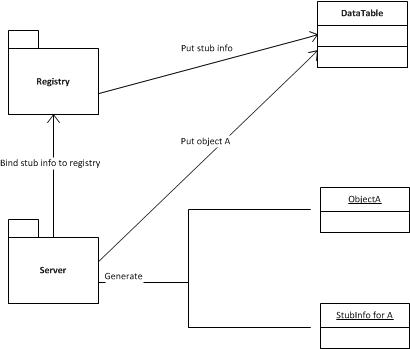
RMI implementation

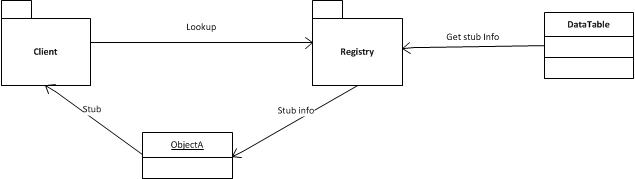
# RMI Framework

## Bind to registry



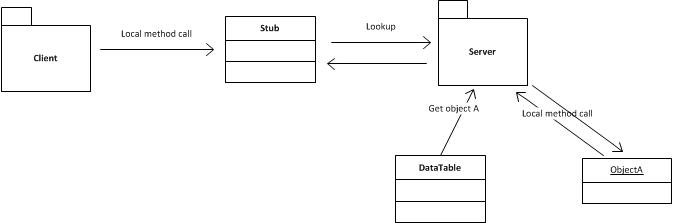
After the server is created, it will initialize the implementation object (Object A), put it in a hashtable, and export its stub info object that contains information about this server, and reference of the implementation object. Then the server binds this stub info object to the registry with a name. The registry put the stub info in a hashtable with given name.

## Get stub



The client will look up the registry with a name. The registry gets the stub info with given name and returns it to client. Client receives stub info object, retrieves its information and uses dynamic proxy to create stub.

## Method call



When the client calls a method on the stub, the stub will send a request (including method name, parameters and reference of implementation object) to the server. The server receives the request, and let one thread to handle it. The thread gets the implementation object with reference, uses java reflection to call implementation object’s method and put the value in replying message.

## Other

We use thread pool as a container for those threads that processing request in the server or registry. And there is message queue to store received message. In the client side, multiple threads can send and wait for result at the same time. All the thread pool, socket connection, hashtable, server and registry are using singleton pattern.

# RMI Design Decision

## Dynamic Proxy

In our RMI implementation, we don’t transmit class files for stub. Instead, we use dynamic proxy. Dynamic proxy is an advanced java feature, which can dynamic generate a class that will invoke proxy method. By using dynamic proxy, we can simply generate a stub class and object on the client at runtime.

Trade-off: every time the client looks up for a stub, the JVM will generate a stub class, which might not be efficient.

## Communication

The communication between different parts of the RMI (registry, server, client), is using serialization and deserialization message object. In the server and registry, when receiving a request, it gives the request to a worker thread in the thread pool to handle. However in the client side, the client has to wait for the result after sending request. At the beginning, we design that only one thread can send message and wait for the result at one time. This thread will occupy the connection resource until it receives the feedback from registry or server. Although this works well, but it is highly inefficient. So we further improve the connection in the client so that multiple threads can send and wait at one time. To achieve this, we add messageId and responseId in the message. After sending message to the server or registry, the client will put the message in a hashtable with messageId as key and wait on this message. When registry or server replying client, they will put client-message’s messageId in the responseId of replying message. When the client receives the message from registry or server, they get the original message by look up in the hashtable with responseId as key. The client clone some new value from the received message to the original message, and notify threads on original message.

Trade-off: None;

## Stub Info

In our RMI implementation, we store stub Info in the registry. At the beginning, we export stub on the server and bind the stub to the registry. However, this approach turns out imperfect, because the registry needs all the interfaces of stub to deserialize. So, we change to store all the stub information (like server host, port, server-side object reference, interfaces name) on the registry. When the client look up stub, the registry will return the relevant stub information, then the client will generate the stub according to the stub information.

Trade-off: None.

## Singleton Multithread

In our RMI implementation, we use a pattern similar to singleton multithread in the sever-side method call. When the sever calls a local method of object, we don’t add a mutex lock on this object. So, different threads can call methods of one object at the same time. Because the variable in the method is thread-safe, so the result is still reliable. This approach will greatly improve the efficiency without affecting the result. Most importantly, without using mutex lock, we can in some degree avoid deadlock.

Trade-off: If the method involves some change to instance or class properties, we need to add some synchronization block in the method.

