From LU Verteilte Systeme (184.167)

Lab2: Lab2

General Remarks

- We suggest reading the following tutorials before you start implementing:
  - **Java RMI Tutorial**: A short introduction into RMI.
  - **JGuru RMI Tutorial**: A more detailled tutorial about RMI.
- Group work is **NOT** allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the forum) are allowed but the code has to be written alone.
- Be sure to check the **Tricky Parts** section for questions!

Submission Guide (for all Labs)

Submission

- You must upload your solution using the **Teaching Tool** before the submission deadline: **25.11.2010, 18:00 CET. -** the deadline is hard! You are responsible for submitting your solution in time. If you do not submit, you won't get any points!
- Do not confuse our lab server (`pasta.dslab.tuwien.ac.at`) with the **Teaching Tool**. The lab server is just for testing purposes. We cannot grade any solutions uploaded there.
- Upload your solution as a **ZIP** file. Please submit only the sources of your solution and the build.xml file (not the compiled class files and no third-party libraries).
- Your submission must compile and run in our lab environment. Use and complete the provided **ant template**.
- Test your solution extensively in our lab environment. It'll be worth the time.
- Before the submission deadline, you can upload your solution as often as you like. Note that any existing submission will be **replaced** by uploading a new one.
- Please make sure that your upload was successful (i.e., you should be able to download your solution - as the tutors will do during the interview).

Interviews

- After the submission deadline, there will be a mandatory interview (Abgabegespräch). You must register for a time slot to the interviews using the **Teaching Tool**.
- You can do the interview only if you submitted your solution before the deadline!
- The interview will take place in the **DSLab**. During the interview, you will be asked about the solution that you uploaded (i.e., **changes after the deadline will not be taken into account!**). In the interview you need to explain your code, design and architecture in detail.
- Remember that you can do the interview **only once**!

Description

In this assignment you will learn:
- the basics of a simple distributed object technology (RMI)

- how to bind and lookup objects with a naming service
- how to implement callbacks with RMI

Overview

In this lab we will build a simple distributed event scheduler similar to **doodle**. The technology we will use for this is Java RMI.

An 'event' in the following description does not denote an event in the technical sense (as in event-driven systems), but refers to some gathering of people for a certain purpose.

Imagine the following scenario: A distributed server system administers events and users. Each server is responsible for different events and users, that is, each event in the system is known to only one server - the same holds true for each single user. An event is identified by a system wide unique name. Furthermore, it contains information about the place to meet, the estimated duration and - most important - a number of possible dates. The user that has created a particular event can invite other users to join the meeting and vote on one or more dates that fit best to this user's busy schedule. After some time the author may decide to finalize the event schedule: The system then automatically calculates the date appropriate to the most attendees and notifies each participating user about the end of the voting.

Have a look at the following two figures: User Alice wants to be part of our event scheduler network and therefore has to register in the system (1.), providing a username and a password (for the sake of simplicity, some details like the password have been omitted in the figures). Since usernames, just like event names, are globally unique in the system, the server Alice is connected to now initializes a 2-phase commit transaction (for details see below) to verify the username *Alice* is available and was not registered on any other server before (2.). Because this is not the case (3.), the server can commit this new username (4.). The other servers can store this information for further requests, i.e., to directly address the server responsible for a particular user. After successful registration (5.), Alice can finally log in by providing her username and the correct password (6.-7.).

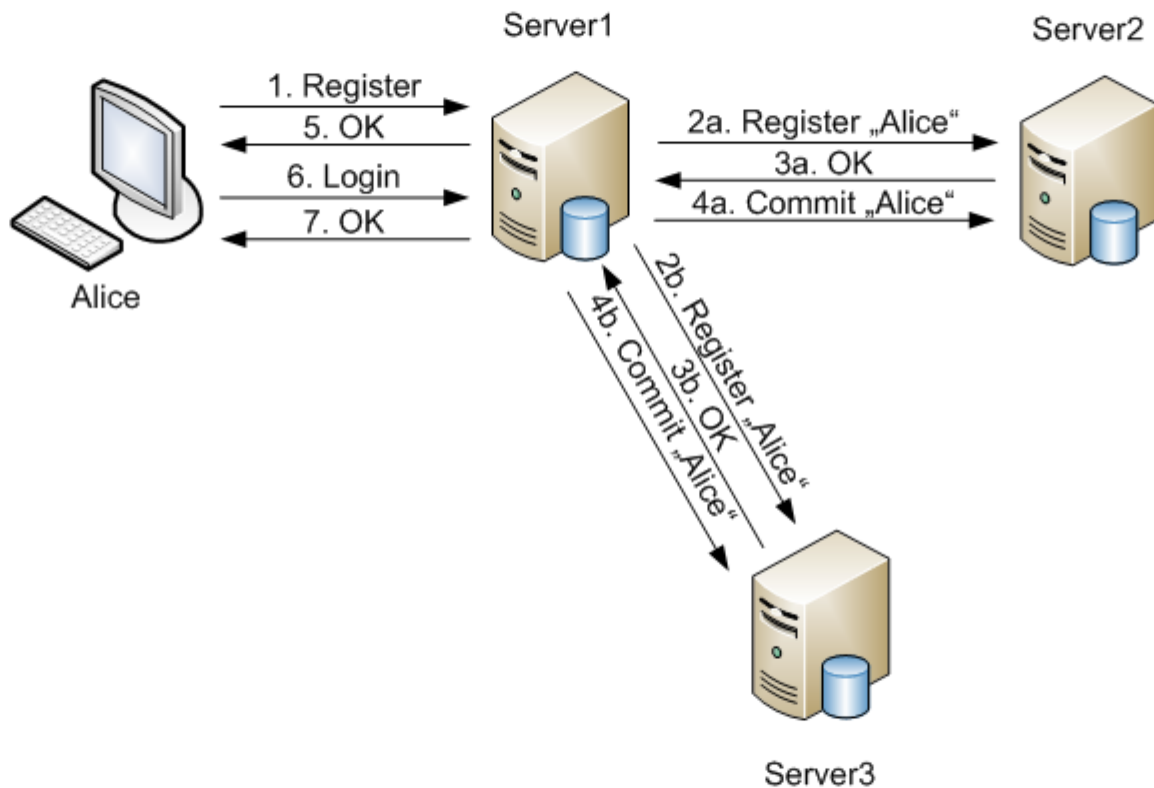

Figure 1: An example of a distributed namespace and the registration of a new user

For Figure 2, assume user Bob has just created a new event. Bob now invites user Bill to participate and vote for one or more possible time slots (1.). User Bill is managed by the same server, therefore the server can directly send him Bob's invitation (2.). Next, Bob decides that Alice should also join the event (3.). Since both users are registered on different servers, Bob's server first has to forward his invitation to Alice's server (4.), which finally submits the invitation (5.). After having a short look at the date options (not part of the figure), Alice gives her vote on the event (i.e., replies to the event invitation) (6.) which again needs to be forwarded to Bob's server which administers the event (7.). Other users may vote on the event until Bob finalizes it (8.). The server therefore calculates the best date and notifies all voters that the event is finally scheduled. Since Alice is registered on a different server, this notification again requires forwarding it to another server first (9.).

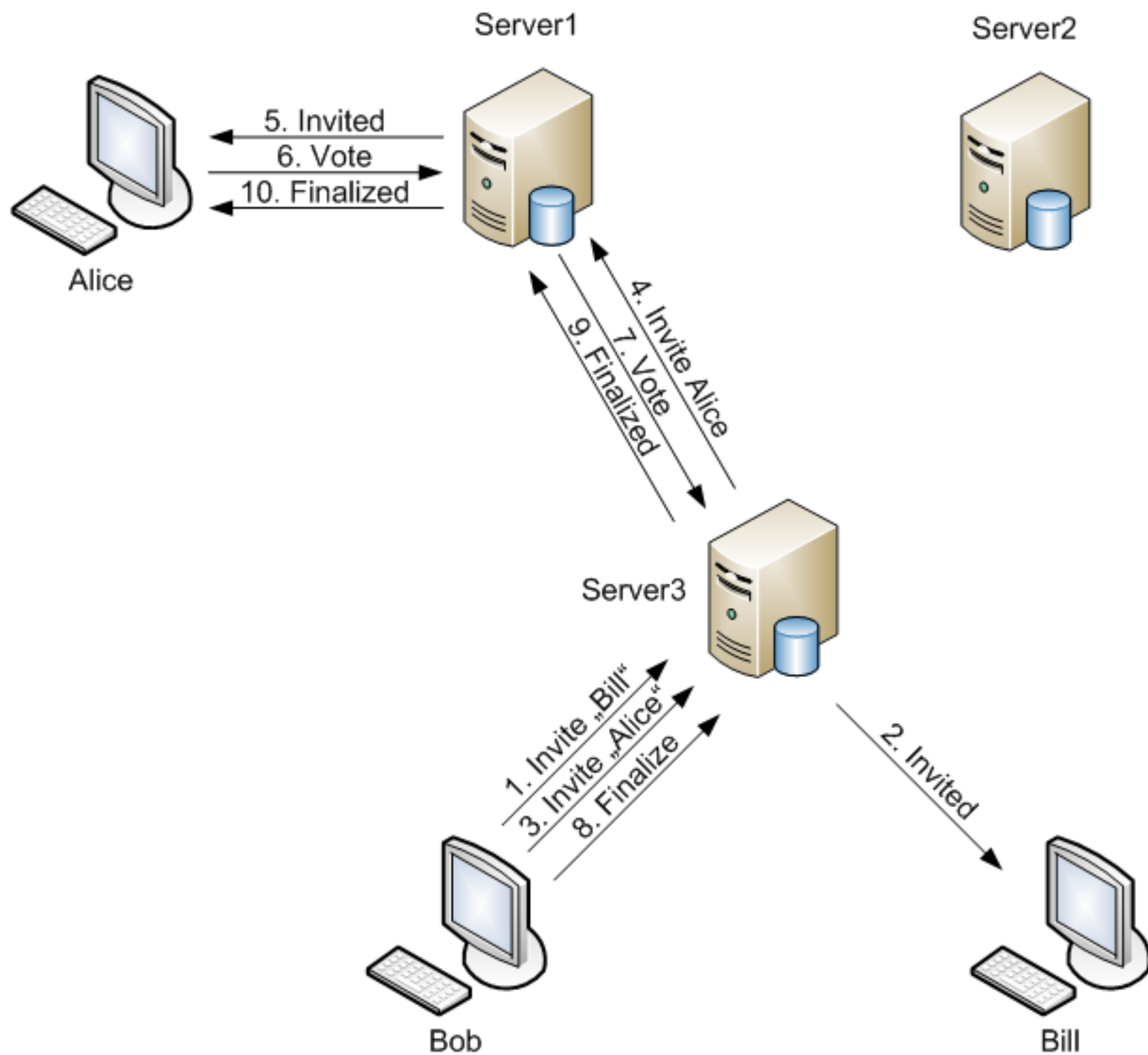

Figure 1: An example of the communication performed to invite users to an event

As already stated, the figures omit some details your final solution needs to consider. These are described in the following sections.

Server

Arguments

The Server application should expect the following arguments:

- `bindingName`: the name this server shall use to bind its remote reference in the RMI registry.
- `initRegistry`: a boolean value, i.e. either *true* or *false*, indicating whether this server is responsible for creating the RMI registry or not.
- `serverNames`: a list of names, separated by space characters, indicating the name of the other servers' remote references.

If any argument is invalid or missing print a usage message and exit.

Implementation Details

The distributed system we are going to build will contain a number of servers handling client and other servers' requests. Each server knows every other server in the network. Concerning RMI, this means that each server must be able to retrieve all remote object references from the `java.rmi.registry.Registry` service. This service can be used to reduce coupling between clients (looking up) and servers (binding): the real location of the server object becomes transparent. In our case, each server will use the registry both for looking up and binding.

One of the first things a server needs to do is to connect to the `Registry`. There is exactly one server in the network that needs to set up the RMI registry (i.e., the `init` argument is *true*). This can be achieved by calling the `LocateRegistry.createRegistry(int port)` method which creates and exports a `Registry` instance on localhost. A properties file (named `registry.properties`) should be read in from the classpath (see the **hint section** for details) to get the port the `Registry` should accept requests on. The properties file is provided and can be downloaded **here**. It also contains the host the `Registry` is bound to. This information is vital to the other servers that need to connect to the `Registry` using the `LocateRegistry.getRegistry(String host,int port)` method.

After obtaining a reference to the `Registry`, this service can be used to bind an RMI remote interface (using the `Registry.bind(String name, Remote obj)` method). Remote Interfaces are common Java Interfaces extending the `java.rmi.Remote` interface. Methods defined in such an interface may be invoked from different processes or hosts. In our case, methods may be invoked by clients as well as by other servers. Since both need a completely different and independent set of remote methods, do the following: The remote object you bind to the registry should contain exactly two methods, one for retrieving a remote callback for servers and one for retrieving a remote callback for clients. Each of these callback objects again implements a remote interface, but the methods defined there are now designated to be either used by a server or a client. This way, you only have to bind a single object to the registry. We will talk about a more important advantage of this approach in a moment.

Binding the remote object is only half the job, of course. The server also needs to lookup the references of the other servers' remote objects using the `Registry.lookup(String name)` method. The names of the other server's remote references were passed on startup (see the `serverNames` argument for details). Since the servers start one after another, this means that in the moment of lookup, a particular server might not have bound its remote reference to the `Registry` yet. Therefore, you need do the lookup in a separate thread, recurrently retry the lookup every 500 milliseconds until the server could be found. The method will return an instance of `Remote`, which you need to cast to the server's main remote interface first. You can then use this instance to retrieve the RMI callback designated for servers.

You are then ready to serve requests. Requests are handled by one of the three types of remote objects (this also means you have to create three different remote interfaces): One implements the methods that may be called by clients, one that is designated to be used by servers, and finally the remote object that was also bound to the `Registry`, containing two methods that return instances of the two remote objects mentioned before. Concerning the method that is used by servers, you can always return the same instance of the respective remote object. However, since some of the interactive commands a client can invoke require some kind of session management, always return a fresh new instance of the remote object that serves client requests. This way, each instance is unique to a particular user and you can easily make sure a user fulfills the prerequisite of the respective commands, e.g., being logged in or being the author of the respective event.

To make your object remotely available you have to export it. This can either be accomplished by extending `java.rmi.server.UnicastRemoteObject` or by directly exporting it using the static method `UnicastRemoteObject.exportObject(Remote obj, int port)`. In the latter case, use 0 as port: This

way, an available port will be selected automatically.

Distributed Transaction using 2-Phase Commit

Every time a user logs in, he/she always uses the same server to access the event scheduling system. That is, each user is managed by exactly one server. The username serves as a globally unique identification in our distributed system. Therefore, to avoid name conflicts when a new user registers, you have to start a distributed transaction.

Just like a regular (local, single database) transaction, a distributed (or federated) transaction also needs to ensure the ACID properties (Atomicity, Consistency, Isolation, Durability). To achieve this, distributed transactions make use of the *2-phase commit protocol*. A sample definition of 2-phase commit is given in **[1]**: *"An atomic commitment protocol which ensures that a transaction is terminated the same way at every site where it executes. The name comes from the fact that two rounds of messages are exchanged during this process"*. For the interested reader, more information can be found on the Web, for instance in **this article** from IBM developerWorks.

The 2-phase commit, applied to our scenario, involves the following two phases: In the first phase, the server asks the other servers whether a user with this username exists or is about to be created by another distributed transaction. The second phase depends on the outcome of the first phase: If the name is not available, registration is not successful and the server has to inform the other servers about this *rollback*. Otherwise, the sever can register the new user and tell the other servers about it (*commit*). This way, each server knows the machine that manages a particular user. This information can be useful for requests the newly registered user is involved in.

The server also needs to maintain all events that were created by the users it manages. Just like a user, an event is globally identified by a name (specified by the creating user). You need to use the same approach mentioned above (2-phase commit) to guarantee an event name is actually available.

Important Points to Consider

For further details on the application logic, please refer to **Client's interactive commands**.

Make sure to synchronize the data structures you use to manage users and events. Even though RMI hides many concurrency issues from the developer, it cannot help you at this point. You may consult the **Java Concurrency Tutorial** to solve this problem.

The data does not need to be persisted after shutting down the server. You can assume that all servers are up before the first user registers, that is, you do not need to implement a protocol that synchronizes the data among distributed servers. You may also assume that the other servers always remain accessible throughout the application's runtime.

For shutting down servers, have a look into the **hint section**. The server has to shut down after pressing the Enter key.

Part B - Client

Arguments

The client application should expect the following argument:

- `serverName`: the name of the remote reference in the RMI registry of the server that shall be responsible for this client.

If the argument is missing print a usage message and exit.

Implementation Details

At startup, the client reads out the `registry.properties` file to obtain the information where the RMI registry is located. The client is then able to retrieve the remote reference of the server using the `serverName` argument provided on startup. Next you should export the client's remote object so that the server can notify the user about certain events. The classes and methods you will need for all these steps have already been explained above: `LocateRegistry.getRegistry(String host,int port)`, `Registry.lookup(String name)` and `UnicastRemoteObject.exportObject(Remote obj, int port)`. Note that a client, in contrast to the servers, must not bind any objects to the `Registry`.

After these steps, the user can already type in commands. Your client needs to check whether all required arguements were provided (print a usage message otherwise), but it is the server's task to check whether a specified user or event exists; that is, your server should be able to deal with wrong inputs.

Some of the following commands require a successfully logged in user. Your server implementation needs to throw an exception and pass it back to the client if this is not the case.

Interactive commands

- `!register <username> <password>`

  This command creates a new user account on the server. The username must be globally unique, i.e., there must not exist another user with the same username on this or any other server in the system. To ensure this, the server might need to use the aforementioned 2 phase commit protocol. This command therefore has two different results: Either the registration is successful and the user can log in in the next step, or the user needs to register with another username. Inform the user about the outcome of the operation.

  E.g.:
  ```
  >: !register Bob 12345
  Username already registered.
  >: !register Alice 12345
  Successfully registered.
  ```

- `!login <username> <password>`

  The credentials provided during registration can be used to log in. The server needs to check the provided credentials and inform the user about the outcome of the operation; any further communication with other servers in the system is not required. Together with the credentials, the client should also send its remote object as a callback object for the server. Without the callback object, the server has no possibility to notify the client about event invitations or event finalizations. A successful login is required for any of the following commands (the only exception from this is `!exit`).

  E.g.:
  ```
  >: !login Alice 54321
  Wrong username or password.
  >: !login Alice 12345
  ```

```
Successfully logged in.
```

- `!create <name> <location> <duration in minutes>`

  This command is used to create a new event with the specified name, location and estimated duration. Note that the name of the event must be globally unique. To ensure this, the server should use the same approach it already uses to fulfill the `!register` command (again, other servers can use the information from this step for further user requests). This also means that a success of this operation is not guaranteed (therefore, inform the user about the outcome of the operation). When storing the event, the server should also append information about the user that created this event. Creating an event requires a logged in user.

  E.g.:
  ```
  >: !create our_meeting library 90
  Event created successfully.
  ```

- `!addDate <name of event> <date: dd.MM.YYYY/HH:mm>`

  Creating an event is not enough. Using this command, the author of the event can add a single possible date. To add more than one option (which makes much sense since this is what all the voting is about) the user has to call this method several times. Please stick to the stated date pattern and consult the **Tricky Parts** section for how to convert a String into a `java.util.Date` object. The only user allowed to add dates is the author of the event itself (i.e., the user that has created it before). Therefore, throw an exception if the user is not the author of such an event. Adding dates is possible until the event is finalized (that is, you can still add dates even though other users might already have voted).

  E.g.:
  ```
  >: !addDate our_meeting 25.11.2010/18:00
  Date option added.
  ```

- `!invite <name of event> <username>`

  After creating an event (and probably adding some date options), the author can invite other users to join the event and vote on the possible date options. Again, the only user allowed to invite other users is the author of the event itself. The server should keep track of the users that were invited to a particular event. It may also need to forward the invitation to a second server responsible for the invited user. A client's remote interface should contain a method to be notified about event invitations. The invitation message the user receives should be printed to the console and should at least contain the event's unique name. However, you do not have to inform the author about the outcome of the operation.

  E.g.:
  ```
  >: !invite our_meeting Bob
  ```

- `!get <name of event>`

  This command retrieves current information about the specified event and prints it to the user's console. It may be called by any user logged in (invitation is not required). The retrieved data does not need to contain information about every user's vote, but should at least specify all possible date options or - if the event has already been finalized - the fixed date of the event. Note that this command might require some communication between different servers, for the server that manages the requesting user and the server that manages the specified event may not be the same. In case the event does not exist, tell the user about

it.

E.g.:
```
>: !get our_meeting
Event: our_meeting
Location: library
Duration: 90 min.
Options: 25.11.2010/17:00 25.11.2010/17:30 25.11.2010/18:00
```

- `!vote <name of event> <date: dd.MM.YYYY/HH:mm> ...`

  The author of the event and every user that has received an invitation for it can vote on the respective event exactly once. Voting a second time or voting on an event the user has not been invited to must be ignored by the server (that is, you do not need to inform the user about the outcome of the operation). Voting works the following way: The user studies the possible date options of the event and specifies the dates he/she can and is willing to attend. When entering the command, multiple dates can be separated by whitespaces.

  E.g.:
  ```
  >: !vote our_meeting 25.11.2010/17:30 25.11.2010/18:00
  ```

- `!finalize <name of event>`

  After some time of voting, the author may decide to fix the date of the event and finalize it. The server then calculates which date option is the most attractive, i.e., has received most votes. In case there are two or more such dates, the server automatically picks the earliest of them. After finalization, voting or adding dates is no longer possible for this particular event. Next, the server needs to notify each participating user (that is, each user that has voted) about the finalization of the event. A notification should result in a message printed to the user's console. If the event does not exist or the is not the author, pass an exception to the calling client.

  E.g.:
  ```
  >: !finalize our_meeting
  ```

- `!logout`

  This command logs out the currently logged in user. The server should remove any existing callback object it may have stored for this client. If the user is no longer logged in, the server can drop any notifications addressed to this user (i.e., you do not have to store invitations and finalization notifications until the user logs in again).

  E.g.:
  `>: !logout`
  You have been logged out.

- `!exit`

  Shuts down the application. Make sure to free all acquired resources orderly before exiting.

Lab port policy

Since each student has to start its own registry service (which requires an unused port for listening for requests), we have to make sure that each student uses its own port (this has to be adjusted in `registry.properties`).
So if you are testing your solution in the lab environment (i.e. on the lab servers) you have to obey the following rule: you may only use the port **10.000 + dslabXXX * 10** for the registry. So if your account is dslab250 you have to use the port 12500.
As you might have thought of your remote objects also require an unused port. But since we are using the registry for mediation purposes, this can be an anonymous (any available port selected by the operating system) port (see **exporting objects** for more details).

Ant template

As in Lab1 we provide a template build file (**build.xml**) in which you only have to adjust some class names. Put your source code into the subdirectory "`src`", place the `registry.properties` file into the "`src`" directory (the ant compile task then copies this file to the build directory). Put the src directory including `registry.properties` and `build.xml` into your submission.
Note that it's **absolutely required** that we are able to start your programs with the predefined commands!

Hints & Tricky Parts

- To make your object remotely available you have to **export** it. This can either be accomplished by extending `java.rmi.server.UnicastRemoteObject` or by directly exporting it using the static method `java.rmi.server.UnicastRemoteObject.exportObject(Remote obj, int port)`. Use `0` as port, so any available port is selected by the operating system.

- Before shutting down a server or client, unexport all created remote objects using the static method `UnicastRemoteObject.unexportObject(Remote obj, boolean force)` – otherwise the application may not stop.

- Since Java 5 it's not required anymore to create the stubs using the **RMI Compiler** (`rmic`). Instead java provides an automatic proxy generation facility when exporting the object.

- Take care of **parameters and return values** in your remote interfaces. In RMI all parameters and return values except for remote objects are passed per value. This means that the object is transmitted to the other side using the java serialization mechanism. So it's required that all parameter and return values are serializable, primitives or remote objects, otherwise you will experience `java.rmi.UnmarshalException`s.

- To create a **registry**, use the static method `java.rmi.registry.LocateRegistry.createRegistry(int port)`. For obtaining a reference in the client you can use the static method `java.rmi.registry.LocateRegistry.getRegistry(String hostName, int port)`. Both hostname and port have to be read from the `registry.properties` file.

- We also provide a template **properties file**. Make sure to set the port according to our policy.

- Reading in a **properties file** from the classpath (without exception handling):

```
java.io.InputStream is =
ClassLoader.getSystemResourceAsStream("registry.properties");
```

```
if (is != null) {
     java.util.Properties props = new java.util.Properties();
     try {
          props.load(is);
          String registryHost = props.getProperty("registry.host");
          ...
     } finally {
          is.close();
     }
} else {
     System.err.println("Properties file not found!");
}
```

- To parse a String and make it a `java.util.Date`, you can use the `java.text.SimpleDateFormat` class:

```
String input = ...
java.text.SimpleDateFormat formatter = new
java.text.SimpleDateFormat("dd.MM.yyyy/HH:mm");
java.util.Date date = formatter.parse(input);
```

Further Reading Suggestions

- **APIs:**
  - RMI: **Remote API**, **UnicastRemoteObject API**, **Registry API**, **LocateRegistry API**
  - Properties: **Properties API**
  - IO: **IO Package API**
- **Tutorials**
  - **JavaInsel RMI Tutorial**: German introduction into RMI programming.
- **Literature**
  - [1] M. Tamer Özsu. Distributed Database Systems. **http://softbase.uwaterloo.ca/~ddbms/publications/ozsu/EIC/eic.pdf**

Retrieved from https://www.infosys.tuwien.ac.at/teaching/courses/dslab/index.php?n=Lab2.Lab2
Page last modified on October 28, 2010, at 06:08 PM CET