Information Systems Institute

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2013 (184.260)

Assignment 3

Submission Deadline: 7.6.2013, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the TUWEL forum) are allowed, but the code has to be written alone. If we find that two students submitted the same, or very similar assignments, these students will be graded with 0 points (no questions asked). We will use automated plagiarism checks to compare solutions. Note that we will also include the submissions of previous years in our plagiarism checks.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution compiles and runs without errors. If you are unsure, test compiling your submission on different computers (e.g., one of the ZID lab computers). Preferred target platform for the entire lab (all 3 assignments) is JDK 7[1]. If you decide to develop your code using JDK 6, some tweaks (endorsed libraries) may be required to get the project running under this version. Before you submit, make sure to test your solution with JDK 7.
- The assignment project is set up using Apache Maven[2]. We provide a project template that contains some code interfaces and JUnit tests which will assist you in developing your code. Please stick exactly to the provided interfaces as we will check your solutions in an automated test environment. The Maven project is split up into multiple modules which correspond to the different sub-parts of the assignments. **Note:** If all unit tests are passing in your solution, it does *not* necessarily mean that you will receive all the points. We will perform additional code checks and run tests that are not provided in the template. The tests included in the template should merely help you get started and assist you in developing your solution. If you want to further increase your test coverage, you may also add new unit tests, but this is optional. Do not modify any of the pre-defined tests - in any case, we will check your code with the original tests from the template.
- The root folder of the template contains the Maven metadata file (`pom.xml`) and four submodule directories (`ass3-aop`, `ass3-event`, `ass3-jms`, `ass3-shared`). Extract these directories and the pom.xml file to a clean folder on your disk - the code from the previous assignments is not required anymore for assignment 3.
- Use MySQL 5.1+[3] as your database management system.
- For the AspectJ development in Task 3, we suggest to use the Springsource Toolsuite (STS)[4], although we do not require it. However, STS eases the development of aspects and advices significantly, as STS includes tooling for syntax-checking and matching of joinpoint definitions (i.e., STS tells you without running the application what your current joinpoints actually match in your application).
- We expect (as can be seen in the persistence-unit configuration of persistence.xml) that you setup a database `dst` that can be accessed by user `root` (without a password). You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Again: make sure that all settings are as expected before you submit!
- Please make sure to add reasonable logging output to help us keep track of what your solution does. No debug output is very bad, and too much (e.g., many screen pages) is just as bad. Aim for a good middle ground, which allows us to check your solution quickly.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/`
[2] `http://maven.apache.org/`
[3] `http://dev.mysql.com/downloads/mysql/5.1.htm#downloads`
[4] `http://www.springsource.com/developer/sts/`

A. Code Part

Command for building/deploying part 1.:	`mvn install -Pass3-deploy`
Command for testing part 1. (in separate terminal):	`mvn install -Pass3-jms`
Command for testing part 2.:	`mvn install -Pass3-event`
Command for testing part 3.:	`mvn install -Pass3-aop`

1. Messaging (18 Points)

In this task you will create a simple JMS-based messaging application for the grid management system.

In the last assignment we have focused on assigning and processing grid computing jobs for a given number of CPUs. Now it is time to have a look at how the system processes jobs where the number of CPUs is not known in advance or was simply not specified.

Please study and complete the JMS configuration file provided in the template (**ass3-jms/src/main/resources/jms_config.xml**). Once you understand the concepts of JMS and Message-Driven Beans (MDB) you should be able to add all the **queues** and **topics** required for this task. An exemplary definition of one queue is already given in the file, so defining additional resources should be straightforward.

After a job is assigned by a user, the grid's scheduler takes care of scheduling and processing the job. To that end, the scheduler creates a new task (which wraps the job) and sends it to the clusters. One of the clusters takes the task, rates the complexity and decides whether one of its computers is able to process this task or not. The scheduler shall be informed if the task cannot be processed. In the other case (i.e., if the task can be processed), the task is forwarded to the computers. Every computer in our system belongs to one cluster and is responsible for a certain task-complexity. According to that, the task is processed by the respective computer. For example if cluster c1 rates a task as `EASY`, the task is processed by a computer that belongs to cluster c1 and is responsible for `EASY` tasks. After the computer has processed the task, the scheduler will be informed.

The steps in the procedure above are performed automatically in the provided test cases. As usual, feel free to extend the template by adding your own test classes (however, avoid changing existing tests).

Your task in the following is to design and implement the described communication based on a message queuing approach.

To keep things simple, there is only one persistent entity you need to manage in the application this time: the `Task` (Figure 1). Besides the mandatory id, this entity contains information about the job, a status, the name of the cluster, which was responsible for rating the task, as well as the complexity of the task. The status field represents the current state of the task in the process described above. The complexity field is set after the cluster in charge has rated the task. You do not need to implement these new entities – the interfaces and implementing classes are provided in the template (see module **ass3-shared**).

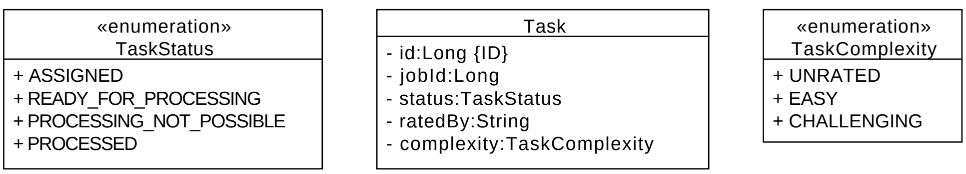

Figure 1: Task Entity

In our scenario a **server** provides the communication infrastructure. Whenever messages are exchanged (between the scheduler and the clusters or the cluster and the computers), the server is responsible for forwarding the messages. **Clients never communicate directly with each other**. This way it is also possible to keep the information about tasks up to date and to store the new status (and any other information that may have changed) to the database.

In total, the clients of our messaging system are (1) the scheduler, (2) possibly multiple clusters and (3) various computers, all of which may act as senders and receivers. Implement the `JMSFactory` methods for instantiating these components (interfaces are provided). Put your implementation of the respective interfaces into `impl` subpackages.

1.a. Scheduler

Our grid system contains a single scheduler which is responsible for assigning new jobs and providing information about existing tasks. Sending and receiving messages takes place concurrently. Use the method `IScheduler.start` to initialize the messaging infrastructure (lookup queues, create connections, etc.). Implement `IScheduler.stop` to close connections and release all resources. In addition, the interface in the template provides the following core methods (also refer to the comments inside the interface file):

- `IScheduler.assign(long jobId)`: Advises the server to create a new Task. The server creates an entry in the database and automatically forwards the task to the next available cluster. The status of the task is set to `ASSIGNED` and the complexity is set to `UNRATED`. In return to this command, the scheduler (asynchronously) receives an object message with a `TaskDTO` object, which contains the *id* of the newly created task.
- `IScheduler.info(long taskId)`: Advises the server to send information about a task identified by the respective id. This data must also contain all the relevant fields.
- `IScheduler.setSchedulerListener(ISchedulerListener listener)`: Sets a listener for the scheduler. The scheduler listener is implemented by the test classes provided in the template; the listener is used to notify about incoming messages from the server (`CREATED, INFO, PROCESSED, DENIED`).
- `ISchedulerListener.notify(InfoType type, TaskDTO task)`: Use this method in your `IScheduler` implementation to notify about status messages received from the server. Note that you do not need to implement `ISchedulerListener` - this interface is implemented by the test classes and serves as the link between the tests and your code.

1.b. Clusters

There may be several clusters listening to the server concurrently. Each of the clusters is identified via a unique name. The server does not know which or how many clusters there are at any given moment, but you can assume there will be at least one at any time. However, it is absolutely important that every task is handled by exactly one cluster **(not more!)**.

Once a cluster is entrusted with a new task (which is simply printed out to the console), the cluster automatically stops listening to the server. **A cluster is never executing more than one task at any time**. In the following we discuss the key interface methods:

- `ICluster.setClusterListener(IClusterListener listener)`: Sets a listener for this cluster. The cluster listener is implemented by the test classes provided in the template; use the listener to determine whether a task should be accepted or not, and to find out which complexity (either `EASY` or `CHALLENGING`) should be assigned to the task.
- `IClusterListener.decideTask(RateTaskDTO task, String clusterName)`: Use this method in your `ICluster` implementation to decide on how a task should be processed. Note that you do not need to implement `IClusterListener` - this interface is implemented by the test classes and serves as the link between the tests and your code.

Each time your cluster receives a task from the scheduler, invoke the listener's `decideTask` method to find out how to proceed. Then, based on the outcome of `decideTask`, the cluster should automatically add all required information to the `RateTaskDTO`. First, update the `ratedBy` field (the name of the cluster). Then, if the listener's decision indicates `ACCEPT`, update the complexity and set the status to `READY_FOR_PROCESSING` in the database. After the successful update, pass the task on to the computers. Otherwise, if the cluster is not able to process this task (i.e., `DENY` decision), do not send the task to the computers, but update the status to `PROCESSING_NOT_POSSIBLE` in the database (and do not update the complexity field). After the update, inform the scheduler about the denied task. On the scheduler side, call the listener's `notify` method to inform the test framework about the denied task.

1.c. Computers

Every Computer belongs to one Cluster and is responsible for exactly one task complexity. The server defines a special communication endpoint to let all computers listen for tasks that 1) are rated by the cluster they belong to, and 2) have the complexity they are responsible for. Therefore, the server application can simply label the request with the cluster's name (the task's ratedBy field), the complexity of the task and propagate it to this endpoint. You can assume that there is at least one computer (possibly more) listening for a certain cluster and complexity, and that all responsible computers compute the task simultaneously and collaboratively.

The computer must be designed in a way that it only receives messages it is responsible for (cluster and complexity), using the labels the server added to the message (check the possibilities to do this with JMS). Your infrastructure should also be able to deal with computers that are currently not listening, otherwise the request might get lost. The core interface methods are as follows:

- `IComputer.setComputerListener(IComputerListener listener)`: Similar to the listeners for clusters, the test framework in the template also injects listeners for computers. The computer listener provides a means to wait until the (simulated) task execution is finished.
- `IComputerListener.waitTillProcessed(ProcessTaskDTO taskObject, String computerName, TaskComplexity acceptedComplexity, String clusterName)`: This method blocks until a computer has successfully finished the processing of a task. Again, this listener interface is implemented by the tests (i.e., you do not have to implement it); simply use the listener in your computer instance to simulate the waiting time. After the simulated task processing has finished, the computer can finally update the task's status to `PROCESSED` and inform the scheduler. You can assume that the computers (which execute a task collaboratively) coordinate themselves to make sure each computer is finished before this command is sent. The server performs the status change immediately, which means that only one of the computers has to send the `processed` command. Hence, if more than one of the involved computers send this command, the first received command leads to the status change and the remaining ones have no effect (since the status is already set to `PROCESSED`).

You should now think about an appropriate message queuing infrastructure and decide about the features the respective queues and topics should provide to their clients. Your solution has to satisfy all the requirements stated above and should be as simple as possible. After configuring this communication infrastructure, implement the three components: scheduler, cluster, and computer.

Note: The persistent entities should never be used for transmission directly. Therefore, we provide DTOs in the template, containing only the relevant information (plain text messages are not sufficient in this assignment, you should be using object messages). The server has to update the information about the persistent tasks every time it retrieves relevant messages.

In case of failures (like unknown task ids, commands to already processed tasks, ...) no distributed communication is necessary. However, your application should be able to deal with such sort of requests.

2. Complex Event Processing (12 Points)

In the following we take a closer look at Complex Event Processing (CEP), a technique that is becoming more and more important in today's business processes and loosely coupled distributed systems. Generally speaking, an "event" is anything (i.e., a phenomenon, happening or similar) that is of interest for the application. CEP collectively refers to techniques for processing of event messages, including event routing, event aggregation, event pattern detection, etc.

The focus of this exercise is to perform event-based queries (which continuously process the new incoming events) and identify event patterns related to the processing of Grid computing tasks in our scenario. For this exercise, we slightly extend the processing flow of the messaging example, and assume that each `Task` assigned to the grid can traverse back and forth between states during its lifetime. In particular, we assume that a `Task` changes from state `ASSIGNED` to `READY_FOR_PROCESSING`, and then either to state `PROCESSED` (if everything goes well), or on to state `PROCESSING_NOT_POSSIBLE` and back to `READY_FOR_PROCESSING` (if an error occurs during processing). Overall, we are interested in the transition between states `ASSIGNED` and `PROCESSED`, and in the transition(s) between states `READY_FOR_PROCESSING` and `PROCESSING_NOT_POSSIBLE`.

Your implementation will be based on Esper[5], a popular open-source engine for CEP. Esper provides the Event Processing Language (EPL), a powerful SQL-like language which covers numerous features tailored to efficient processing of queries over event streams. The detailed Esper reference can be found here[6]. A good starting point for your own work is the Esper tutorial[7]. The required dependencies are already included in the template. This time, you do not need to store any entities to the database; simply keep the `Task` instances in memory using Esper. For simplicity, the following description speaks of `Task` instances, but in fact you should again use the Task DTOs (i.e., do not feed the entity classes into Esper).

The core classes/interfaces required in your code to properly interact with the test framework are `IEventProcessing` (initialization of queries, publication of events) and `EventingFactory` (object instantiation).

The detailed requirements are as follows:

- Initialize the Esper platform and obtain an instance of `EPServiceProvider` and `EPAdministrator`. Register the `Task` class from exercise 1 of this assignment as an event type with Esper. Provide a method to pass `Task` objects as events to Esper. You may choose an arbitrary name for the `EPServiceProvider` instance, e.g., "EsperEngineDST".
- Use the dynamic type definition capabilities of Esper (hint: `"create schema ..."`) to define three new event types. The names of the event types are `TaskAssigned`, `TaskProcessed` and `TaskDuration` (please use the constants in `dst.ass3.event.Constants`). Each of the three types should have a `jobId` property (which corresponds to `jobId` in the `Task` class). `TaskAssigned` and `TaskProcessed` have a `timestamp` property, and `TaskDuration` has a `duration` property. All of the mentioned properties are of type `long`. Note that you should register these event types in Esper *without* creating any corresponding Java classes.
- Create and execute 3 EPL queries which generate events for the three types `TaskAssigned`, `TaskProcessed` and `TaskDuration`. Evidently, `TaskAssigned` is triggered when a `Task` event with status `ASSIGNED` occurs, and `TaskProcessed` is triggered in case of a `Task` event with status `PROCESSED`. The `TaskDuration` events combine the information of the two previous task types and contain the processing duration. Be sure to correlate the `jobId` property of the `TaskAssigned` and `TaskProcessed` types in your query. You can assume that the `TaskDuration` query does not need to consider more than 10.000 historical events (i.e., there are never more than 10.000 events between any two correlated Task events).
- Implement the following three EPL queries and provide a listener which outputs (to stdout) new events that result from executing the queries:
  - Receive notifications about all events of type `TaskDuration`.
  - Upon arrival of each `TaskDuration` event, emit a `AvgTaskDuration` event (string defined in `dst.ass3.event.Constants`) that computes the average execution duration over all tasks that finished within the last 15 seconds.

[5] http://esper.codehaus.org/

[6] http://esper.codehaus.org/esper-4.5.0/doc/reference/en/html_single/

[7] http://esper.codehaus.org/tutorials/tutorial/tutorial.html

  - Use pattern matching facilities of EPL to detect tasks which have 3 times attempted and failed to execute (i.e., switched 3 times between the status `READY_FOR_PROCESSING` and the status `PROCESSING_NOT_POSSIBLE`).

- The test classes provided in the template simulate various non-trivial sequences of input events of type `Task`. The events are passed to your implementation of `IEventProcessing`, which is instantiated by the test classes using the factory `EventingFactory`. You can feed the received events directly to the Esper querier, i.e., for this task you do not need to use the messaging infrastructure from earlier to transmit and modify the status of the Tasks. Be sure to thoroughly test your solution - event-based and asynchronous processing is inherently prone to timing and synchronization faults.

Note that the example queries above cover only a very small subset of Esper's capabilities. Use the Esper reference to get familiar with some of the additional core concepts (e.g., *contexts*). During the interview sessions you should be prepared to report on your experiences with CEP and the strengths/weaknesses of Esper. Also, think about the core differences of the Esper processing model as opposed to querying a standard database like MySQL.

3. Dynamic Plugins Using AspectJ (14 Points)

Another important aspect of modern application servers is the dynamic loading and deployment of plugins or applications. This feature is also a nice demonstration for using reflection and class loading at runtime, so we will again implement a (simplified) custom solution of our own. Put the code for this task into the **ass3-aop** subproject.

- **3.a. Plugin executor (4 Points)**

  Implement the `IPluginExecutor` interface provided in the template. To allow the test framework instantiate instances of this interface, implement the factory method in `PluginExecutorFactory`. `IPluginExecutor` is the main component responsible for executing plugins. It has to monitor (i.e., repeatedly list the contents of) several directories to detect whether new **.jar** files were copied to these directories or existing .jar files were modified. (Note: In addition to monitoring for changes, also check the directories once when initializing your application. Note 2: Monitoring of sub-directories is *not* required.) The executor then scans the file and looks for classes that implement the `IPluginExecutable` interface. If some plugin executable is found, the executor spawns a new thread and calls its execute method. For this, you should be using a thread pool. Take care of class loading: there must not be any problem with the concurrent execution of different plugins containing classes with equal names. Also make sure to free all acquired resources after the execution of a plugin has been completed. The second method in the interface (`IPluginExecutor.interrupted()`) will be important later on, you can leave this method body empty for now. Note: if you detect that a plugin .jar file has changed and the plugin is currently still executing (the code in the previous version of the .jar file), you do not need to terminate the existing plugin instance (i.e., you can simply start a new version of the plugin with the updated code, and let the old one terminate normally).

- **3.b. Logging Plugin Executions (6 Points)**

  Now we want to implement some (decoupled) logging facilities for our plugin executor framework. To this end, we will make use of AspectJ and Aspect-Oriented Programming (AOP). The required AspectJ dependencies are already part of the Maven template. The tests in the template have been configured to use *run-time weaving* (i.e., objects are instantiated and then the aspects are dynamically weaved into the underlying class definitions of these objects). One alternative would be *load-time weaving* (i.e., using the Java agent mechanism to weave aspects into the classes at class loading time), which we do not use for this assignment.

  Before starting to develop, you should familiarize yourself with the concepts of aspects, advices, joinpoints and pointcuts (you can also tackle Theory Question 5 as you go along). Then, refer to the AspectJ Development Kit Developer's Notebook[8] for support on how these concepts are

[8] http://www.eclipse.org/aspectj/doc/released/adk15notebook/

implemented in AspectJ. Use the annotation-based development style[9] to define your aspects.

Your first task is now to write a simple logging aspect for plugins. Essentially, the aspect should write a single line of logging output before a plugin starts to execute, and after a plugin is finished. The bare class definition of `LoggingAspect` is already included in the template - add the required methods and annotations to this class. The log message can be very brief, but needs to contain the actual class name of the plugin:

```
[java] Plugin dst3.dynload.sample.PluginExecutable started to execute
[java] Plugin dst3.dynload.sample.PluginExecutable is finished
```

In some cases, users of the plugin framework might want to disable logging for some plugins. The template defines a method annotation `Invisible`. Whenever an `IPluginExecutable.execute()` is annotated as invisible, its execution should not be logged. Make sure that this condition is already considered in the pointcut definition of your logging advice (i.e., you should **not** match just any plugin method and filter out invisible plugins in your Java code).

Additionally, your logging aspect should re-use the logger of the plugin, if the plugin has defined one. That is, if the plugin has a member field of a subclass of `java.util.logging.Logger`, your log statements should be written to that logger. If no such logger is defined, use `System.out`. Configure the logging system to print all log messages of at least level INFO or higher.

- **3.c. Plugin Performance Management (4 Points)**

  Plugin frameworks like the one we are implementing often need some way to influence the execution of the managed plugins. Hence, we now implement some means to interrupt plugins whose execution takes too long.

  The template already defines a method annotation (`Timeout`), which has one Long parameter. Users can use this annotation on `IPluginExecutable.execute()` methods to define the "normal" maximum execution time of their plugins. Then, annotate and implement the aspect class `ManagementAspect` (see template) to enforce this defined maximum execution time. Have the aspect hook into each invocation of `IPluginExecutable.execute()` to keep track of the currently running plugins and their start time. From the start time you can then derive the current execution duration, e.g., by polling in regular intervals or using a timer task. If a plugin is detected that takes longer than its maximum defined time, call this plugin's `interrupted()` method. You do not need to take any further action (i.e., we can assume that the developer of the plugin actually terminates the plugin if this callback is invoked). However, keep in mind that the `interrupted()` method can itself take some time to execute, so do not block while the client is cleaning up. If no timeout is defined for a plugin, you can assume that the plugin can run for as long as it needs to.

The package `dst.ass3.aop.sample` in the template contains a couple of simple plugin examples for testing. You may optionally add your own plugin classes to test your solution more thoroughly.

[9] http://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj.html

B. Theory Part

The following questions will be discussed during the practice lesson. At the beginning of the each lesson we hand out a list where you can specify which questions you have prepared and are willing to present. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a correct and well-founded answer, you will lose **all** points for the theory part of this assignment.

4. Class loading (1 point)

Explain the concept of class loading in Java. What different types of class loaders do exist and how do they relate to each other? How is a class identified in this process? What are the reasons for developers to write their own class loaders?

5. AOP Fundamentals (2 points)

Explain the concept of Aspect Oriented Programming (AOP). Think of typical usage scenarios. What are aspects, concerns, pointcuts and joinpoints, and how do these concepts relate to each other? Why is it so important to write minimally matching pointcut definitions?

6. Weaving Times in AspectJ (1 point)

What happens during weaving in AOP? At what times can weaving happen in AspectJ? Think about advantages and disadvantages of different weaving times.

7. Esper Processing Model (2 point)

Study the details of the Esper processing model (available in the online reference of Esper). Describe the core API elements, and illustrate the main EPL query types based on an exemplary event timeline.