

# Mobile und Verteilte Datenbanken

## Java RMI

Vorlesung  
Wintersemester 2013/2014

*Sven Groppe* [groppe@ifis.uni-luebeck.de](mailto:groppe@ifis.uni-luebeck.de)



Institut für Informationssysteme

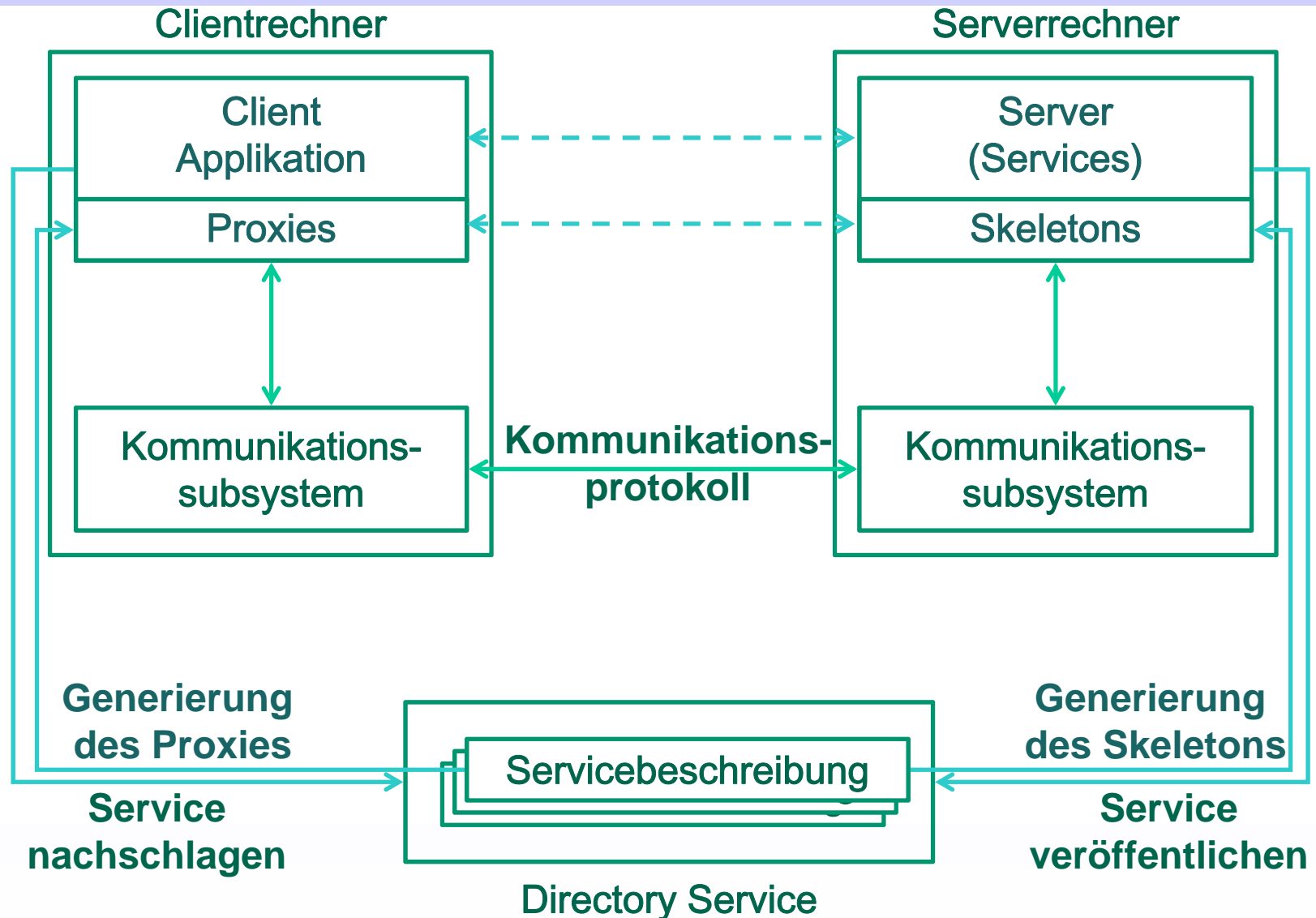


Universität zu Lübeck

# Kommunikations-Middleware

- Bietet höhere Kommunikations-Dienste für verteilte Anwendungen an
  - Verteilte Objekt-Technologie: erlaubt die Interaktion von Objekte über Netzwerke
    - Java RMI: für Java
      - Client und Server müssen in Java implementiert werden
    - Corba: Programmiersprache-unabhängig
      - Clients und Server können in verschiedenen Sprachen implementiert werden
  - RPC: Prozeduraufrufe über Netzwerke
    - erlaubt Aufruf der entfernten Prozeduren als wären es lokale Prozeduren
  - Web Services
    - Dienste sind über das Internet aufrufbar

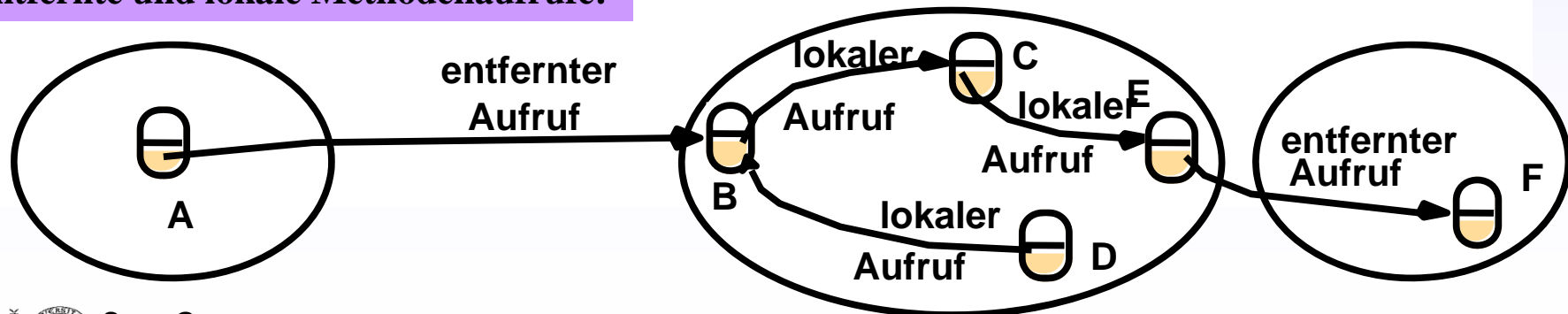
# Kommunikationsarchitektur - Allgemeines Schema



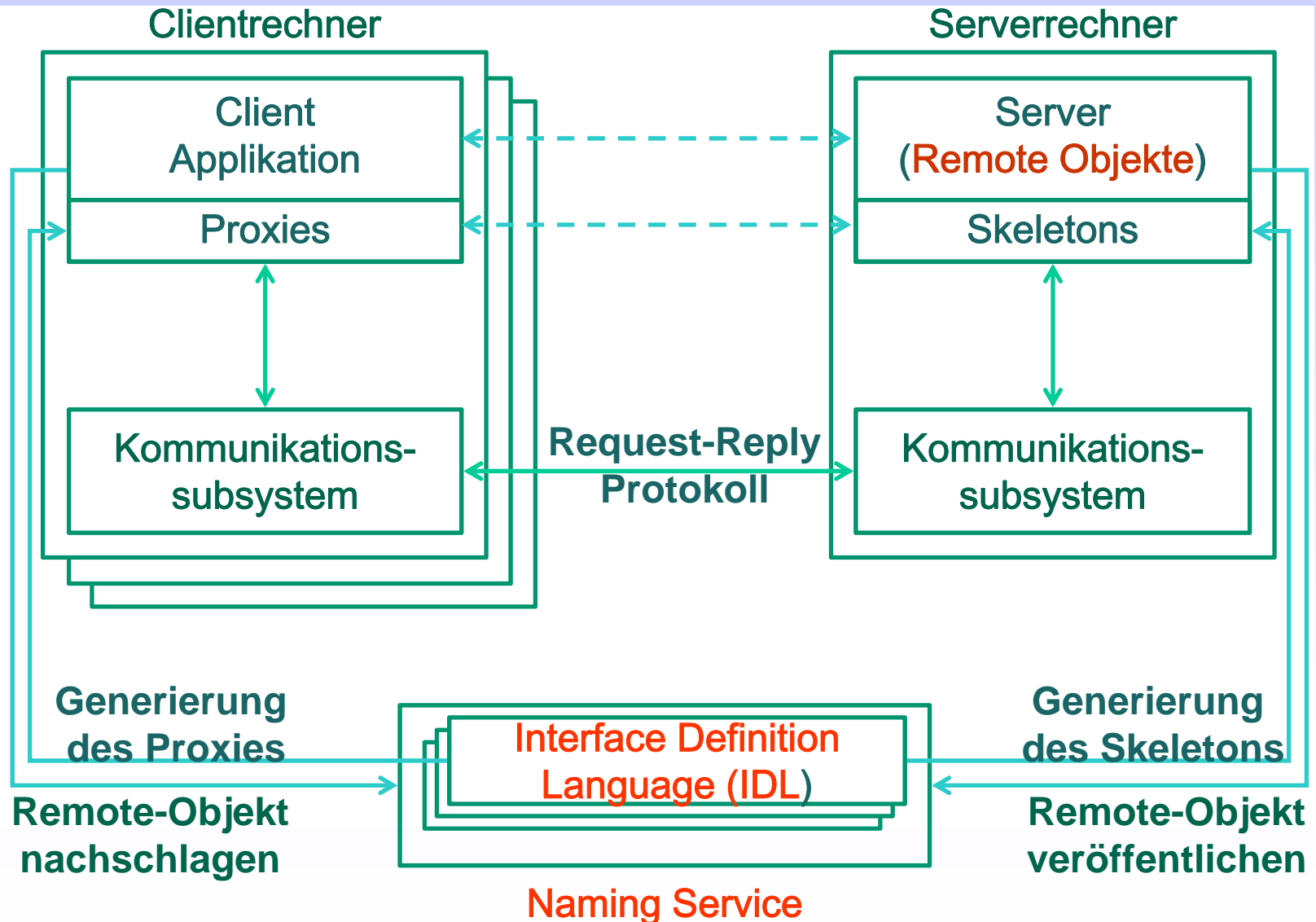
# Verteiltes Objekt-Modell

- **Methodenaufrufe können zwischen Objekten in verschiedenen Prozessen passieren**
  - Solche Methodenaufrufe heißen **entfernte Methodenaufrufe**
- **Objekte, die entfernte Methodenaufrufe erlauben, sind entfernte/remote Objekte**
- **Im Kern des verteilten Objektmodells:**
  - **Entfernte Objekt Referenzen:** ein Bezeichner für ein entferntes Objekt
    - Clients müssen die Referenz kennen, um die Methoden aufrufen zu können
  - **Entfernte Schnittstelle:** spezifiziert, welche Methoden entfernt aufrufbar sind
    - Remote Objekt implementiert die Methoden

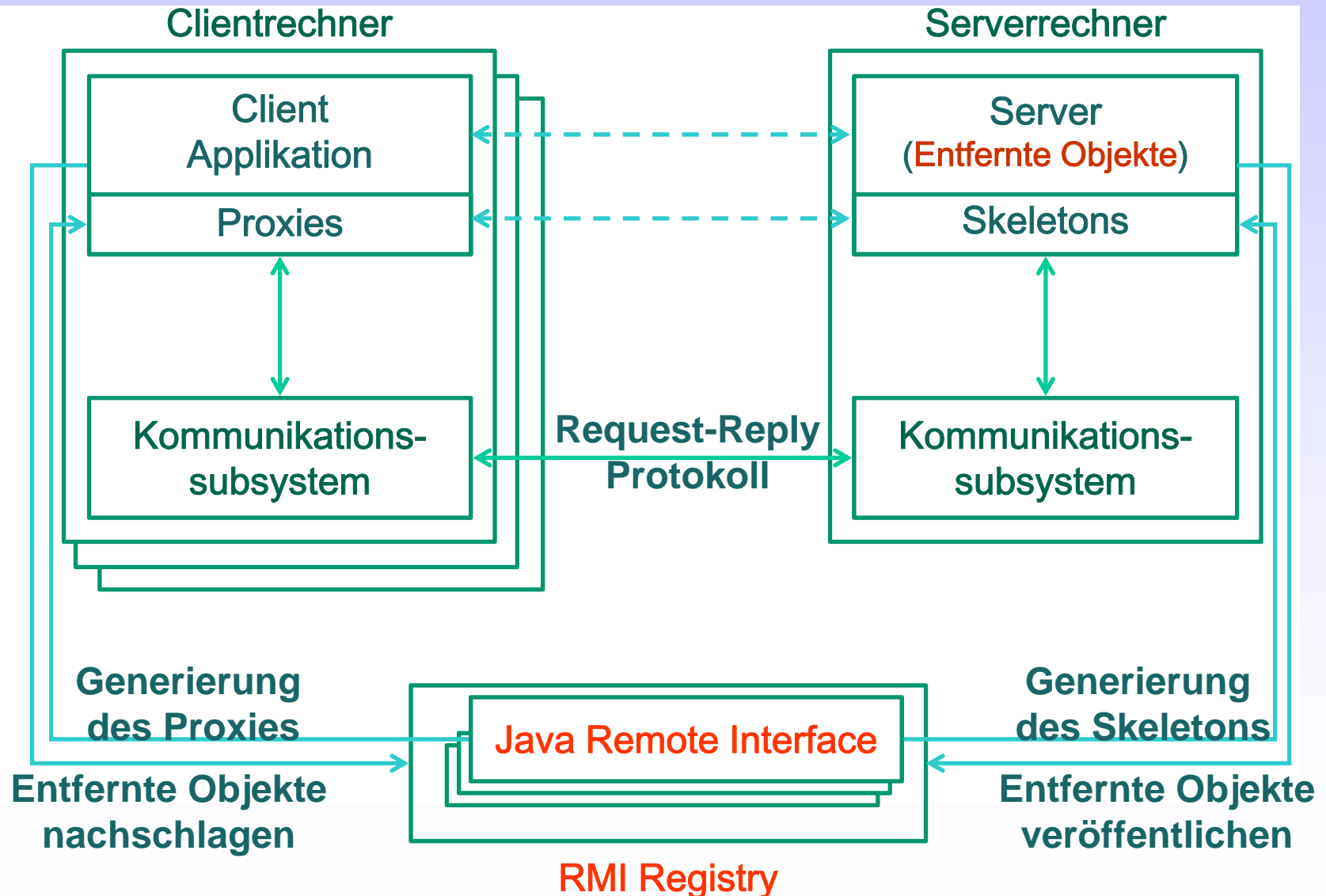
## Entfernte und lokale Methodenaufrufe:



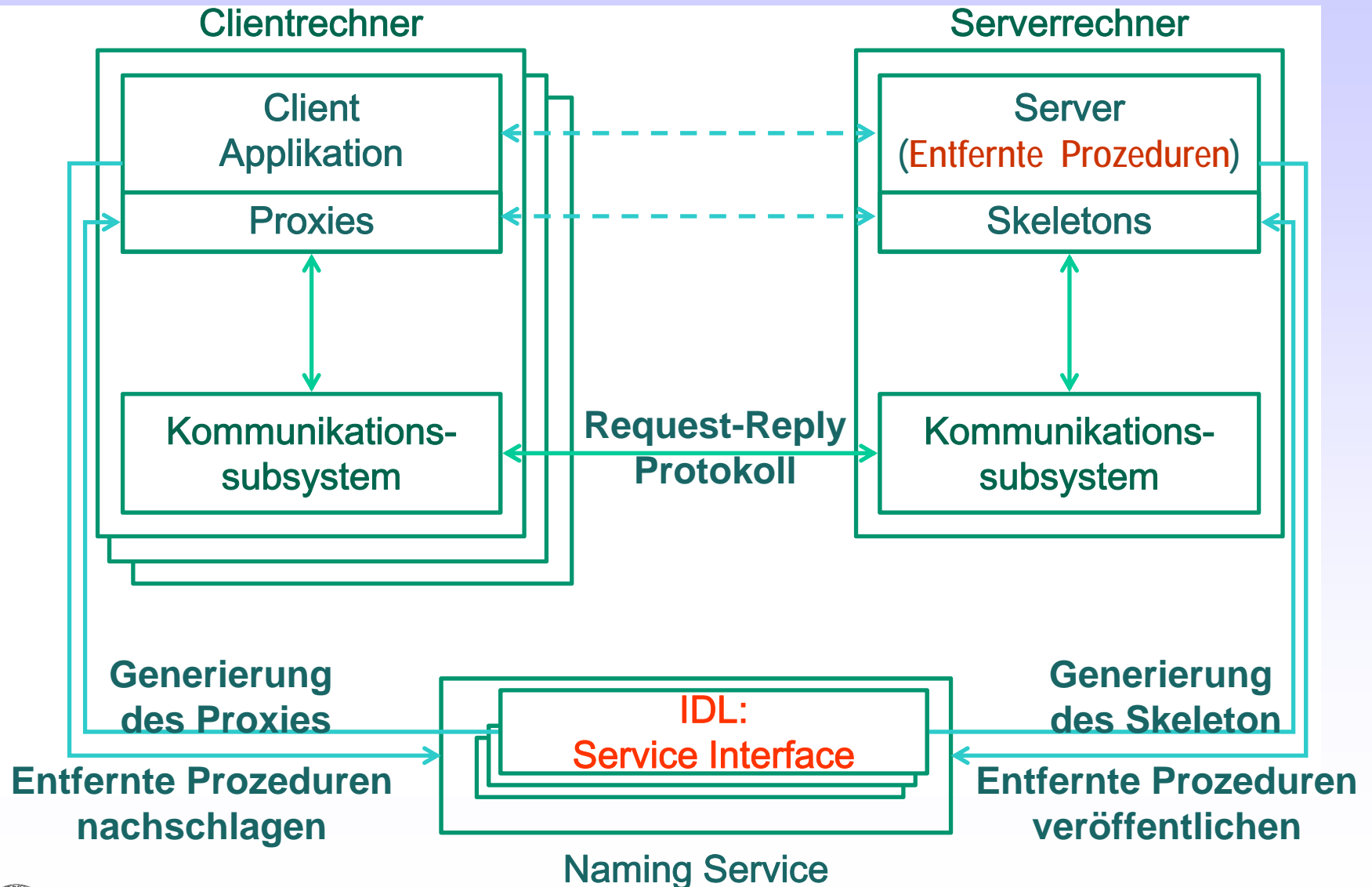
# Kommunikationsarchitektur – Corba



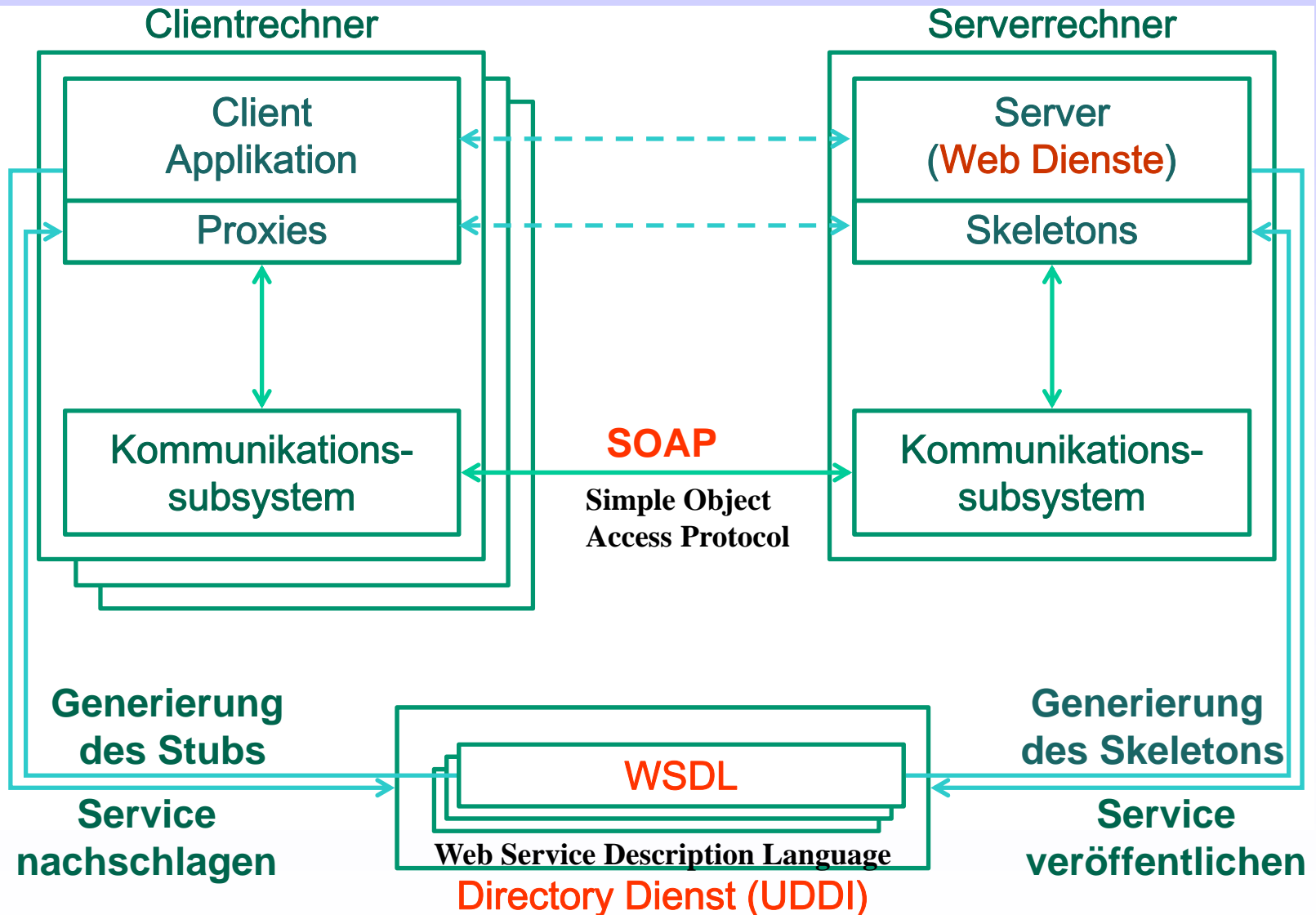
# Java Remote Method Invocation (Java RMI)



# Remote Procedure Call (RPC)



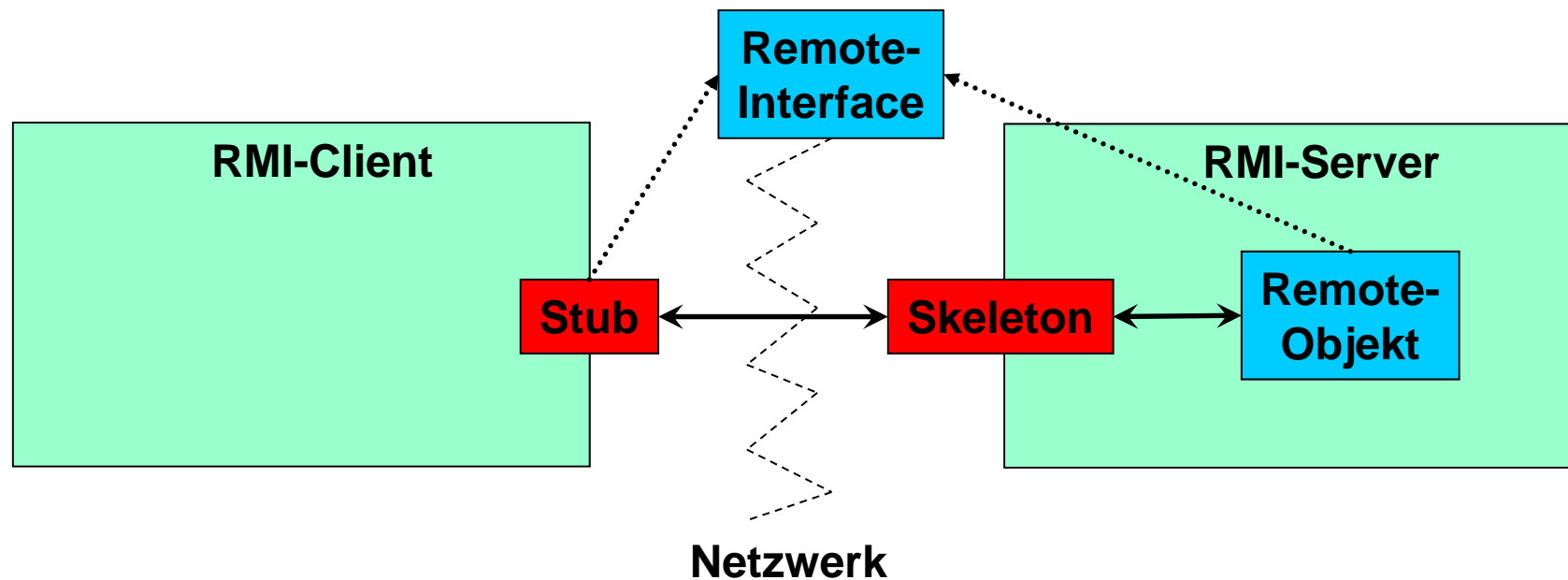
# Kommunikationsarchitektur – Web Services





# Java Remote Method Invocation (RMI)

- Realisierung von
  - verteilter Kommunikation und
  - Methodenaufrufe entfernter Objekte möglich



# Remote-Interface, Stub und Skeleton

- Remote-Interface vereinbart Methoden zwischen Client und Server  
z.B.:

```
public interface ServerInterface extends Remote {  
    public String transmitBinding(Bindings b) throws RemoteException;  
}
```

- Remote-Objekt implementiert Remote-Interface
- Client-Stub
  - implementiert das Remote-Interface
  - erledigt Kommunikation zwischen Client-Stub und Server-Skeleton auf Seite des Clients
- Server-Skeleton
  - erledigt Kommunikation zwischen Client-Stub und Server-Skeleton auf Seite des Servers
  - ruft Methoden des Server-Objektes auf
- Client-Stub und Server-Skeleton werden „automatisch“ generiert

# Server: Remote-Objekt und Java-Registry

- Bereitstellung der Server-Funktionalität
- implementiert Remote-Interface  
z.B.:

```
public class ServerProc extends UnicastRemoteObject implements ServerInterface {  
  
    protected ServerProc() throws RemoteException { super(); }  
  
    public String transmitBinding(final Bindings b) throws RemoteException {  
        System.out.println("Bindings retrieved:");  
        System.out.println(b.toString());  
        return "Server retrieved the bindings " + b.toString(); }  
}
```

- muss unter einem Namen bei der Java-Registry auf dem Server gebunden werden:  
z.B.:

```
try {  
    final Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);  
  
    final ServerProc obj = new ServerProc();  
    // Bind this object instance to the name "serverProc"  
    Naming.rebind("serverProc", obj);  
  
} catch (final Exception e) { System.out.println(e.getMessage()); }
```

# Client

- Lookup des Server-Objektes und Erhalten des Stubs, der das Server-Interface implementiert
- Danach Methodenaufrufe auf Stub-Objekt

z.B.:

```
try {
```

```
    ServerInterface stub =
```

```
        (ServerInterface) Naming.lookup("//pc18/serverProc");
```



Hat die Form: **//{hostname|ipaddress} [:port]/objectname**  
(anderes Bsp.: **//pc18.pool.ifis.uni-luebeck.de:1099/serverProc**)

```
System.out.println("Receiving text "
```

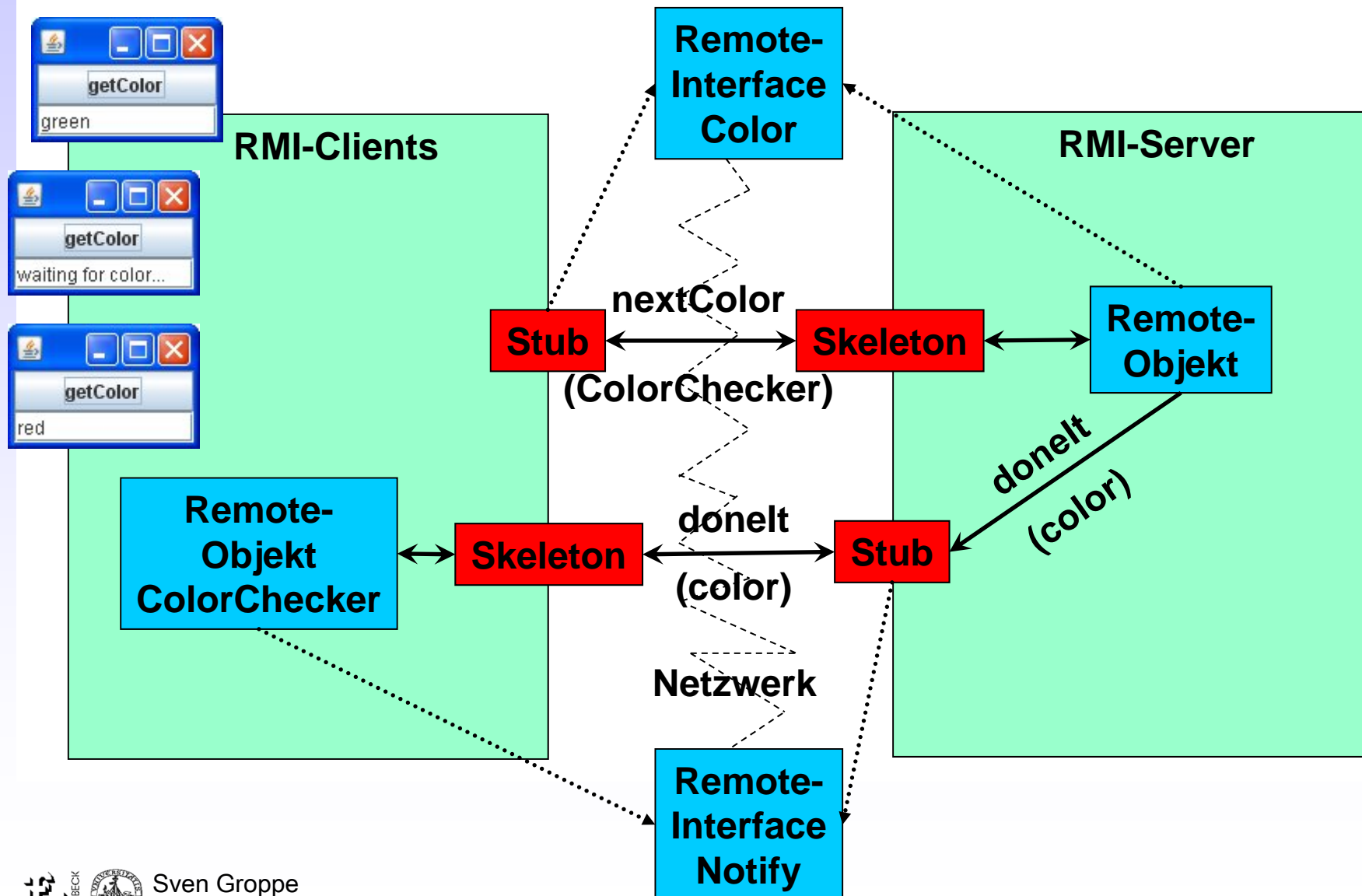
```
    + stub.transmitBinding(Bindings.createNewInstance());
```

```
} catch (final Exception e) { System.out.println(e.getMessage()); }
```

# Callbacks

- Bis jetzt:
  - Client ruft Methode eines Server-Objektes auf
- Server ruft Methode auf einen Client auf
  - => Callbacks
  - Dazu „Übergabe eines Remote Objektes“ vom Client an den Server

# Beispiel



# Remote Interfaces

- Für Server-Objekt:

```
public interface Color extends Remote {  
    public void nextColor(Notify n) throws RemoteException;  
}
```

- Für Client und Callback-Methode:

```
public interface Notify extends Remote {  
    public void doneIt(String color) throws RemoteException;  
}
```

# Server

```
public class Main extends UnicastRemoteObject implements Color {
    DoColor doColor;
    public Main() throws RemoteException {}

    public void nextColor(Notify n) throws RemoteException {
        doColor = new DoColor(n);
        doColor.start();
    }

    public static void main(String[] args) {
        try { Main server = new Main();
            final Registry reg =
                LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            Naming.rebind("COLOR", server);
        } catch (java.net.MalformedURLException e) {System.out.println(e);}
        catch (RemoteException e) {System.out.println(e);}
    }
}
```



# Thread auf Server

```
class DoColor extends Thread {
    private static String colors[]={"red","green","blue","orange","white"};
    private static int index = 0;
    private Notify notify;
    private static ReentrantLock lock=new ReentrantLock();

    public DoColor(Notify n) {
        notify = n;
    }

    public void run() {
        try { lock.lock();
            int i=0;
            try { index= (index + 1) % colors.length;
                i=index;
            } finally{ lock.unlock(); }
            Thread.sleep(3000); // simulating a lot of CPU work
            notify.done(i(colors[i]));
        } catch(Exception e) { System.out.println(e); };
    }
}
```

# Client

```
public class ColorClient extends
    javax.swing.JFrame {
    private javax.swing.JTextField colorLabel;
    private javax.swing.JButton getColor;
    private Color color;
    private ColorChecker notifyColor;

    public ColorClient() {
        initComponents();
        try { Remote remoteObject =
            Naming.lookup("COLOR");
            color = (Color) remoteObject ;
            notifyColor = new
                ColorChecker(colorLabel);
        } catch(Exception e){
            System.out.println(e); };
    }

    public static void main(String args[]) {
        new ColorClient().setVisible(true); }
}
```

```
private void initComponents() {
    getColor = new javax.swing.JButton();
    colorLabel=new javax.swing.JTextField();
    setDefaultCloseOperation(
        javax.swing.WindowConstants.EXIT_O
        N_CLOSE);
    getColor.setText("getColor");
    getColor.addActionListener(new
        java.awt.event.ActionListener() {
        public void actionPerformed(
            java.awt.event.ActionEvent evt) {
            try { colorLabel.setText(
                "waiting for color...");
                color.nextColor(notifyColor);
            } catch(Exception e) {
                System.out.println(e); }}}});
    getContentPane().add(getColor,
        java.awt.BorderLayout.CENTER);
    getContentPane().add(colorLabel,
        java.awt.BorderLayout.SOUTH);
    pack();
}
}
```

# ColorChecker – Remote Objekt auf dem Client

```
public class ColorChecker extends UnicastRemoteObject implements Notify {  
  
    private javax.swing.JTextField textfield;  
  
    public ColorChecker(javax.swing.JTextField tf) throws RemoteException {  
        textfield = tf;  
    }  
  
    public void donelt(String color) {  
        try {  
            textfield.setText(color);  
        } catch (Exception e){  
            System.out.println(e);  
        }  
    }  
}
```