Information Systems Institute

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2011 (184.260)

Assignment 2

Submission Deadline: 3.5.2011, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the forum[1]) are allowed but the code has to be written alone.
- No deadline extensions are given. Start early and after finishing your assignment upload your submission as a zip file in the DSG-Tool[2]. 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 at a later point.
- Make sure that your solution includes all the libraries and dependencies and it compiles without errors. If we cannot compile your solution you cannot get all the points.
- Before solving the tasks concerning Enterprise Java Beans, we recommend reading Part IV of the Java EE 6 Tutorial[3]. If you are already familiar with EJB 3.0, this link[4] may give you a good impression of the new features in EJB 3.1 (find parts 1-4 of this series linked in the reference section there). However, note this has been published as a preview: not every feature mentioned made it into the final specification.
- Concerning Messaging, we still recommend reading chapter 31[5] and chapter 32[6] of the old Java EE 5 Tutorial as the latest version of the tutorial omits mentioning many details in this regard.
- You should use Java 1.6.14+[7], MySQL 5.1[8] as your database management system and GlassFish v3[9] as your application server. Task 1 explains how to set up Glassfish in detail.
- For your solution, use the provided project stubs:

  All needed libraries are already included if you want to use another one, feel free to integrate it. We expect (as you can see in the persistence-unit configuration of persistence.xml) that you setup a database dst that can be accessed by 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). Build scripts for ant[10] are already included, you may modify them, but the submitted solution has to be compilable and runnable with the predefined targets.

[1] https://www.infosys.tuwien.ac.at/teaching/courses/dst/forum/
[2] https://stockholm.vitalab.tuwien.ac.at/dsg-teaching-web/student
[3] http://download.oracle.com/javaee/6/tutorial/doc/bnblr.html
[4] http://www.theserverside.com/news/1363649/New-Features-in-EJB-31-Part-5
[5] http://download.oracle.com/javaee/5/tutorial/doc/bncdq.html
[6] http://download.oracle.com/javaee/5/tutorial/doc/bncgv.html
[7] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[8] http://dev.mysql.com/downloads/mysql/5.1.html#downloads
[9] http://glassfish.java.net/downloads/v3-final.html
[10] http://ant.apache.org/

A. Code Part

1. Enterprise Java Beans 3.0 (26 Points)

Your task is to develop an enterprise application for the grid management system introduced in Assignment 1. Reuse your entity classes from task 1 (basic mapping and inheritance). Code from any other tasks is no longer required.

Firstly, install GlassFish and set GLASSFISH_HOME to the '**glassfish**'-directory right inside (!) the directory you installed GlassFish to. Start the application server by calling '**asadmin start-domain**' from the **bin**-directory, and, before calling '**ant setup**' from our build-file, make sure the file '**dst-ds.xml**' in the setup-directory is configured correctly. After a restart ('**asadmin restart-domain**') everything should be configured correctly. Also take a look at the GlassFish administration console at http://localhost:4848. By default, GlassFish uses the following file for logging (which may be important for debugging your solution):

- **GLASSFISH_HOME/domains/domain1/logs/server.log**

Please study the build file (**server/build.xml**) to understand the deployment process and complete the dist target, such that the classes the client needs (and only those) are included within the client library. In the client build file (**client/build.xml**) you have to complete the run target.

Put the server code into the **1_ejb/server** project, the client code into the **1_ejb/client** project. Make reasonable decisions concerning the type of session bean (stateful, stateless, singleton) to use for each task, and whether the beans should be remotely or locally available. Follow the principles of minimal visibility and minimal accessibility.

1.a. Session beans (14 Points)

- **Create a bean for testing purpose (TestingBean)**:

  Provide a method that inserts at least two grids, two users, two memberships, two clusters, five computers and one job, whose execution has started 30 minutes ago and has not finished yet (end field is set to null). Obviously, storing a grid may require to store some other entities as well. To test your solutions of the following tasks you may come back to this bean at a later point to store additional data. This method may be invoked by your client directly.

- **Create a bean for managing prices (PriceManagementBean)**:

  Our system requires a manageable pricing model for the charges of using the Grid. We assume that the user has to pay two different payments for every assigned job. As a sort of down payment, the user first pays a small fee when assigning a job. The purpose of this fee from the Grid provider's viewpoint is to hedge against the risk of non-payment. Hence, new users have to pay more than long-term users who have already proven to be reliable. In particular, this fee decreases with increasing number of jobs a user has submitted (and paid) in the past. For the second fee, the user's account is debited with the variable price for the job's execution time after the job has finished.

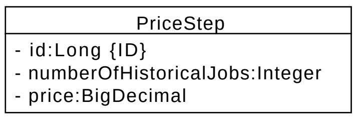

Figure 1: PriceStep Entity

  The PriceManagerBean is a helper to adminster the "steps" of the price curve. This bean provides a method to store the price steps in the database, according to the number of jobs the user has previously executed. (e.g.: 30€ for less than 100 executed jobs, 15€ for 100-1000 jobs, 5€ for 1000-5000 jobs ...). The *PriceStep* entity that should be stored is depicted in Figure 1.

The PriceManagementBean should also offer a method to get the fee for a given number of executed jobs. This bean only serves as a helper for the beans you will implement next. All prices should be retrieved from the database once the application server initializes the application and after that stored in memory (to avoid time-consuming database reads at runtime). At this point, creating the required persistent entity should be straightforward. The concrete implementation of the bean should be designed by you. In the end, the bean has to provide a possibility to set prices for a number of historical jobs and get the fee for a specified number of historical jobs.

Note that especially the last method (retrieving fees) may get called quite often, so you should keep performance in mind and think about the default behavior of concurrency and transactions managed by the container. However, as a change of prices is not expected to happen all that often, you can directly store new values in the database (do not forget to update your in-memory data structures, which you have loaded at application server startup, though!).

- **Create a bean for general management concerns (GeneralManagementBean)**:

  For now, this bean only has to provide a way to set prices using the bean you just created. We will extend this bean at a later point. However, note that all methods the bean will provide will share no state and will be invoked independently of each other. This bean should also be invokable by the client directly.

- **Create a bean that allows users to assign jobs for several grids (JobManagementBean)**:

  First of all this bean provides a method for the user to login with username and password.

  For more convenience the system provides the possibility to assign several jobs in a single transaction. To do this the user can add jobs for a single grid to a temporary job list by specifying the id of the grid, the number of CPUs, the workflow and parameters (as list) of the job. Think of the temporary job list as a 'shopping card' for jobs: users add jobs to the list one after another (possibly over a longer timespan) and submit them all in one go.

  When the bean receives requests to add jobs to the temporary job list, it first has to check if there are enough free computers for this grid left. To do this check if the sum of the CPUs of all free computers in the grid is larger than (or equal to) the CPUs necessary for the job. If this is not the case the job cannot be scheduled now, and the user is informed (throw a meaningful custom exception). If it is possible to execute the job the bean has to assign it to concrete computers. In this assignment we will ignore complex scheduling issues. You can simply query all free computers from the database and start assigning free computers at random until the sum of CPUs of all assigned computers is equal to (or larger than) the number of CPUs required by the job.

  After the list has been submitted to the system, all jobs in the list are started immediately (set the executions start field to now and the status to `SCHEDULED` – it is not possible to submit jobs where the scheduling starts in the future). When submitting, make sure to check again if the jobs can still be scheduled the way you planned (since the system has many users it is possible that another user was scheduling jobs in parallel using the same computers). Think of a suitable protocol to achieve this. If you find out that it is not possible anymore to execute the job, notify the user with an exception, as above.

  You should also provide a way to remove all jobs for a specific grid from the temporary job list, as well as a method to get the current amount of jobs for each grid in this list.

  Note that no data is written to the database before the temporary job list is finally submitted. So provide a method to do the final submission. Before the final submission, the user has to login at some point in the conversation. When the assignments are finally submitted, store the data (jobs and executions) to the database only if all chosen computers are available. Make sure this method executes transactionally secure, so that no data is written to the database if something goes wrong (i.e.: computers are not available). If the submission was successful, discard the bean. Otherwise, throw meaningful exceptions so that the user can react to this and modify the job assignment (in case of an exception the bean should not be discarded!).

1.b. Timer Service and asynchronous method invocations (7 Points)

- **Implement a timer service used for simulation**

  Up to this point we only assigned jobs to the grid. Now, in order to test our solution, we need to simulate that jobs are actually going to finish at some point. For this we use a timer service. The timer service periodically (for instance every minute) takes the jobs that are running (start is set but end is null) and completes them (i.e., set the status to `FINISHED`, set the end date, and so on).

- **Provide a method to retrieve the bill of a user**

  Now you should extend the GeneralManagementBean that we have started earlier. Implement a method which can be used to retrieve the total bill of a user (given by username). That means that for each of this user's finished but unpaid jobs compute the costs for the execution (with the grid's `costsPerCPUMinute`) and sum them up. Additionally, add the scheduling costs (the static costs that incurred for assigning the jobs in the first place). Use your PriceManagementBean to calculate the scheduling costs. Don't forget the user's discount, if he has a membership for a certain grid. Return the bill as a simple String. The bill should contain the total price, the price per job, the setup costs and execution costs and the number of computers that have been used per job. As soon as the bill for one job is finished you can set the payment status of this job to paid.

  As collecting this data may take some time, the method should be executed asynchronously. The client simply invokes this method, and immediately gets control back. This way, the client can continue processing while the calculation is running, and receives the collected data asynchronously when it's finished. Check out the possibilities that EJB 3.1 provides for this purpose.

1.c. Audit-Interceptor (5 Points)

- **Develop an audit interceptor and apply it to the JobManagementBean**:

  Since the JobManagementBean is essential to our system we now implement some simple logging to get some insight into the internal workings of the bean. We implement our logger as an audit interceptor. The interceptor should persist the invocation time, method name, parameters (index, class and value) and result value (or exception value in case of failure). You may simply invoke the toString() method of objects to convert results to persistable strings (but make sure to check for null values). Note that transactions are usually rolled back in case of an exception, and that this behavior would also influence our interceptor by default. Therefore, check how to bypass this behavior to be able to persist the audit even in case of a failure (e.g., if a job cannot be scheduled). Add a method to the GeneralManagementBean to retrieve all these audits (do not directly return entities, but create simple data transfer objects!).

1.d. Client application

Now the only thing left to do is write a client application to test our EJB system.

The included **appserv-rt.jar** already contains a preconfigured jndi.properties file sufficient for your needs. So calling `Context ctx = new InitialContext();` is enough, you dont have to supply any properties. JNDI names for remote beans are standardized since Java EE 6; look them up using the following pattern: `java:global[<app-name>]/<module-name>/<bean-name>#<interface-name>`, e.g. `java:global/dst2_1/TestBean`.

In your client program you have to execute the following steps (users hit <Enter> to proceed to the next step):

- Insert your test data.

- Use your GeneralManagementBean to set some prices.
- Afterwards obtain a reference to your JobAssignmentBean, try to login with invalid values, then login successfully, add some valid job assignments, request the current assigned amount of jobs and finally successfully submit your temporary job list.
- Replay the last step with a different user, but this time try to assign more jobs for a grid than there are free computers. Delete the job assignments for a grid, request the assigned amount of jobs and finally successfully submit your temporary job list.
- Wait for some time so that your jobs are finished.
- Use the GeneralManagementBean to get the bill for all finished jobs.
- Finally get all saved audits from the Audit-Interceptor.

For each step you need to provide reasonable output (so we can easily keep track of what is going on). Please test properly, in order to avoid problems when we check your assignment. However, your solution will be run only once, so you may hardcode IDs.

2. Messaging (18 Points)

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

Please study the build file (**server/build.xml**) to understand the deployment process and complete the dist target, such that the classes the clients need are included within the client library. In the clients build files (***/build.xml**) you have to complete the specified run-*-targets we will use to start your clients with.

Another file to study is the JMS-configuration file (server/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.

In the first part of the assignment we have focused on assigning and processing jobs for a given number of CPUs. Now it's 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.

After a job is assigned by a user, the grid's scheduler takes care of the scheduling and processing of the job. To do that 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 his computers is able to process this task or not. If the task can't be processed the scheduler will be informed. On the other hand (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. When the computer has processed the task, the scheduler will be informed.

In reality, the steps in the procedure above can be performed mostly automatically. However, for testing purposes and to illustrate the correct functioning of your solution, you will simulate the procedure using commands that are typed manually via a command line client (see below 2.a, 2.b and 2.c).

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 2). Besides the obligatory 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.

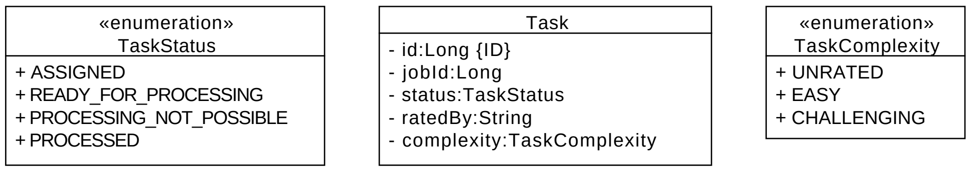

Figure 2: Task Entity

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.

2.a. Scheduler

At any point in time, there is **exactly one** active scheduler that sends to and receives messages from the server. Sending and receiving messages takes place concurrently. To send messages, the application listens to user input. The scheduler provides the following console (command line) commands:

- `assign <job-Id>`

  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 receives the *id* of the newly created task. This id is needed for the following request (it is enough to simply write the id to the console).

- `info <task-Id>`

  Advises the server to send information about a task identified by the respective id. This data must also contain all the relevant fields. It is sufficient to print the information to the console, again.

- `stop`

  Exits the application.

2.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!)**. Do not implement any additional commands than the ones stated in the following for this.

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 doing more than one task at any time**. The cluster provides the following commands:

- `accept <task-complexity>`

  Indicates that the cluster rated the task with either `EASY` or `CHALLENGING` and sends a message to the server. The cluster application should automatically add all required information to this message so the server can identify the respective task, update the ratedBy field (the name of the cluster), the complexity and the status to `READY_FOR_PROCESSING` in the database. After the update the server sends the task to the computers.

- `deny`

  Indicates that the cluster's computers are not able to process this task. The application sends a message to the server and automatically adds all required information to this message so the server can identify the respective task, update the ratedBy field (the name of the cluster) and the status to `PROCESSING_NOT_POSSIBLE` in the database (the complexity field shall not be updated). After the update the server informs the scheduler about the denied task. On the scheduler side, it's enough to print the information to the console.

- `stop`

  Exits the application.

2.c. Computers

Every Computer belongs to one Cluster and is responsible for exactly one task complexity. The server defines a special communication endpoint for all computers that are listening for tasks that were rated by their cluster and have their task complexity. 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 that messages that are intended for it (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 Computer supports the following commands:

- `processed <task-id>`

  Indicates that the computers have finished the processing of the task successfully. The server can finally update the task's status to `PROCESSED` and informs 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`).

- `stop`

  Exits the application.

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. However, your solution has to satisfy all the requirements stated above and should be as simple as possible. After configuring this communication infrastructure, implement the server application and all three clients.

Never use the persistent entity for transmission directly. Instead create DTOs yet again, containing the relevant information only (plain text messages are not sufficient in this assignment). 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.

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.

3. EJB Lifecycles (1 Point)

Explain the lifecycle of each bean type defined in the EJB 3.1 specification. What optimizations can the EJB container perform for the respective type? Also think about typical use cases the respective bean type provides to the developer.

4. Dependency Injection (2 Point)

Explain the way dependency injection is performed by the EJB container. What kind of resources may be injected into a bean, and what are the different annotations that can be used?

5. Java Transaction API (2 Point)

The EJB architecture provides a mechanism for distributed transactions. Explain the two ways how transactions can be defined. How is the concept of distributed transactions accomplished behind the scenes, i.e. what tasks have to be performed by the EJB container?

6. Security (1 Point)

Security is another important requirement every application server has to provide some solutions for. Explain the different annotations provided by EJB for this purpose and think about typical use cases. Is using these annotations all you need to make your application secure?