and wherein your application deadlock might have occurred had you not found a solution to the problem.
- Synchronization: what is it, where have you used it, and a compelling reason behind your decisions. Points will be reduced for overusing synchronization and unnecessary interference with the liveness of the program.
- Check if your implementation follows the guidelines detailed in “Documentation and Coding Style”.

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

GOOD LUCK!!!