Distributed Systems
Middleware Examples

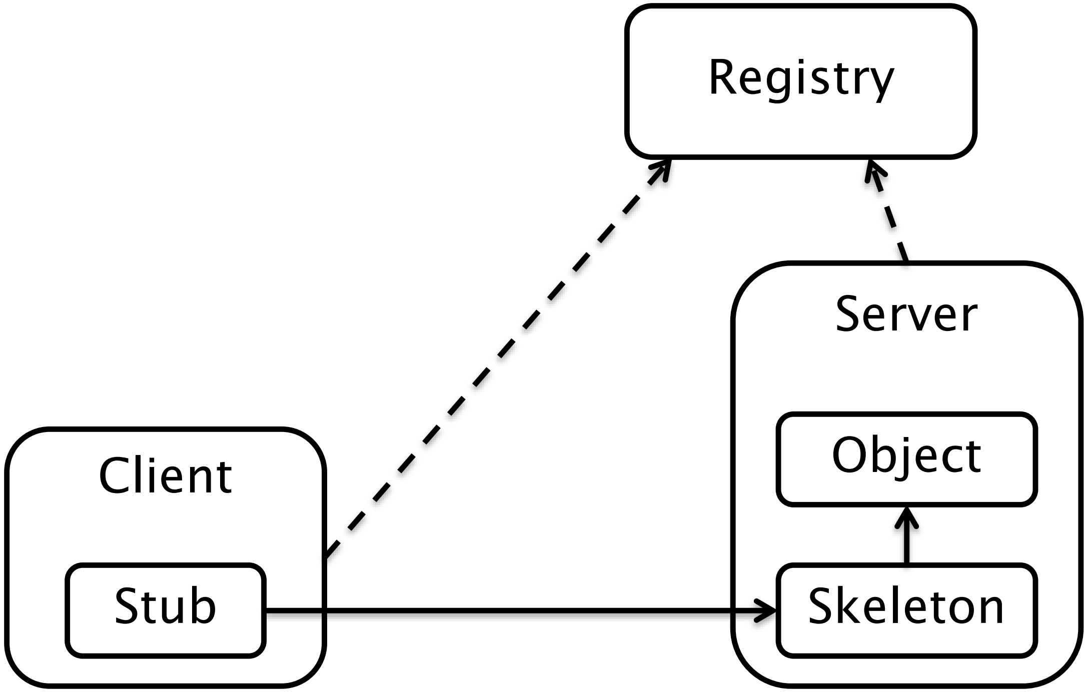
Dr.-Ing. Matthäus Wander

Universität Duisburg-Essen

Java RMI

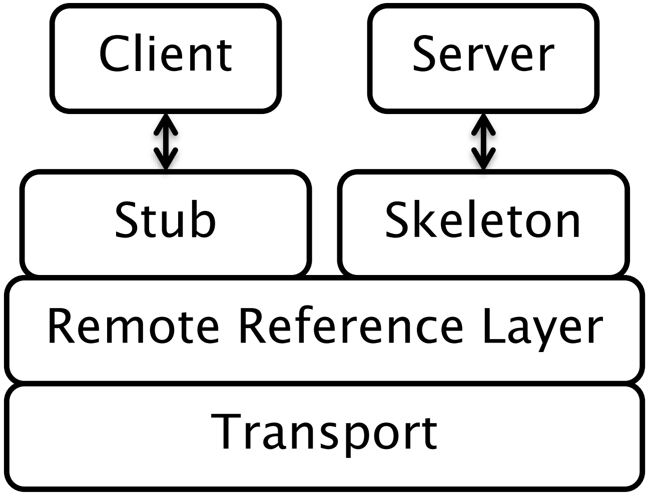
- Distributed Objects System
- Remote Method Invocation (RMI)
    - At-most-once semantics
- Integrated into Java
    - Relatively easy to use
    - Does not use an IDL
    - Not compatible with platforms other than Java

System Architecture



Layers

- Stub/Skeleton
  - Stub: client proxy object
  - Skeleton: server dispatcher and proxy object
  - (Un-)Marshalling



- Remote Reference Layer
  - Translates between local/remote references
  - Object activation
- Transport
  - Connection handling, network transmission

Remote Object

- Remote object resides on server and is accessed by client(s) via RMI middleware
- Remote object implements a Java interface
  - Must extend *java.rmi.Remote*
  - *Remote* is an empty interface, serves as marker that this object will be treated different from local objects
- Server exports remote object
  - Creates proxy („skeleton“), opens listening socket

⇨ How does the client locate the remote object?

Naming / Registry

- Server binds remote objects at a registry
    - Under a given name

    -
      ```
      registry = LocateRegistry.getRegistry(2223);
      ```

    -
      ```
      registry.bind("ChatService", remoteObj);
      ```
- Registry can run on server host or another host
    - Holds references to remote objects, including TCP/IP endpoints
    - Practical problem: server must know its own IP address when binding object (may fail with multiple IP addresses or NAT)

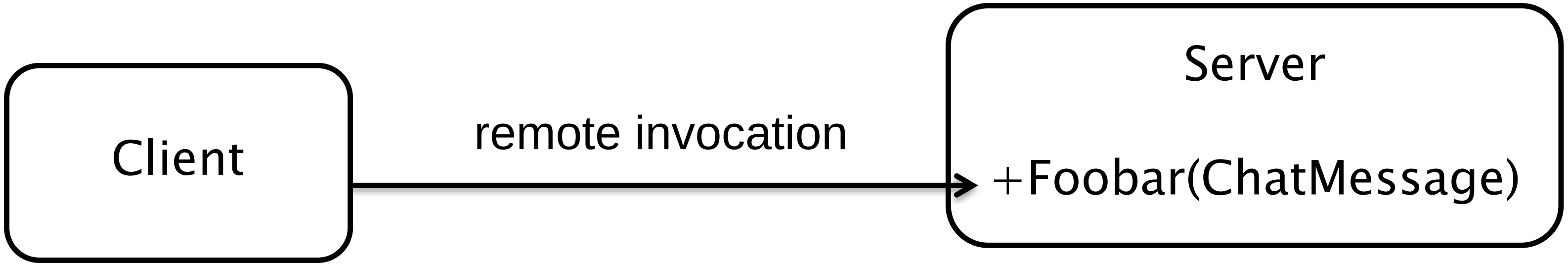
Client Lookup

- Client looks up remote object by registry host + port + object name
  - Can be looked up by URL (via *java.rmi.Naming*)
  - E.g. „//debby.vs.uni-due.de:2223/ChatService“
- Registry lookup returns object of type „Remote“
  - Cast to interface of remote object
  - `chatService = (ChatService) registry.lookup("ChatService");`
- RMI middleware automatically creates proxy („stub“) for remote objects
  - Client must know the interface of the remote object

Parameter Passing

- All parameters and return value must be serializable
  - Applies to primitives (int, long, double, …)
  - Applies to most standard classes (List, Set, Map, …)
- Custom classes must implement *java.io.Serializable* interface
  - Empty interface, usually works without extra code
- Java serializer uses reflection to serialize object
  - Goes through all object fields, serializes each
  - Throws exception with non-serializable data (e.g. file handle, socket)

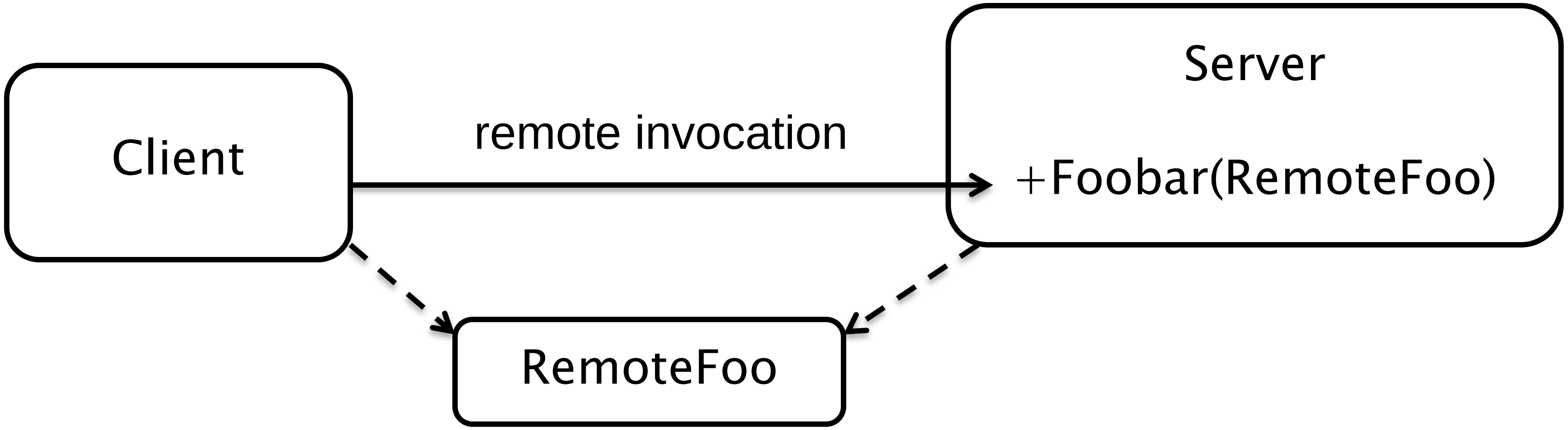
Parameter Passing



```
public class ChatMessage implements Serializable {
  public String nickname;
  public String message;
}
```

- Parameters are copied to server, but changes are not sent back to client!
  - Call-by-value semantics, although objects are passed with call-by-reference in local methods

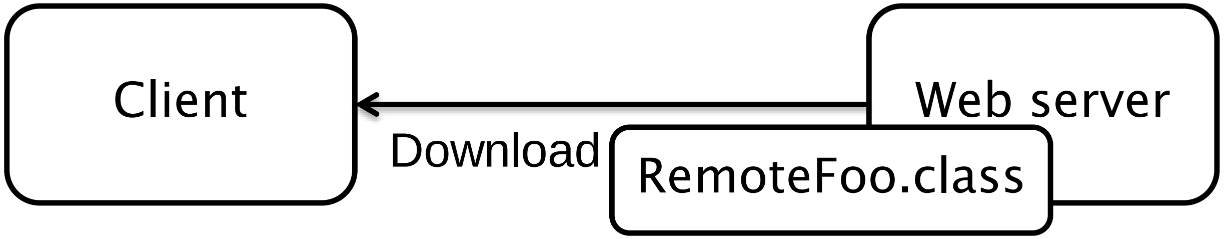
Parameter Passing



- Remote object references can be also passed as parameters to the server
  - Reference (the stub) is copied and transmitted
  - Stub contains IP endpoint ⇨ distribution transparency
  - Remember: actual object resides on one server, independent of number of references to it

Code Distribution

- Clients and servers both need to know:
  - The interface of the remote object
  - Implementation of all parameters and return types
  - ⇨ How to distribute code (.class files)?
- Deploy same .class files with clients and servers
  - Problem: software updates, protocol updates
- Dynamic code loading



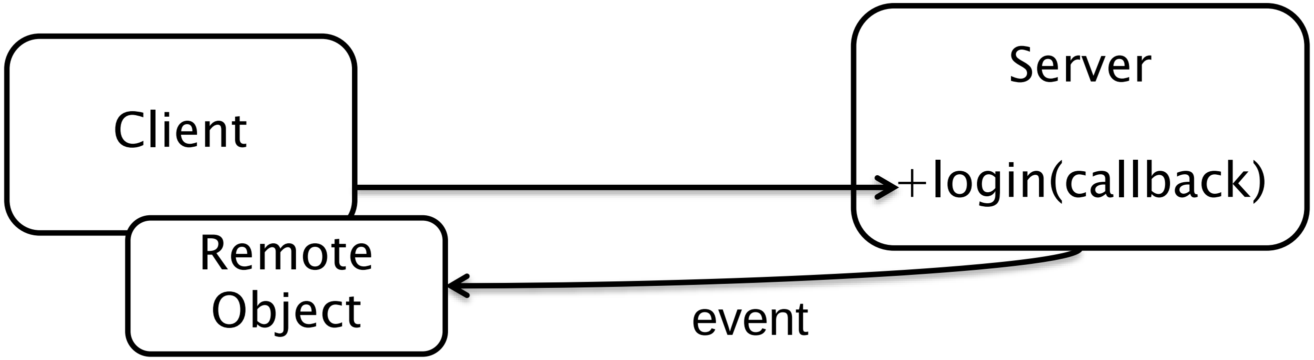
Exception Handling

```
public interface ChatService extends Remote {
  public ChatContext login(String nickname) throws RemoteException;
}
```

- Methods of a remote object interface may throw a RemoteException
  - When distribution transparency fails, e.g. server down
  - Thrown by stub, must be handled by client
- Server may throw exceptions, too
  - All exceptions are serializable
  - Transfered over network, thrown by stub

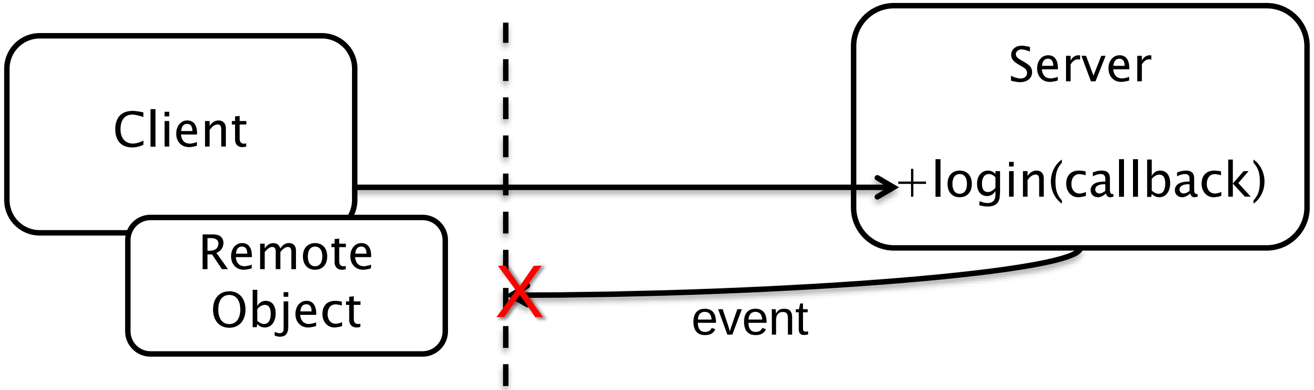
Push Event from Server to Client

- How to notify the client when a server event has happened?
- Polling: client regularly asks server
- Callback: server invokes method on client
  - Client must pass remote object stub to server



Network Address Translation

- RMI uses TCP as transport
  - Remote objects are accessed via a TCP connection
  - Clients usually behind NAT router ⇨ callback fails



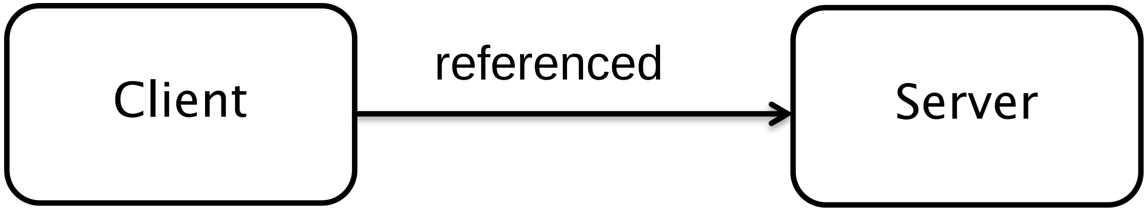
Garbage Collection

- Java uses Garbage Collection (GC) to remove no longer needed objects

⇨ How to remove old remote objects?

  - Reference counting

- Client stubs inform server of remote reference
  - „referenced“ message transfered over network
  - When client GC removes stub ⇨ send „unreferenced“



Garbage Collection

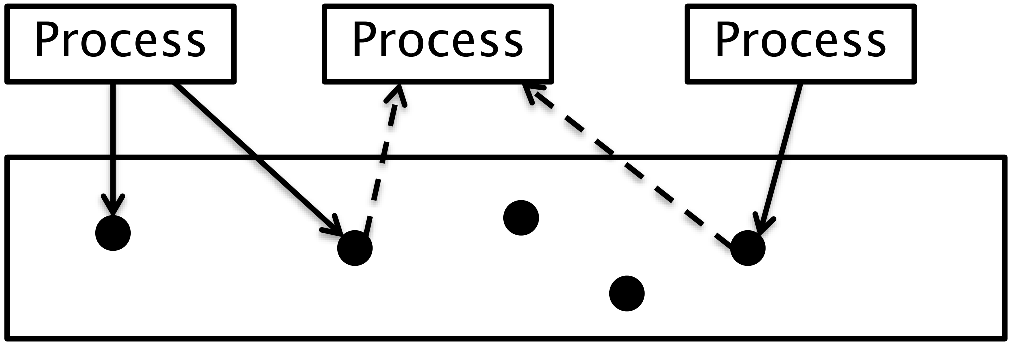
- How to deal with client crashes?
- Remote objects have a lease time
  - Clients must actively renew their lease
  - Happens automatically by RMI middleware
- Lease expires ⇨ server unreferences object
- 0 references ⇨ object is removed by server GC

Conclusion

- RMI integrated into Java language
  - Platform-specific (no generic IDL)
- Distribution visible to application developer
  - *Remote* interface, RemoteException handling
- Behaves mostly like local method calls
  - But: no call-by-reference for serialized parameters
- Practical problems:
  - Does not work well with NAT (e.g. client callbacks)
  - Not very efficient (e.g. reflection is rather slow)

Tuple Spaces / JavaSpaces

- Model for building distributed systems
  - Tuple Spaces: generic concept
  - JavaSpaces: Java-specific implementation
  - Jini / Apache River: middleware for JavaSpaces
- Put data entries into a shared space
- Get notifications about new entries



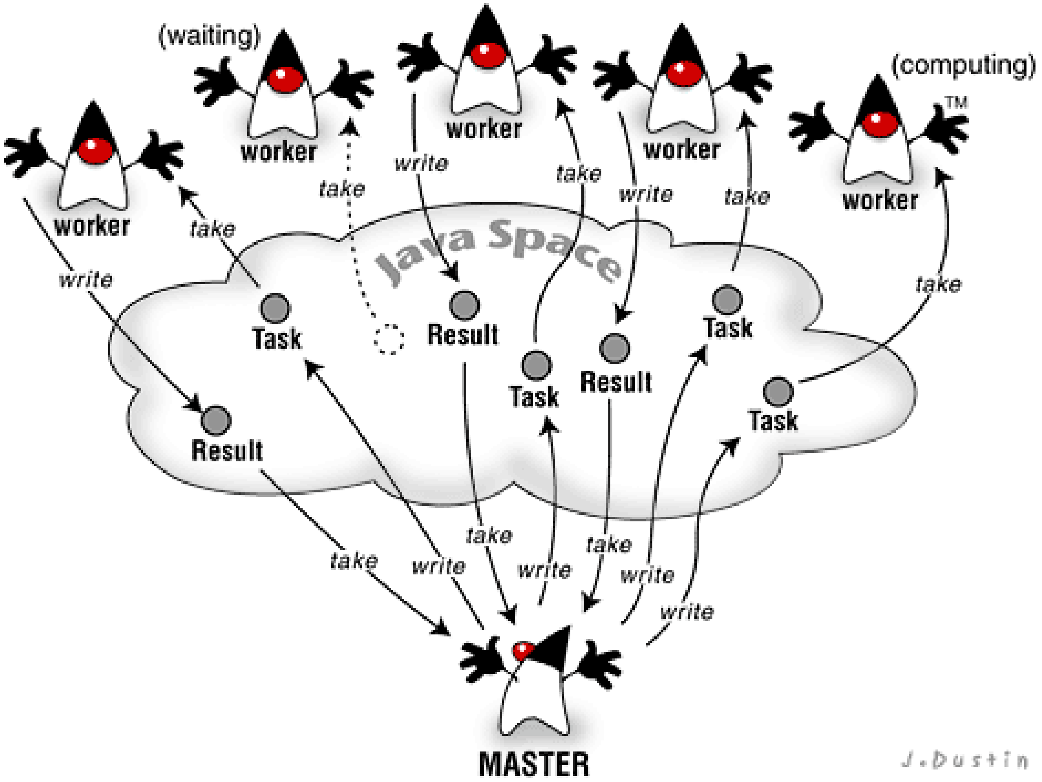
Concept

- Distributed shared space
- Atomic operations
  - Write(): put entry into space (no overwrite!)
  - Read(): get copy of entry
  - Take(): get copy of entry and remove it
- What is an entry? Serializable object
- How to modify an entry?
  - Take, modify local copy, write

  ⇨ implicit synchronization, no locking or coordination necessary

Example Use Case: Distributed Computing

- Master writes tasks into space and takes results
- Workers take tasks and write results



Looking up entries

- Entries are read by associative lookups
  - Not: lookup by a single identifier
- Create entry template and read(template):
  - Take some custom class, e.g. MyEntry
  - Set some fields that must match, e.g. MyEntry.name=„foo“
  - Leave others null that can match against any value, e.g. MyEntry.value=null
- Will return an MyEntry instance that matches
  - Or block until one becomes available

Conclusion

- Easy communication and synchronization
  - Transactions (multiple take/write) also possible
- Persistent data storage built into concept
- Easily scalable (add more processes/workers)
- Very different model
  - How to map other applications than master-worker?
  - E.g. a chat? A multiplayer card playing game?
- Inefficient implementation (RMI-based)