From Verteilte Systeme UE, 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 detailed tutorial about RMI.
- Be sure to check the **Hints & Tricky Parts** section for questions!

Updates

Submission Guide (for all Labs)

Submission

- You must upload your solution using the **Teaching Tool** before the submission deadline: **29.11.2012, 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!
- Lab 2 is a group assignment, but **only one person per group needs to (and should) upload the solution**. One group member can upload your solution as often as you like - any existing submission will be **replaced** by uploading a new one. If multiple (i.e., both) group members upload a solution, **the submission with the latest timestamp** will be graded!
- Do not overburden our lab servers by implementing remotely on the server. The lab server is just for testing purposes. Further on, you have to submit your solution in the **Teaching Tool**. We cannot grade any solutions uploaded on the lab server!
- 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). It is essential that your solution works with the provided and build file - do **not** submit solutions that use Maven or other build/dependency management systems!
- You can use log4j for logging, but you must not include the library in your submission! Have a look at the provided project template in section **ant template**.
- 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.
- 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

The goal of this assignment is to extend the auction system with four additional components:
- An **analytics server** receives events from the system and computes simple statistics/analytics.
- A **billing server** which manages the bills charged by the auction provider.

- A **management client** which interacts with the analytics server and the billing server.
- A **testing component** to generate client requests for automated load testing of the system.

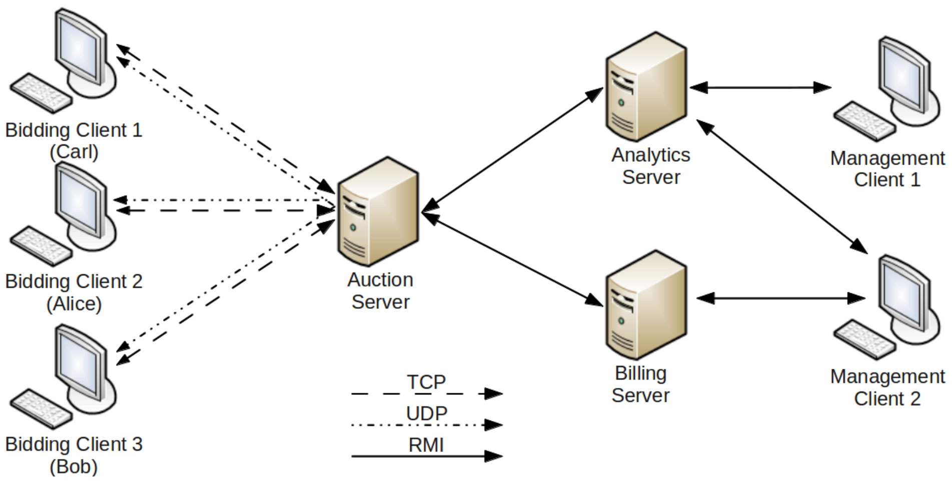

Figure 1: System Components and Interactions.

Analytics Server

The purpose of the analytics server is to monitor the execution of the auction platform and to provide simple statistics about its usage. The analytics server should receive event updates from the auction server that you have implemented in assignment 1. Moreover, the analytics server should allow one or multiple management clients to subscribe for event notifications. To that end, the analytics server provides two RMI methods:

- A **subscribe** method is invoked by the management client(s) to register for notifications. The method must allow to specify a filter (specified as a regular expression string) that determines which types of events the client is interested in (see details further below). Moreover, the subscribe method receives a callback object reference which is used to send notifications to the clients. Study the RMI callback mechanism to solve this. The method returns a unique subscription identifier string.
- A **processEvent** method is invoked by the bidding server each time a new event happens (e.g., user logged in, auction started, ...). The analytics server forwards these events to subscribed clients, and possibly generates new events (see details further below) which are also forwarded to clients with a matching subscriptions. If the server finds out that a subscription cannot be processed because a client is unavailable (e.g., connection exception), then the subscriptions is automatically removed from the analytics server.
- An **unsubscribe** method is invoked by the management client(s) to terminate an existing event subscription. The method receives the subscription identifier which has been previously received from the **subscribe** method.

Event Type Hierarchy

The processEvent method of the analytics server should receive an object of type **Event**. To transport the necessary event payload (user name, auction ID, bid price, ...), implement a simple event hierarchy which is illustrated in the figure below. The sub-types inherit from the abstract class Event, and the business logic inside the processEvent method should determine the concrete runtime type of the incoming events, in order to take appropriate action. Each event contains a unique identifier (ID), a type string, and a timestamp, and specialized event types contain additional data. For instance, the *AuctionEvent* defines a member variable *auctionID* that carries the auction identifier which this event applies to.

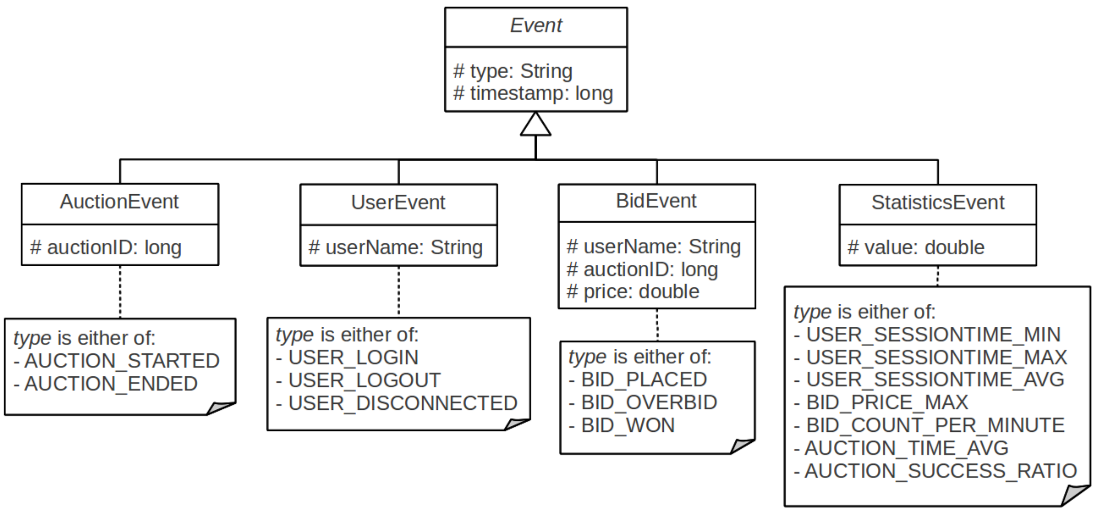

Figure 2: Event Hierarchy and Basic Event Properties.

Statistics Events

The analytics server is responsible for processing the raw events and creating the following aggregated statistics event types:

- *USER_SESSIONTIME_MIN / USER_SESSIONTIME_MAX / USER_SESSIONTIME_AVG*: minimum/maximum/average duration over all user sessions. A user session starts with a login and ends if the user logs out (or gets disconnected unexpectedly).
- *BID_PRICE_MAX*: maximum over all bid prices.
- *BID_COUNT_PER_MINUTE*: number of bids per minute, aggregated over the whole execution time of the system.
- *AUCTION_TIME_AVG*: average auction time, aggregated overall historical auctions logged during the execution of the system.
- *AUCTION_SUCCESS_RATIO*: ratio of successful auctions to total number of finished auctions. An auction is successful if at least one bid has been placed.

Note that an incoming event (e.g., type AUCTION_ENDED) may trigger the generation of multiple new events (e.g., types AUCTION_TIME_AVG, AUCTION_SUCCESS_RATIO).

Event Subscription Filter

The *subscribe* method of the statistics server should allow to set a simple subscription filter. The filter is a Java regular expression which is matched against the *type* variable of the Event class. For instance, to subscribe for all user-related and bid-related events, the filter `"(USER_.*)|(BID_.*)"` is used.

Billing Server

The billing server provides RMI methods to manage the bills of the auction system. To secure the access to this administrative service, the server is split up into a BillingServer RMI object (which basically just provides login capability) and a BillingServerSecure which provides the actual functionality.

Pricing Curve

Administrators can use the billing server to set the pricing curve in terms of fixed price and variable price steps. In a typical pricing curve, the auction fee percentage decreases with increasing auction price (price of the winning bid). For example, the price steps could look something like this:

Auction Price	Auction Fee (Fixed Part)	Auction Fee (Variable Part)
0	1.0	0.0 %
(0-100]	3.0	7.0 %
(100-200]	5.0	6.5 %
(200-500]	7.0	6.0 %
(500-1000]	10.0	5.5 %
> 1000	15.0	5.0 %

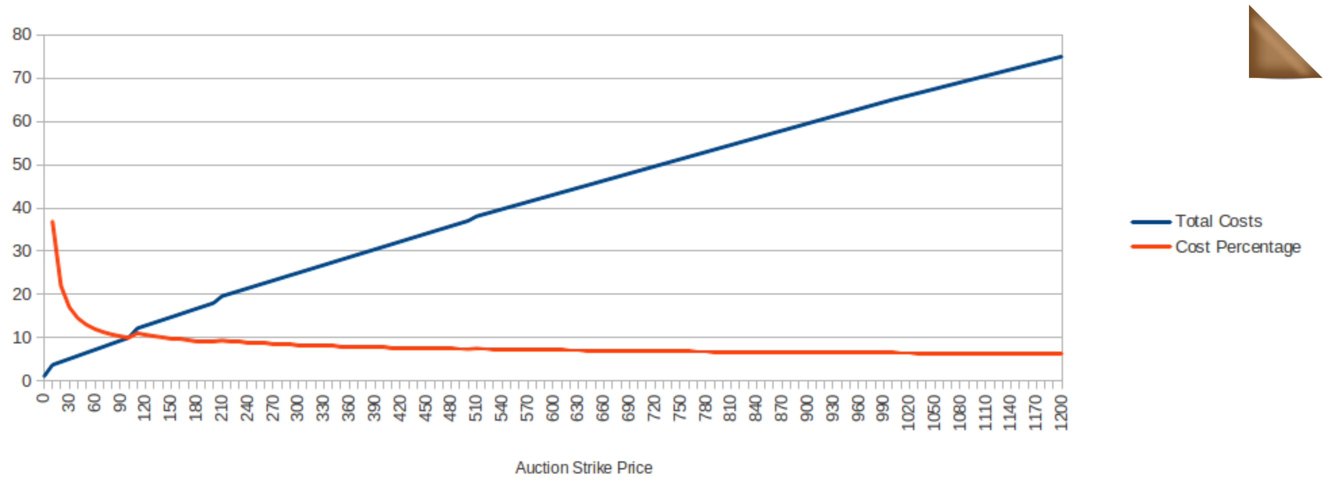

Figure 3: Pricing Curve

Billing Server RMI Methods

The *BillingServer* class provides the following RMI methods (exception declarations not included):

- `BillingServerSecure login(String username, String password)`: The access to the billing server is secured by user authentication. To keep things simple, the username/password combinations can be configured statically in a config file `user.properties`. Each line in this file contains an entry "<username> = <password>", e.g., "john = f23c5f9779a3804d586f4e73178e4ef0". Do not put plain-text passwords into the config file, i.e., store an MD5 hash of the password. Use the *java.security.MessageDigest* class to obtain the MD5 hash of a given password. If and only if the login information is correct, the management client obtains a reference to a *SecureBillingServer* remote object, which performs the actual tasks.

The *BillingServerServer* provides the following RMI methods (exception declarations not included):

- `PriceSteps getPriceSteps()`: This method returns the current configuration of price steps. Think of a suitable way to represent the list of price step configurations inside your PriceSteps class.
- `void createPriceStep(double startPrice, double endPrice, double fixedPrice, double variablePricePercent)`: This method allows to create a price step for a given price interval. Throw a RemoteException (or a subclass thereof) if any of the specified values is negative. Also throw a RemoteException (or a subclass thereof) if the provided price interval collides (overlaps) with an existing price step (in this case the user would have to delete the other price step first). To represent an infinite value for the `endPrice` parameter (e.g., in the example price step "> 1000") you can use the value 0.
- `void deletePriceStep(double startPrice, double endPrice)`: This method allows to delete a price step for the pricing curve. Throw a RemoteException (or a subclass thereof) if the specified interval does not match an existing price step interval.

Management Client

The management client communicates with the analytics server and the billing server. The client should provide all necessary commands to interact with the analytics server and the billing server.

The following commands are used to interact with the billing server (please follow the output format illustrated in the examples):

- `!login <username> <password>`: Login at the billing server.

```
> !login bob BobPWD
bob successfully logged in
```

- `!steps`: List all existing price steps

```
bob> !steps
 Min_Price Max_Price Fee_Fixed Fee_Variable
 0         0         1.0       0.0%
 0         100       3.0       7.0%
 100       200       5.0       6.5%
 200       500       7.0       6.0%
 500       1000      10.0      5.5%
```

Note: the table columns in the output do not have to be perfectly aligned.

- `!addStep <startPrice> <endPrice> <fixedPrice> <variablePricePercent>`: Add a new price step.

```
bob> !addStep 1000 0 15 5
Step [1000 INFINITY] successfully added
bob> !steps
 Min_Price Max_Price Fee_Fixed Fee_Variable
 0         0         1.0       0.0%
 0         100       3.0       7.0%
 100       200       5.0       6.5%
 200       500       7.0       6.0%
 500       1000      10.0      5.5%
 1000      INFINITY  15.0      5.0%
```

- `!removeStep <startPrice> <endPrice>`: Remove an existing price step.

```
bob> !removeStep 0 100
Price step [0 100] successfully removed
bob> !removeStep 111 222
ERROR: Price step [111 222] does not exist
```

- `!bill <userName>`: This command shows the bill for a certain user name. Print the list of finished auctions (plus auction fees) that have been created by the specified user. To determine the fees of the bill, apply the current price steps configuration to the prices of each of the user's auctions.

```
bob> !bill bob
 auction_ID strike_price fee_fixed fee_variable fee_total
 1          0            1         0            1
 17         700          10        38.5         48.5
 19         120          5         7.8          12.8
```

- `!logout`: Set the client into "logged out" state and discard the local copy of the *BillingServiceSecure* remote object. After this command, users have to use the "!login" command again in order to interact with the billing server.

```
bob> !logout
bob successfully logged out
> !bill bob
ERROR: You are currently not logged in.
```

Provide reasonable output with all relevant information for each of the commands (as illustrated in the examples above). Also print an informative error message if an exception is received from the server (Error log messages should start with the string "`ERROR:`").

The following user commands are provided by the management client to communicate with the analytics server:

- `!subscribe <filterRegex>`: subscribe for events with a specified subscription filter (regular expression). A user can add multiple subscriptions.

```
bob> !subscribe '(USER_.*)|(BID_.*)'
Created subscription with ID 17 for events using filter
'(USER_.*)|(BID_.*)'
```

- `!unsubscribe <subscriptionID>`: terminate an existing subscription with a specific identifier (`subscriptionID`).

```
bob> !unsubscribe 17
subscription 17 terminated
```

Similar to the analytics server, the management client (i.e., the RMI callback object) should implement a **processEvent** RMI method. This method is invoked by the analytics server each time an event is generated that matches a subscription by this client. To display incoming events to the user, simply print the event time, event type and all additional event properties to the command line. You should support two modes of printing, namely *automatic* and *on-demand*:

- `!auto`: This command enables the automatic printing of events. Whenever an event is received by the clients, its details are immediately printed to the command line.
- `!hide`: This command disables the automatic printing of events. Incoming events are temporarily buffered and can later be printed using the "`!print`" command. This should be the default mode when the client is

  started.
- `!print`: Print all events that are currently in the buffer and have not been printed before (an excerpt o possible example trace is printed below).

```
bob> !print
USER_LOGIN: 31.10.2012 20:00:01 CET - user alice logged in
USER_LOGIN: 31.10.2012 20:01:03 CET - user john logged in
BID_PLACED: 31.10.2012 20:01:50 CET - user alice placed bid 3100.0 on auction 32
BID_PRICE_MAX: 31.10.2012 20:01:50 CET - maximum bid price seen so far is 3100.0
BID_COUNT_PER_MINUTE: 31.10.2012 20:01:50 CET - current bids per minute is 0.563
USER_LOGOUT: 31.10.2012 20:03:11 CET - user john logged out
USER_SESSION_TIME_MIN: 31.10.2012 20:03:11 CET - minimum session time is 62 seconds
USER_SESSION_TIME_MAX: 31.10.2012 20:03:11 CET - average session time is 324 seconds
USER_SESSION_TIME_AVG: 31.10.2012 20:03:11 CET - maximum session time is 778 seconds
```

Note: The value of BID_COUNT_PER_MINUTE depends on the previous events which are not visible in this trace. For the example, an arbitrary value of 0.563 has been chosen for illustration. (Arbitrary values have also been chosen for the aggregated session durations.)

In both variants, it must be ensured that each distinct event is presented to the user only *once*, even if it matches multiple subscriptions. It is up to you whether you solve this requirement server-side (never send out duplicate events to any client) or client-side (receive duplicate events, but print each event only once on the command line). If you need to temporarily store a list of already published or already printed events, make sure that the events are eventually freed and that your program does not run out of memory over time.

Load Testing Component

In practice, online auction systems are highly concurrent and should provide robustness and scalability, even for a large number of clients. Concurrency issues are hard to detect under normal operation, but generating artificial load on the system may help to detect problems and inconsistencies. Hence, you should create a load testing component to test the system's scalability and reliability. The following test parameters should be configurable in the properties file `loadtest.properties` (see template):

- *clients*: Number of concurrent bidding clients
- *auctionsPerMin*: Number of started auctions per client per minute
- *auctionDuration*: Duration of the auctions in seconds
- *updateIntervalSec*: Number of seconds that have to pass before the clients repeatedly update the current list of active auctions
- *bidsPerMin*: Number of bids placed on (random) auctions per client per minute

To place a bid, the test client selects a random auction from the list of active auctions (if any). When your test clients place bids, you need to ensure that each bid is higher than the previous bids. To achieve this, simply set the bid price that corresponds to the number of seconds that have passed since the auction was started, with decimal values (e.g., 10,342 for ten seconds and 342 milliseconds).

Moreover, the test environment should instantiate a management client with an event subscription on any event type (filter ".*"). During the tests, the management client should be in "auto" mode, i.e., print all the incoming events automatically to the command line.

Scaling up the number of clients would block too many port numbers for the UDP notifications. Hence, for assignment 2 you should disable the UDP notifications (part of assignment 1), i.e., do not open a UDP socket on the bidding clients, and do not send UDP notifications from the auction server to the clients.

Play around with different test settings and try to roughly determine the limits of your machine. By limits we mean the configuration values for which the system is still stable, but becomes unstable (e.g., runs out of memory after some time) if any of these values are further increased (or decreased). Monitor the memory usage with tools like "top" (Unix) or the process manager under Windows, and let the test program execute for a sufficiently long time (around 5-10 minutes). Obviously, the load test can also help you identify issues in your implementation (e.g., deadlocks, memory leaks, ...).

In your submission, include a file **evaluation.txt** in which you provide your machine characteristics (operating system version, number and clock frequency of CPUs, RAM capacity) and the key findings of your evaluation (at least the limit values for all configuration parameters). **Note:** Please try to **avoid** running your tests on our **lab servers**; however, if it is absolutely necessary that you run the tests on the servers, **make sure that your tests terminate after a short time** (say, 5 minutes)! The servers will be monitored and long-running resource-greedy processes may be forcibly killed without prior notice! If you decide to use the servers, only execute the final run of your evalution on the servers, but please develop your code somewhere else (e.g., on the PCs in the DSLab) to avoid blocking the server resources for other students. Also, we advise you not to use (and rely on) the Lab servers in the last days before the deadline.

Implementation Details and Hints

Implementation Details

RMI uses the `java.rmi.registry.Registry` service to enable application to retrieve remote objects. 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, the servers (analytics server and the billing server) will use the registry for binding and the client will look up the remote objects.

One of the first things the servers need to do is to connect to the `Registry`, therefore the RMI registry needs to be set up before its first usage. 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 from the classpath (see the **hint section** for details) to get the port the `Registry` should accept requests on. The properties file is provided in the template. It also contains the host the `Registry` is bound to. This information is vital to the client application that needs to connect to the `Registry` using the `LocateRegistry.getRegistry(String host,int port)` method. Note that a client, in contrast to the management component, need not bind any objects to the `Registry`.

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, we need to bind two objects to the registry: an instance of BillingServer and an instance of AnalyticsServer. If you prefer to bind only a single object to the registry, you may choose to implement an additional "facade" object which returns instances of the two server classes (however, this is not required).

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.

For managing the configurable data (e.g., user credentials) you will have to read a .properties file. The `user.properties` file must be located in the server's classpath. The properties file is provided and can be downloaded **here**.

Important Points to Consider

Make sure to synchronize the data structures you use to manage price steps, events, etc. 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.

No data needs to be persisted after shutting down the server. You can assume that the management component is up first at all, so the clients won't have problems by looking up the management components.

Group Teamwork

Lab 2 is a group assignment and the group is responsible for the solution as a whole. You may form a group with someone you know, but you should also be able to work with some unknown team colleagues (developing teamwork skills is an explicit goal of the study curricula at TU Wien, and also a requirement you will often encounter in industry jobs). We recommend that you solve the tasks together in pair programming sessions. Should you decide to split up the work internally (e.g., analytics server and management client for one student, billing server and testing component for the other student), you need to ensure that each group member understands and is able to explain *all the code* in detail! In case you split up the work, add a file **teamwork.txt** to your submission which roughly explains who was working on which part.

Lab port policy

Since it is not possible to open ServerSockets or DatagramSockets on ports where other services are already listening for requests and each student has to start its own registry service (which requires an unused port for listening for requests), we have to make sure each student uses its own port range. That means that if you are testing your solution in the lab environment (i.e., on the lab server) you have to obey to the following rule: you may only use ports between 10.000 + dslabXXX * 10 and 10.000 + (dslabXXX + 1) * 10 - 1. So if your account is dslab250 you may use the ports between 12500 and 12509 inclusive. Note that you can use the same port number for TCP and UDP services (e.g., it is possible to use TCP port 12500 and UDP port 12500 at the same time). 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,

ports, and RMI names. Further on we give you the opportunity to use log4j. At the interview (Abgabegesp our tutors will use the library, that is included in the (**template project**). Do not use any other third-party libraries. Put your source code into the subdirectory "`src`", the files `registry.properties`, `loadtest.properties` and `user.properties` are already in 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.

- You should not use a Security Manager for Lab2. So you do not need a policy files.

- 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.UnmarshalExceptionS`.

- 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 registry.properties file in the template. 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!");
}
```

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.

Retrieved from https://www.infosys.tuwien.ac.at/teaching/courses/dslab/index.php?n=Lab2.Lab2
Page last modified on October 31, 2012, at 09:14 PM CET