Java's RMI model

Runtimes for concurrency and distribution

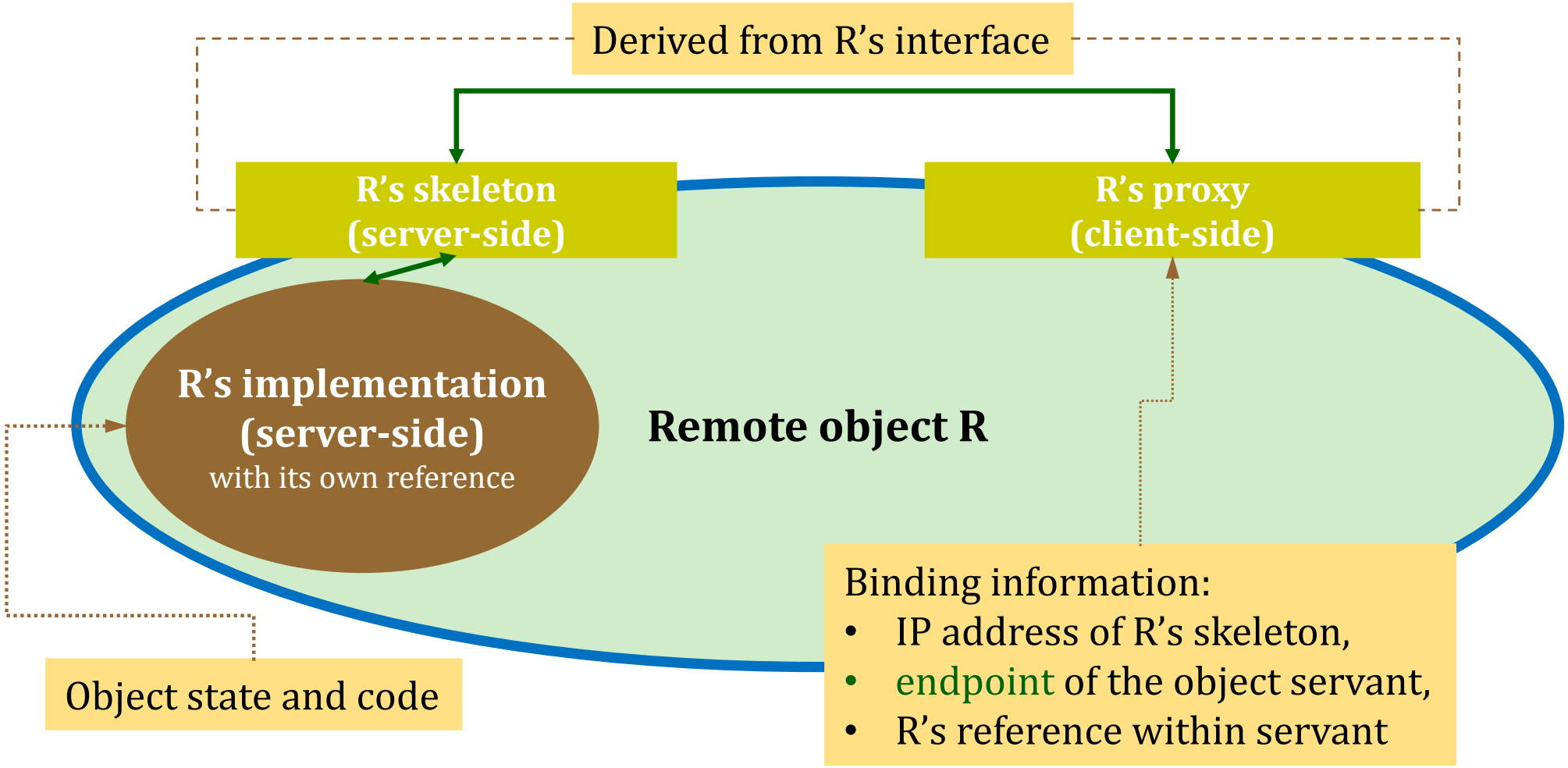
Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

Model architecture – 1

- An object-oriented rejuvenation of RPC
  - Client-server-over-TCP architecture
  - At-most-once request-reply semantics
- Nicest traits
  - Object as the most natural *unit of distributed system*
  - Its interface as the most natural "*distributable*" part
  - The server object never moves
    - It resides where its state (implementation) resides
    - It lives as long as the heap where it resides stays alive
- Principal defect
  - Full transparency **not** warranted

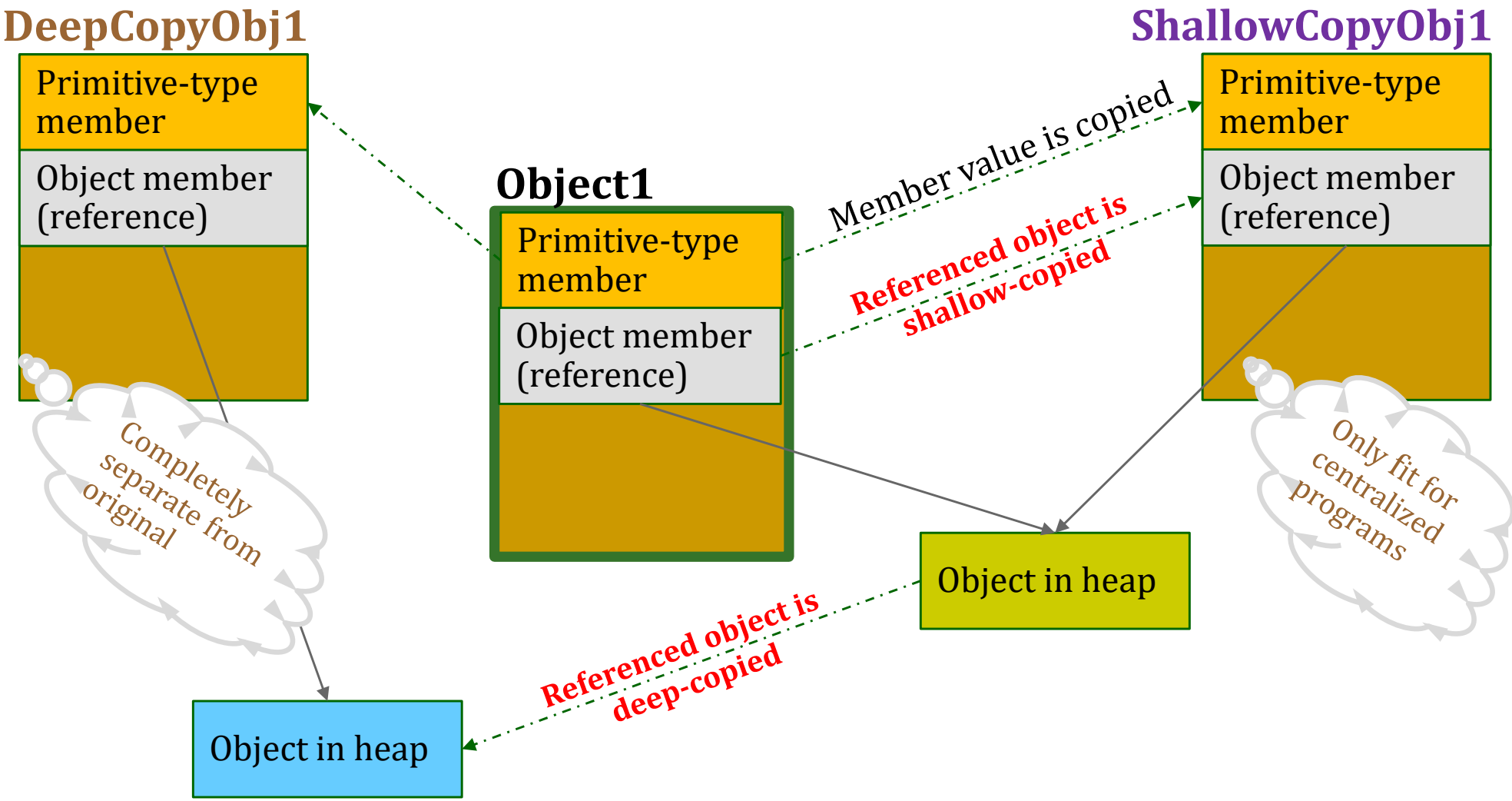
Model architecture – 2



Transparency holes

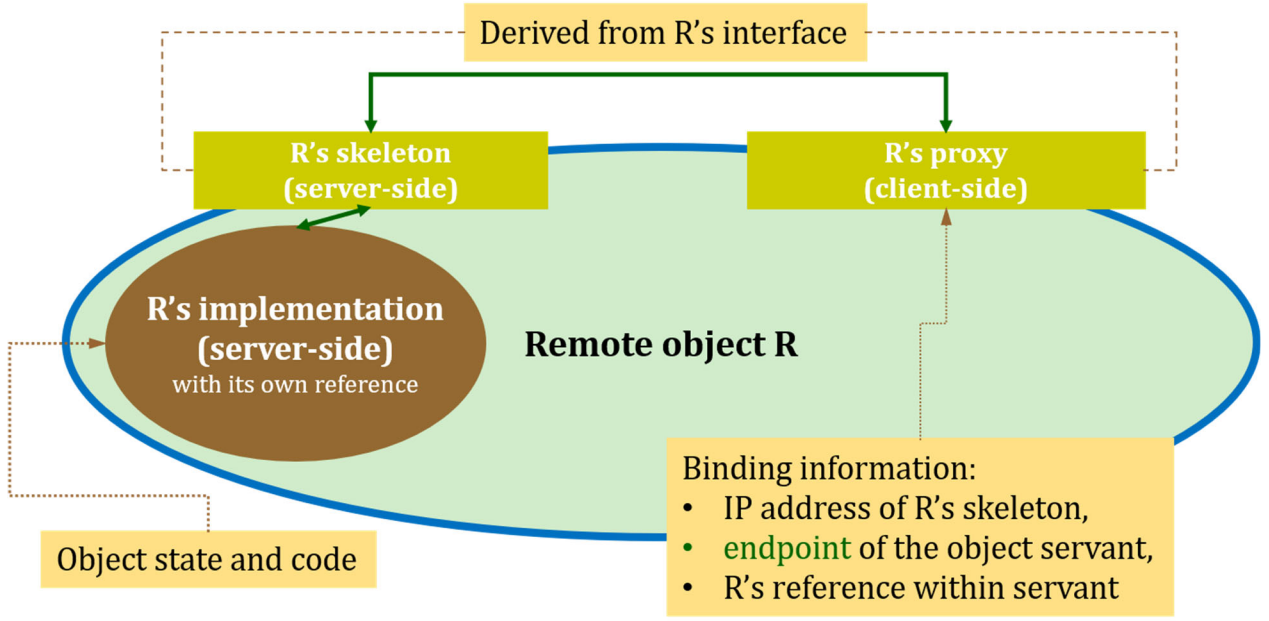
- Remote object **not** equal to local object
  - Remote object cannot be cloned as fully as local one
    - Client-side proxy **not** involved in server-side cloning
  - Binding to clone requires new proxy
- Access control to remote object is server-side **only**
  - Does not involve proxies
    - Proxy sharing at client side may or may not serialize
  - Data race if remote method implementation is not **`synchronized`**
- Call parameters treated **differently** in remote calls
  - Parameter type must allow marshalling (**`serializable`**)
    - Impossible for node-local types (threads, files, sockets, …)
    - Unwanted for those intrinsically insecure (`FileInputStream`)
  - Local objects in remote calls passed by *deep* copy
  - Remote objects in remote calls passed by *reference*

Shallow copy vs deep copy

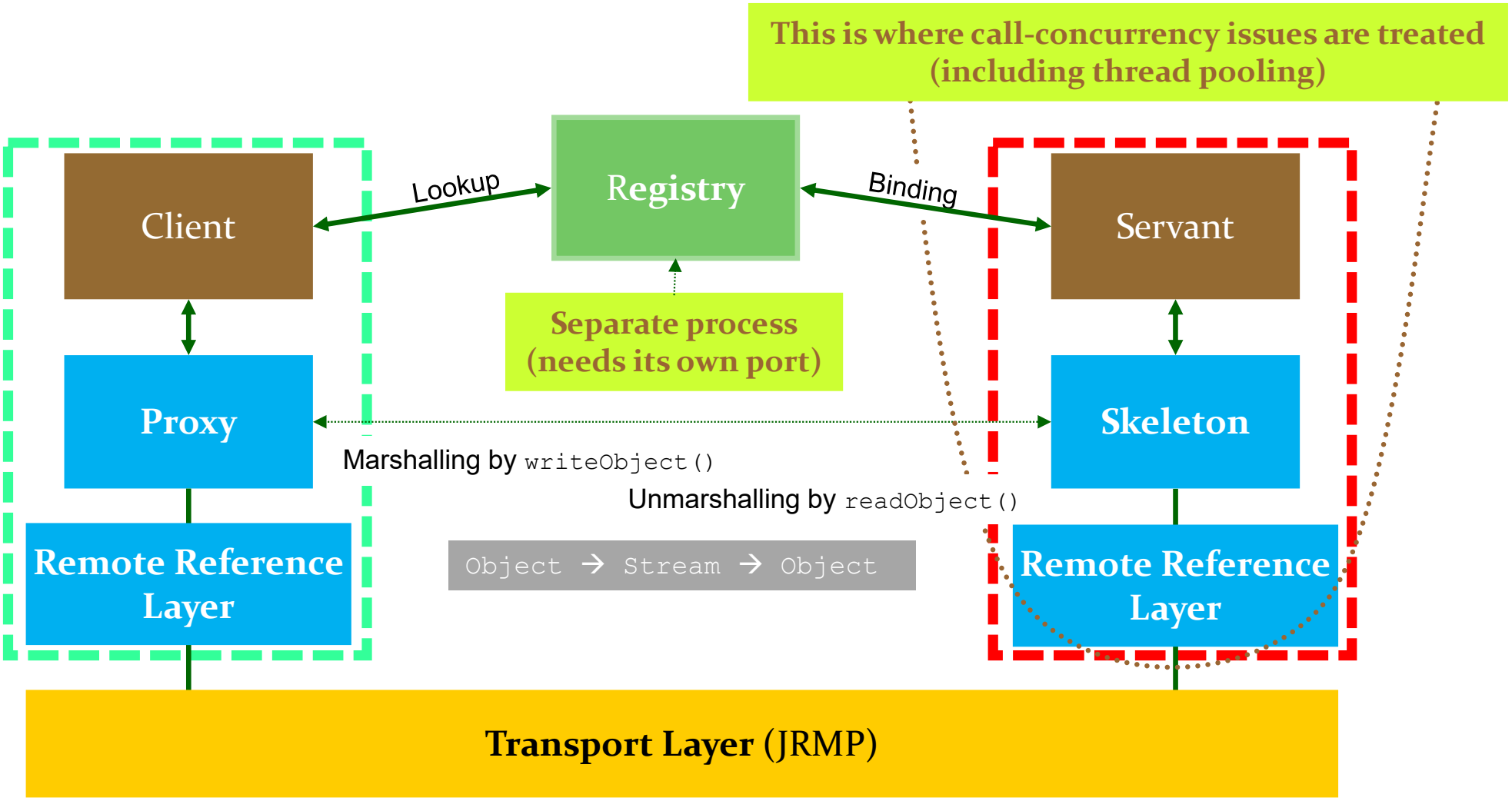


Model architecture – 3

- **R**'s proxy turns remote invocation to **R** into a TCP-level message for care by the **Remote Reference Layer** (Java's middleware)
- Call destination specified as an augmented TCP **endpoint**
  - R's node IP address, port number, R's ID at local RRL, protocol



Model architecture – 4

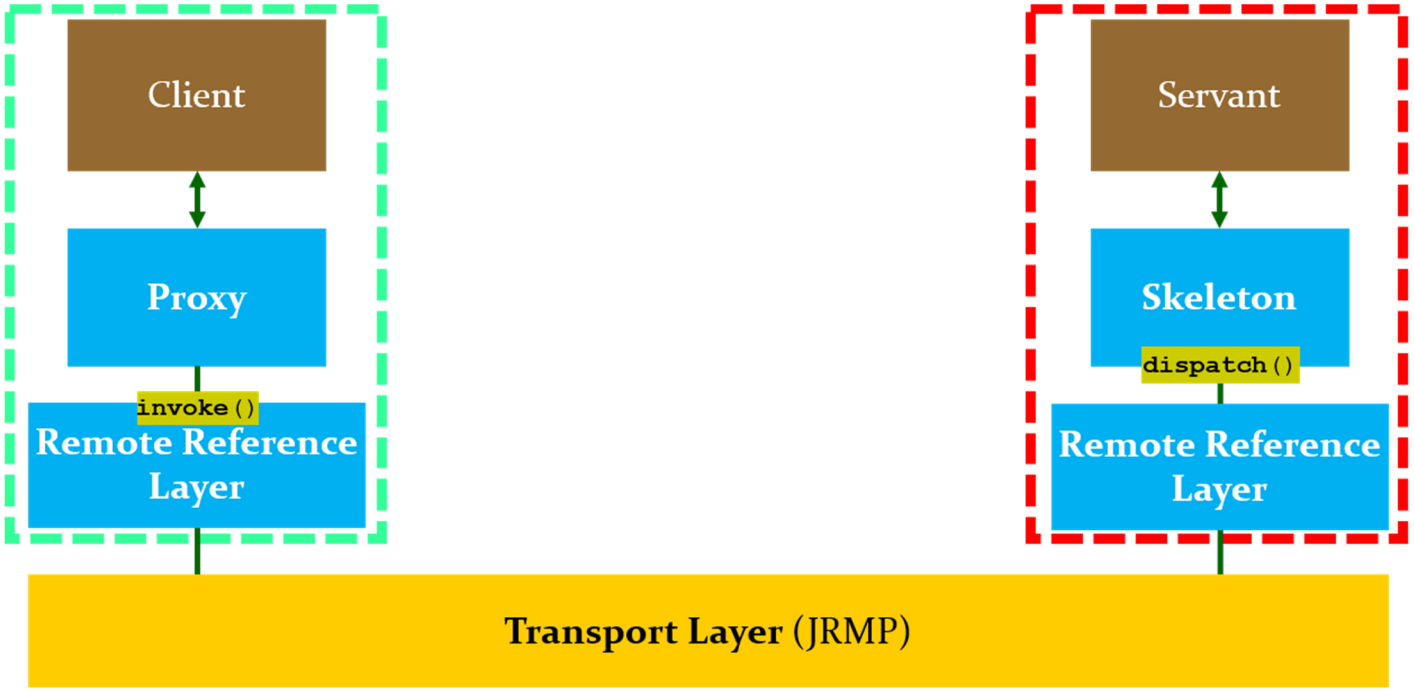


Model architecture – 5

- Serializable objects transferred as "*recipe-and-ingredients*"
  - Portability within JVM allows reproducing wanted object at destination
  - Original `.class` suffices for **by-value** parameters
    - Can be done, such files are fully local to the caller
  - **By-reference** mode needed for objects that cannot be reproduced outside of local node
- The proxy itself is serializable
  - Can be transferred the same as normal parameters
  - The very principle used for **binding** client to server

Model architecture – 6

- **Proxy** is actual target of client's call
  - Proxy reifies call and forwards it to client-side RRL using `invoke()` method of `java.rmi.server.RemoteRef`
- **Skeleton** receives `dispatch()` call from server-side RRL with reified call as parameter
  - Skeleton unmarshalls reified call and makes invocation on client's behalf



What happens under the hood – 1

1. Servant creates instance of remote object, which must extend `UnicastRemoteObject`
   - Constructor for `UnicastRemoteObject` enables remote object (server) to receive incoming RMI calls
   - TCP socket bound to arbitrary port is created
   - Middleware thread is created to listen on that socket
2. **Servant registers remote object with RMI registry, whose entry contains the corresponding proxy**
   - `RMIRegistry` holds proxies and hands them to clients on request
   - Proxy contains info to "call back" to the servant on client call
3. **Client obtains proxy by calling RMI registry**
   - If server specified a codebase for clients to obtain proxy's `.class`, registry return will include that
   - Client can then use codebase to construct proxy in-place

What happens under the hood – 2

4. When client issues RMI, proxy creates `RemoteCall` object [**now deprecated**]
   - That object opens socket connection to servant on port specified in proxy, and sends RMI header information to it

5. Proxy calls `RemoteCall.executeCall()` to cause RMI to happen [**now deprecated**]
   - Proxy serializes call arguments into Java stream object and marshals them over the connection

6. When client connects to servant's socket, *new thread is forked* on servant's side to serve the incoming call
   - Original thread keeps listening to original socket for calls from other clients
   - The very same logic as used by the Apache Web Server …

What happens under the hood – 3

7. Servant reads RMI header information and creates `RemoteCall` object to unmarshall incoming RMI arguments [**now deprecated**]

8. Servant calls skeleton's `dispatch()` method, which calls target object method and pushes return result back to socket

9. Return value of RMI is unmarshalled at client side, and returned from proxy back to client

Concurrency control

- **RMI Spec @ 3.2 Thread Usage in RMI**
  - A method dispatched by the RMI runtime to a remote object implementation **may or may not** execute in a separate thread
  - The RMI runtime makes no guarantees with respect to mapping invocations to threads
  - Since remote method invocation on the same remote object **may execute concurrently**, a remote object implementation needs to make sure its implementation is thread-safe
  - "*It's your problem, baby*"

This needs reentrancy

- Calls from the same client are certainly sequential
  - Unless the client has shared the proxy
- Calls from parallel clients need server-side handling

Use example: servant

```
package echo;
public interface Echo extends java.rmi.Remote {
  String call (String message) throws java.rmi.RemoteException;
}
```

This goes to Registry at servant's node
(**rebind** overwrites previous, if any; **bind** disallows overwriting)

```
package echo; import java.rmi.*; import java.rmi.server.*;
public class EchoServer extends UnicastRemoteObject implements Echo {
   public EchoServer( String name ) throws RemoteException {
      try { Naming.rebind (name,this); } catch (Exception e) {
         System.out.println ("Exception in EchoServer: " + e.getMessage());
         e.printStackTrace();} }
   public String call (String message) throws RemoteException {
      System.out.println("Echo's method call invoked: [" + message + "]");
      return "From EchoServer:- Thanks for your message: [" + message + "]"; }
   public static void main (String args[]) throws Exception {
      if (System.getSecurityManager() == null)
         System.setSecurityManager ( new RMISecurityManager() );
      String url = "rmi://" + args[0] + "/Echo";
      EchoServer echo = new EchoServer (url);
      System.out.println("EchoServer ready!"); }
}
```

Use example: client

```
package echo; import java.rmi.*; import java.rmi.server.*;
public class EchoClient {
   public static void main (String args[]) {
      int i;
      if (System.getSecurityManager() == null)
         System.setSecurityManager ( new RMISecurityManager() );
      try {
         System.out.println ("EchoClient ready!");
         String url = "rmi://" + args[0] + "/Echo";
         System.out.println ("Looking up remote object " + url + " ...");
         Echo echo = (Echo) Naming.lookup (url);
         String toMsg = (String) args[1];
         for (i = 1; i<6; i++) {
            toMsg = toMsg + "-" + i;
            System.out.println ("Message " + i + " to Echo: [" + toMsg + "]");
            String fromMsg = echo.call (toMsg);
            Thread.sleep (2000);
            System.out.println ("Message from Echo: \n\t" + fromMsg + "\n"); }
      } catch (Exception e) {
         System.out.println ("Exception in EchoClient: " + e.getMessage());
         e.printStackTrace(); } }
}
```

`echo` is the proxy and has the type of the `Echo` interface!

Use example

- Prior to Java 5, applications using RMI had to be compiled in **two steps**
  - First step was classic `javac`
  - Second step, `rmic`, was to generate proxy (stub) and skeleton based on actual remote object
- Since Java 5, proxy generated on-the-fly, and skeleton is taken care of by `javac`