Distributed Systems/Middleware
Core Communication Facilities

Alessandro Sivieri
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
sivieri@elet.polimi.it

http://corsi.dei.polimi.it/distsys

Slides based on previous works by Alessandro Margara

Politecnico di Milano

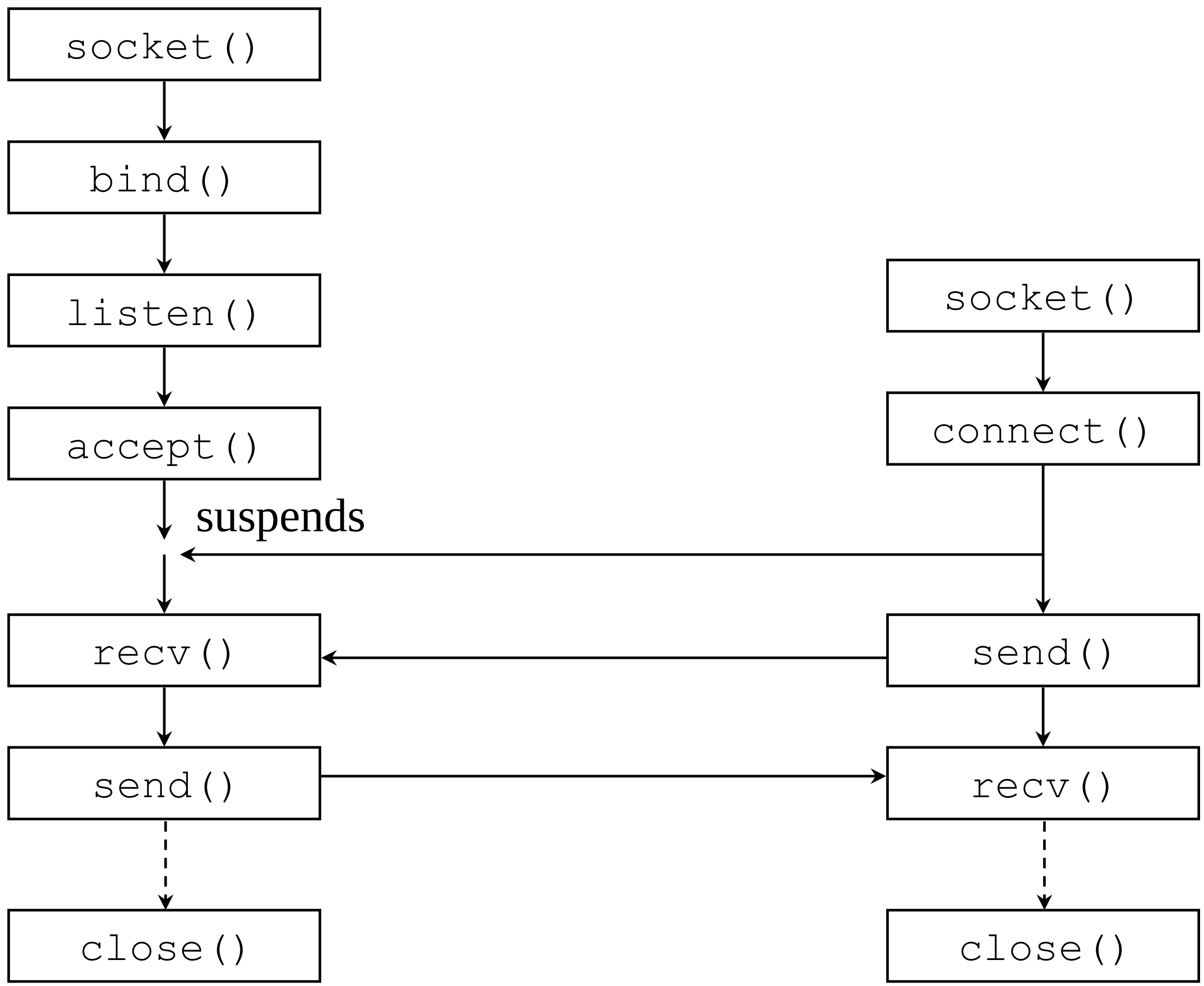
Contents

- **Fundamental inter-process communication services**
  - **Unicast**
    - **Stream (TCP) and datagram (UDP) sockects in C and Java**
  - **Multicast**
    - **Multicast datagrams in Java**
- Remote procedure call
  - Fundamentals
  - RPC in C (under Unix/Linux)
- Remote method invocation
  - Fundamentals
  - Java RMI

From network protocols to communication services

- Unicast TCP and UDP (and multicast IP) are well know network protocols
  - The related RFCs describe how they work in practice (on top of IP)
  - But how a poor programmer can take advantage of such protocols?
- Socket is the answer!
  - First appeared in Unix BSD in 1982
  - Today available for every platform
- Sockets provide a common abstraction for inter-process communication
  - Unix and Internet sockets exists. Here we are interested in the latter
  - Allows for connection-oriented (stream i.e., TCP) or connectionless (datagram, i.e., UDP) communication

Stream sockets in C



Data structures

Politecnico di Milano

- Generic socket address
```
struct sockaddr {
       u_short sa_family; /* socket type */
       char sa_data[14];  /* address */
}
```
- Internet socket address
```
struct sockaddr_in {
       u_short sin_family; /* socket type = AF_INET */
       u_short sin_port;   /* port number  */
       struct in_addr sin_addr; /* IP address (4 bytes) */
       char sin_zero[8];   /* padding */
}
```
  - `sin_port` and `sin_addr` must be in "network byte order"
- IP address (a structure for historical reasons)
```
struct in_addr {
    u_long s_addr; // that's a 32-bit long, or 4 bytes
}
```
- Routines to convert from the platform byte order to the network and vice versa
```
u_long htonl(u_long hostlong)
u_short htons(u_short hostshort)
u_long ntohl(u_long netlong)
u_short ntohs(u_short netshort)
```
- Routines to convert a string address "131.112.45.67" in a numeric (`u_long`) address and vice versa
```
u_long inet_addr(char* str)
char *inet_ntoa(struct in_addr addr)
```

Main API

```
int socket(int family, int type, int protocol)
```

- family is PF_UNIX or PF_INET
- type is SOCK_STREAM or SOCK_DGRAM
- protocol is IPPROTO_UDP or IPPROTO_TCP (or just 0)
- Returns a socket file descriptor or -1

```
int bind(int sockfd, struct sockaddr *myaddr, int
   addrlen)
```

- sockfd is the socket file descriptor to bind
- myaddr is the address to be bound
- addrlen is the length of the myaddr structure
- Returns 0 or -1

```
int listen(int sockfd, int maxconn)
```

- sockfd is the socket file descriptor
- maxconn is the maximum number of pending connections queued by the OS
- Returns 0 or -1

Politecnico di Milano

Main API

```
int accept(int sockfd, struct sockaddr *partner, int
   *len)
```

- sockfd is the socket file descriptor
- partner is the address of the accepted client
- len is the length of the partner structure
- Returns the socket file descriptor of the accepted connection or -1

```
int connect(int sockfd, struct sockaddr *server, int
   len)
```

- sockfd is the socket file descriptor
- server is the address of the other host to connect to
- len is the length of the server structure
- For stream sockets the call connects to the socket that is bound to the address specified by server. For datagram sockets the server is the address to which datagrams are sent by default, and the only address from which datagrams are received. Returns 0 or -1

Main API

```
int send(int sockfd, char *buf, int len, int flags)
```

- sockfd is the socket file descriptor
- buf is a pointer to the buffer that contains the message to send
- len is the message length
- flags specify advanced flags (use 0 for simplicity)
- Returns the number of bytes transmitted or -1

```
int sendto(int sockfd, char *buf, int len, int flags, struct
   sockaddr *to, int tolen)
```

- sockfd, buf, len, flags as before
- to is the address of the recipient
- tolen is the length of the to structure

```
int recv(int sockfd, char *buf, int len, int flags)
```

- sockfd is the socket file descriptor
- buf is a pointer to the buffer that will be filled with the message received
- len is the buffer length
- flags is the same as before
- Returns the number of bytes received or -1

```
int recvfrom(int sockfd, char *buf, int len, int flags,
   struct sockaddr *from, int *fromlen)
```

- sockfd, buf, len, flags as before
- from is filled with the address of the sender
- fromlen is the length of the from structure

Example: TCP in C, the client

Politecnico di Milano

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[]) {
  int sock, port;
  char *host, *msg;
  struct sockaddr_in sa;
  /* Parse command line arguments */
  if(argc<3) {printf("Usage: client
    <host> <port> [<message>]\n");
    exit(-1);}
  host=argv[1];
  port=atof(argv[2]);
  if(argc==3) msg="Let me try...";
  else msg=argv[3];
  /* Set the destination address */
  memset(&sa, 0, sizeof(struct
    sockaddr_in));
  sa.sin_family=AF_INET;
  sa.sin_port=htons(port);
  sa.sin_addr.s_addr=inet_addr(host);
  /* Create the socket & connect it */
  sock=socket(AF_INET, SOCK_STREAM, 0);
  if(sock<0) {
    perror("creating the socket");
    exit(-1);
  }
  if(connect(sock, (struct sockaddr *)
    &sa, sizeof (sa)) < 0) {
    perror("binding");
    exit(-1);
  }
  /* Send the message */
  if(send(sock,msg,strlen(msg),0)<0) {
    perror("writing");
    exit(-1);
  }
  /* Close the connection */
  close(sock);
  return 0;
}
```

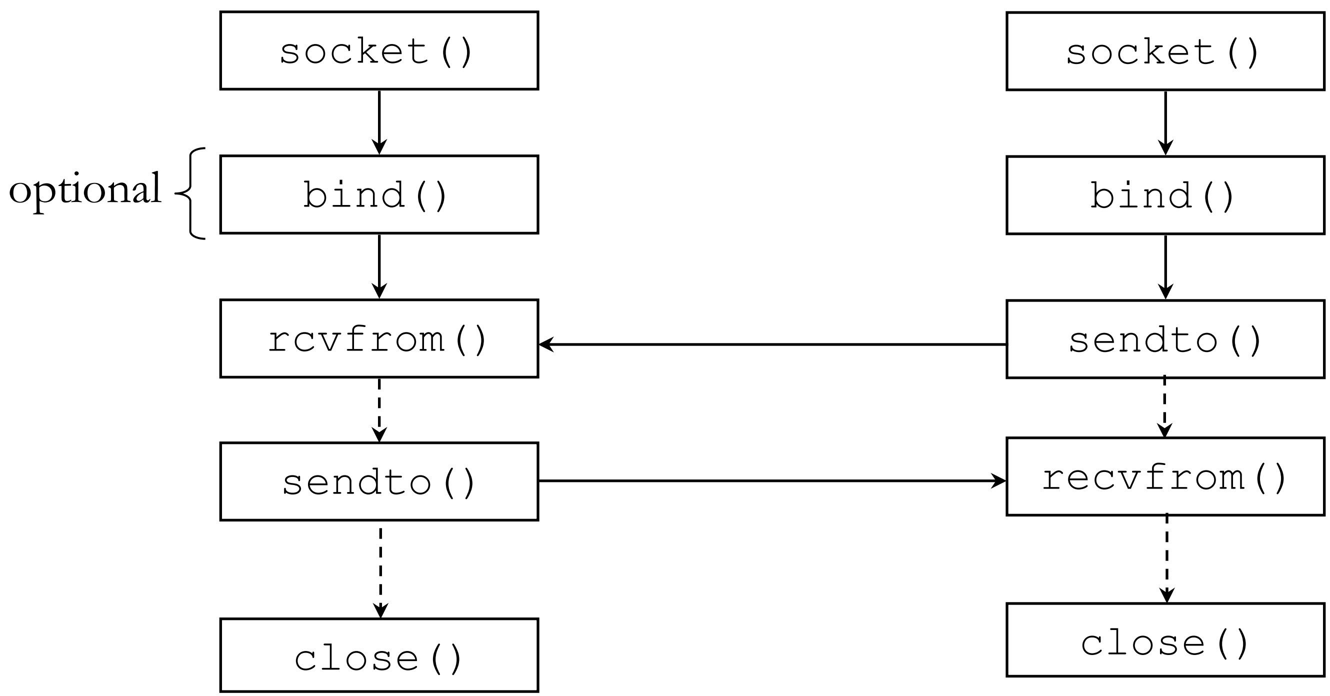
Example: TCP in C, the server

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFLEN 1024

int main(int argc, char *argv[]) {
  int sock, s, port, n;
  struct sockaddr_in sa;
  char buf[BUFLEN];
  /* Parse command line arguments */
  if(argc!=2) {
    printf("Usage: server <port>\n"); exit(-1);
  }
  port = atof(argv[1]);
  /* Set the socket addr to accept connections */
  memset(&sa, 0, sizeof(struct sockaddr_in));
  sa.sin_family = AF_INET;
  sa.sin_port = htons(port);
  sa.sin_addr.s_addr = htonl(INADDR_ANY);
  /* Create the socket and bind it */
  sock = socket (AF_INET, SOCK_STREAM, 0);
  if(sock<0) {
    perror("creating the socket");exit(-1);
  }
  if(bind(sock,(struct sockaddr *)&sa,
     sizeof(sa))<0){
    perror("binding");
    exit(-1);
  }
  if(listen(sock,0)<0) {
    perror("listening"); exit(-1);
  }
  /* Accept incoming requests (main loop) */
  while(1) {
    /* Accept a new connection */
    s = accept(sock,NULL,NULL);
    if(s<0) { perror("accepting"); exit(-1); }
    /* Receive incoming data */
    while(1) {
      memset(buf, 0, sizeof(buf));
      if((n = read(s, buf, BUFLEN-1))<0)
        perror("Receiving data");
      else if(n==0) break;
      else {
        printf("%s\n",buf);
        if(buf[n-1]==4) break; /* stop if EOT
   */
      }
    }
    close(s);
  }
  close(s);
  close(sock);
  return 0;
}
```

Exercises

- Write a multithreaded version of the previous server capable of managing multiple connections in parallel
- We have to implement a client/server chat application
  - The server accepts tcp connections from incoming clients
  - Once connected, a client sends its nickname to the server
    - Afterwards, it sends all the sentences (i.e., "\n" terminated sequences of characters) typed by its user to the server
  - The server retransmits each sentence received to all the other clients, with the sender's nickname added at the beginning of the sentence
  - Clients write on sceen all the sentences received
  - Implement the client in C

Datagram sockets in C



Exercise

- Write a C program that waits for datagrams and immediately resends them back

Politecnico di Milano

Socket programming in Java

- Same approach as in C, but much simpler
- Five main classes (part of the `java.net` package):
  - `InetAddress`: provide methods to obtain the address of an host known its hostname (accesing the DNS) and vice versa
  - `ServerSocket`: Used by the server to accept incoming connections
  - `Socket`: The main class for stream communication
  - `DatagramPacket`: The messages sent through `DatagramSockets`
  - `DatagramSocket`: Used to send or receive `DatagramPackets`
- Actual stream communication is realized by using the input and output streams associated to sockets (i.e., through the classes provided for standard, stream-oriented I/O)

Example: The TCP server in Java

Politecnico di Milano

```
import java.net.*;
import java.io.*;

public class TCPServer {
  public static void main(String[] args) throws java.io.IOException {
    int port;
    ServerSocket sock;
    Socket s;
    BufferedReader in;
    String line;
    // Parse command line arguments
    if(args.length!=1) {
      System.err.println("Usage: java Server <port>");
      System.exit(-1);
    }
    port = Integer.parseInt(args[0]);
    // Create the server socket
    sock = new ServerSocket(port);
    // Accept incoming requests (main loop)
    while(true) {
      // Accept a new connection
      s = sock.accept();
      // Receive incoming data
      in = new BufferedReader(new InputStreamReader(s.getInputStream()));
      while((line = in.readLine())!=null) {
        System.out.println(line);
} } } }
```

Politecnico di Milano

Exercises

- Write a Java client for the previous server
- Write a multithreaded version of the previous server capable of managing multiple connections in parallel
- Write a Java server of the chat application described before

Example: UDP in Java

```
import java.net.*;

public class UDPPeer {
  public static void main(String[] args) throws Exception {
    DatagramSocket sock = null;
    DatagramPacket pack = null;
    String msg = null;
    InetAddress dest = null;
    int port = 0;
    // Parse command line arguments
    if(args.length==1) port=Integer.parseInt(args[0]);
    else if(args.length==3) {
      dest = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
      msg = args[2];
    } else {
      System.err.println("Usage: java Peer [<port> | <host> <port> <msg>]"); System.exit(-1);
    }
    if(msg!=null) {
      // Create a new socket & send a packet
      sock = new DatagramSocket();
      pack = new DatagramPacket(msg.getBytes(), msg.length(), dest, port);
      sock.send(pack);
    } else {
      // Create a new socket & receive a packet
      sock = new DatagramSocket(port);
      pack = new DatagramPacket(new byte[128],128);
      sock.receive(pack);
      msg=new String(pack.getData(), pack.getOffset(), pack.getLength());
      System.out.println(msg);
    }
    sock.close();
} }
```

Exercise

Politecnico di Milano

- Write a Java program that sends datagrams to the "immediate replier" formerly developed in C, waiting for them to come back and printing the round-trip time

Politecnico di Milano

A short note on Java serialization

- The classes `ObjectInputStream` and `ObjectOutputStream` enable direct reading and writing on a serialization stream
  - E.g., `os.writeObject(myObj);`
- To be serialized, an object must implement the `Serializable` interface
- Serialization performs a *deep copy* of the object (object closure)
  - The serialized image contains all the objects in the root, the objects in these latter objects, and so on
  - Primitives types are simply placed on the stream
  - Classes are serializable, but only a class descriptor is inserted
    - useful for resolving the class, possibly through dynamic downloading
- The programmer can prevent serialization of an attribute by declaring it `transient`
- Serialization can be redefined
  - Typically done in conjunction with class loader redefinition

IP multicast: Fundamentals

Politecnico di Milano

- IP multicast is a network protocol to efficiently deliver UDP datagrams to multiple recipients
- The Internet Protocol reserve a class D address space, from 224.0.0.0 to 239.255.255.255, to multicast *groups*
- Component interested in receiving multicast datagrams addressed to a specific group must *join* the group
  - IP multicast provides only open groups: It is not necessary to be a member of a group in order to send datagrams to the group
- As in TCP and UDP it is also necessary to specify a *port*
  - It is used *by the OS* to decide which process on the local machine to route packets to
  - Unlike in TCP or UDP, there can be many sockets which receive multicast datagrams off a single local port
- Note: most routers are configured to *not* route multicast packets outside the LAN

Politecnico di Milano

IP multicast in Java

- Java provides a subclass of `DatagramSocket`, the `MulticastSocket` class, to easily implement multicast communication
- It adds two main methods to `DatagramSocket`
  - `joinGroup`
  - `leaveGroup`

Example: Multicast in Java

Politecnico di Milano

```
import java.net.*;

public class MulticastPeer {
  public static void main(String[] args) throws Exception {
    MulticastSocket sock = null;  DatagramPacket pack = null;
    InetAddress group = null;  int port = 0;
    String msg = null;
    // Parse command line arguments
    if(args.length<2 || args.length>3) {
      System.err.println("Usage: java MulticastPeer <group <port> [<msg>]");
      System.exit(-1);
    }
    group = InetAddress.getByName(args[0]);  port = Integer.parseInt(args[1]);
    if(args.length==3) msg = args[2];
    if(msg!=null) {
      // Create a new multicast socket & send a packet
      sock = new MulticastSocket();
      pack = new DatagramPacket(msg.getBytes(), msg.length(), group, port);
      sock.send(pack);
    } else {
      // Create a new socket & receive a packet
      sock = new MulticastSocket(port);
      sock.joinGroup(group);
      pack = new DatagramPacket(new byte[128],128);
      sock.receive(pack);
      msg = new String(pack.getData(), pack.getOffset(), pack.getLength());
      System.out.println(msg);
    }
    sock.close();
} }
```

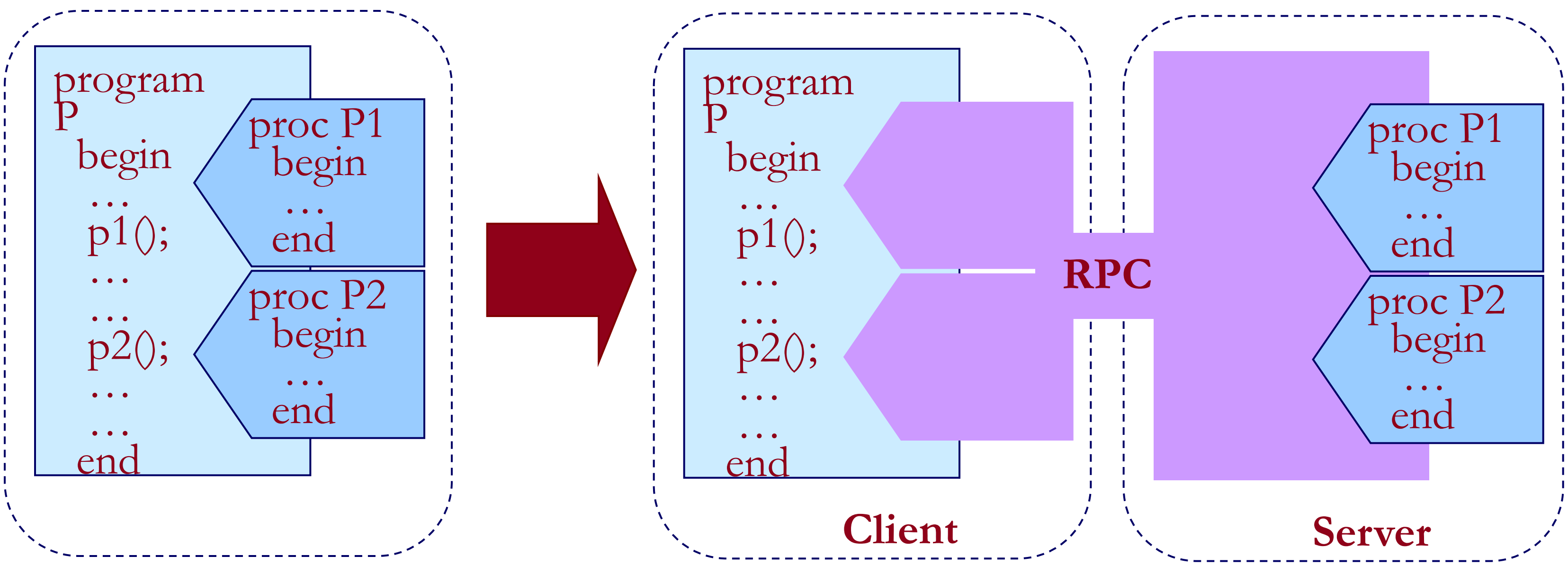
Politecnico di Milano

Contents

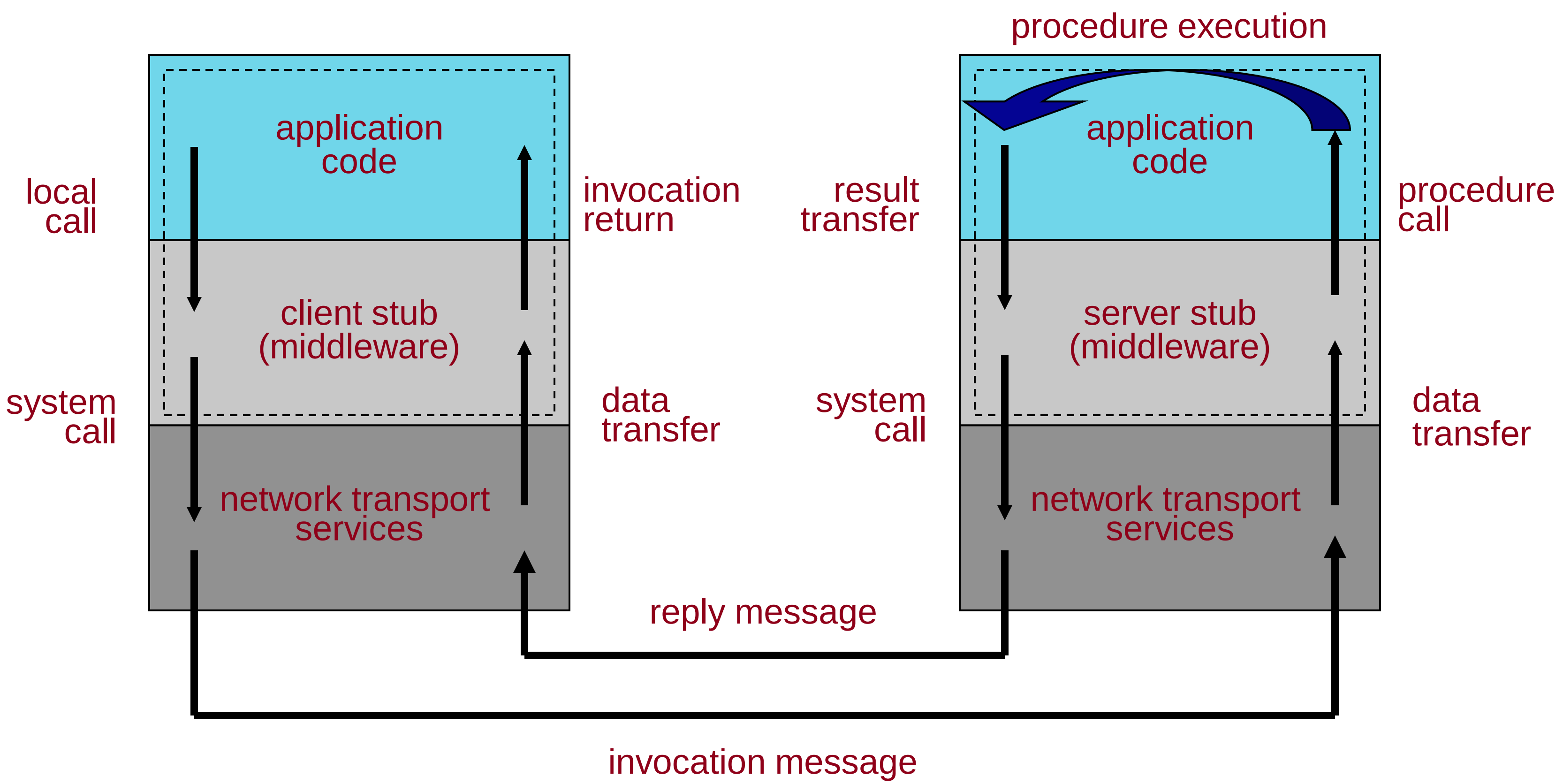
- Fundamental inter-process communication services
  - Unicast
    - Stream (TCP) and datagram (UDP) sockects in C and Java
  - Multicast
    - Multicast datagrams in Java
- **Remote procedure call**
  - **Fundamentals**
  - **RPC in C (under Unix/Linux)**
- Remote method invocation
  - Fundamentals
  - Java RMI

RPC: Fundamentals

- Problem: client-server interaction is handled through the OS primitives for I/O ➔ difficult to develop applications
- Idea (Sun Microsystems in the early 80s): enable remote access through the well-known procedure call programming model



RPC: How does it works



Parameter passing: Marshalling and serialization

- Passing a parameter poses two problems:
  - Structured data (e.g., structs/records, objects) must be ultimately flattened in a byte stream
    - Called *serialization* (or pickling, in the context of OODBMSs)
  - Hosts may use different data representations (e.g., little endian vs. big endian, EBCDIC vs. ASCII) and proper conversions are needed
    - Called *marshalling*
- Middleware provides automated support:
  - The marshalling and serialization code is automatically generated from and becomes part of the stubs
  - Enabled by:
    - A language/platform independent representation of the procedure's signature, written using an *Interface Definition Language* (IDL)
    - A data representation format to be used during communication

Politecnico di Milano

The role of IDL

- The *Interface Definition Language* (IDL) raises the level of abstraction of the service definition
  - It separates the service *interface* from its *implementation*
  - The language comes with “mappings” onto target languages (e.g., C, Pascal, Python…)
- Advantages:
  - Enables the definition of services in a language-independent fashion
  - Being defined formally, an IDL description can be used to automatically generate the service interface code in the target language

Politecnico di Milano

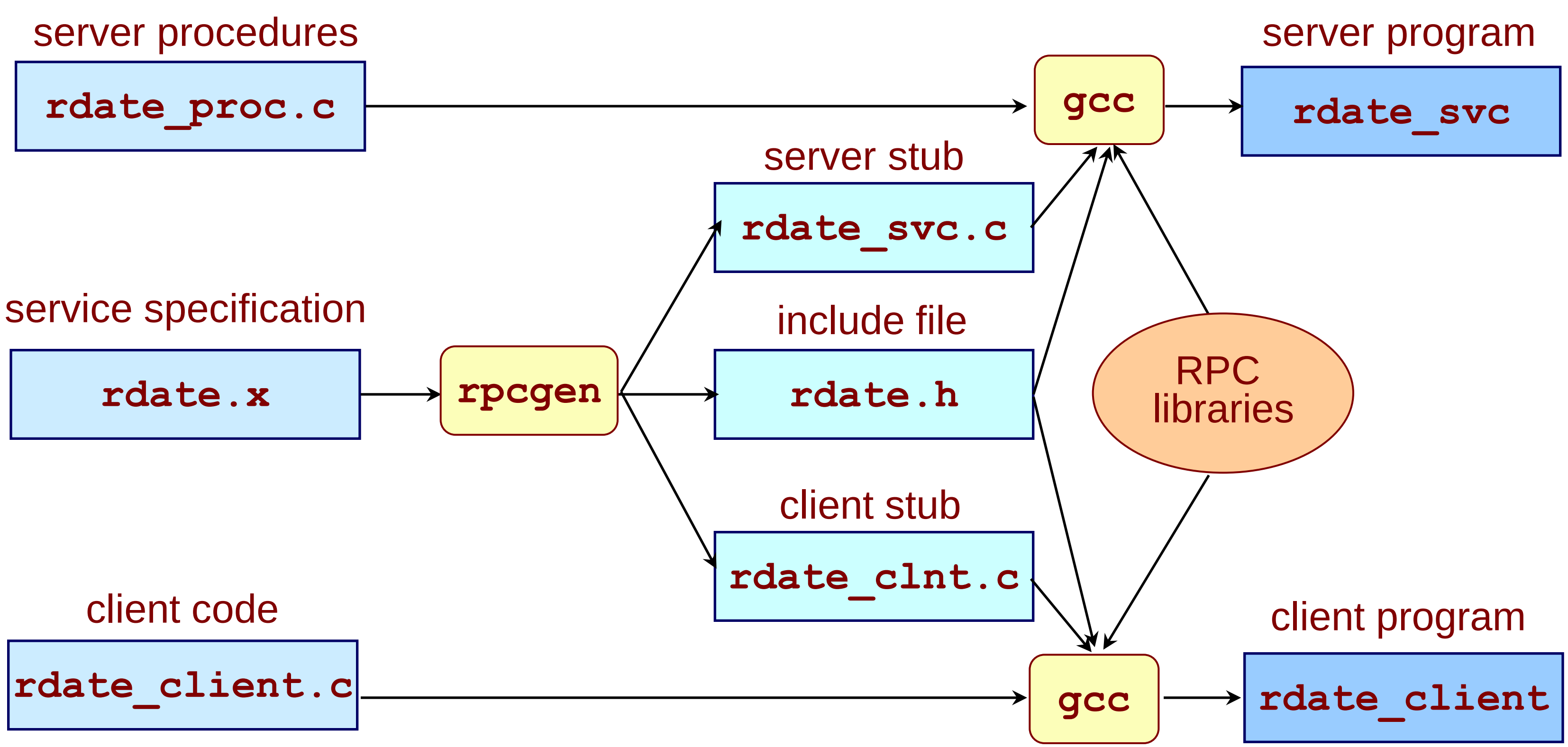
Sun RPC (ONC RPC)

- Sun Microsystems' RPC (also called Open Network Computing RPC, ONC RPC) is the *de facto* standard over the Internet
  - At the core of NFS, and many other services
  - Found in modern Unix systems (e.g., Linux)
- Data format specified by XDR (eXternal Data Representation)
  - Initially only for data representation, then extended in a proper IDL
- Transport can use either TCP or UDP
- Parameter passing:
  - Only pass by copy is allowed (no pointers)
  - Only one input and one output parameter
- Provision for DES security

DCE RPC

- The Distributed Computing Environment (DCE) is a set of specifications and a reference implementation
  - From Open Software Foundation (OSF), recently renamed as Open Group, no-profit standardization organization
- Several invocation semantics are offered
  - At most once, idempotent, broadcast
- Several services are provided on top of RPC:
  - Directory service
  - Distributed time service
  - Distributed file service
- Security is provided through Kerberos
- Microsoft's DCOM and .Net remoting are based on DCE
- Recently extended towards distributed objects

Sun RPC: Development cycle



rdate.x

```
program RDATE_PROG {
  version RDATE_VERS {
    long   BIN_DATE(void) = 1; /* procedure number */
    string STR_DATE(long) = 2; /* procedure number */
  } = 1;                       /* version number   */
} = 0x20000001;                /* program number   */
```

Politecnico di Milano

rdate_proc.c

```
#include <time.h>
#include "rdate.h"

long * bin_date_1_svc(void *argp, struct svc_req *reqp) {
  static long  result;
  result = (long) time(NULL);
  return &result;
}

char ** str_date_1_svc(long *argp, struct svc_req *reqp) {
  static char * result;
  result = ctime((time_t *) argp);
  return &result;
}
```

Politecnico
di Milano

rdate_client.c

```
#include <stdio.h>
#include "rdate.h"
int main (int argc, char *argv[]) {
  CLIENT *clnt; long  *t; char **str;
  if (argc < 2)
    { printf ("usage: %s server_host\n", argv[0]); exit (1); }
  clnt = clnt_create(argv[1], RDATE_PROG, RDATE_VERS, "udp");
  if (clnt == NULL)
    { clnt_pcreateerror(argv[1]); exit(1); }
  t = bin_date_1((void*)NULL, clnt);
  if (t == (long *) NULL)
    { clnt_perror(clnt, "call failed"); }
  str = str_date_1(t, clnt);
  if (str == (char **) NULL)
    { clnt_perror (clnt, "call failed"); }
  printf("Date at host %s is %s\n", argv[1], *str);
  clnt_destroy (clnt);
  exit (0);
}
```

Politecnico di Milano

Exercise

- Using Sun's RPC write a client/server program to read the first 100 bytes of a file located on a remote machine (knowing its full path)

Politecnico di Milano

Passing parameters by reference

- How to pass a parameter by reference?
  - Many languages do not provide a notion of reference, but only of pointer
  - A pointer is meaningful only within the address space of a process…
- Often, this feature is simply not supported (as in Sun’s solution)
- Otherwise, a possibility is to use call by value/result instead of call by reference
  - Semantics is different!
  - Works with arrays but not with arbitrary data structures
  - Optimizations are possible if input- or output-only

Politecnico di Milano

Contents

- Fundamental inter-process communication services
  - Unicast
    - Stream (TCP) and datagram (UDP) sockects in C and Java
  - Multicast
    - Multicast datagrams in Java
- Remote procedure call
  - Fundamentals
  - RPC in C (under Unix/Linux)
- **Remote method invocation**
  - **Fundamentals**
  - **Java RMI**

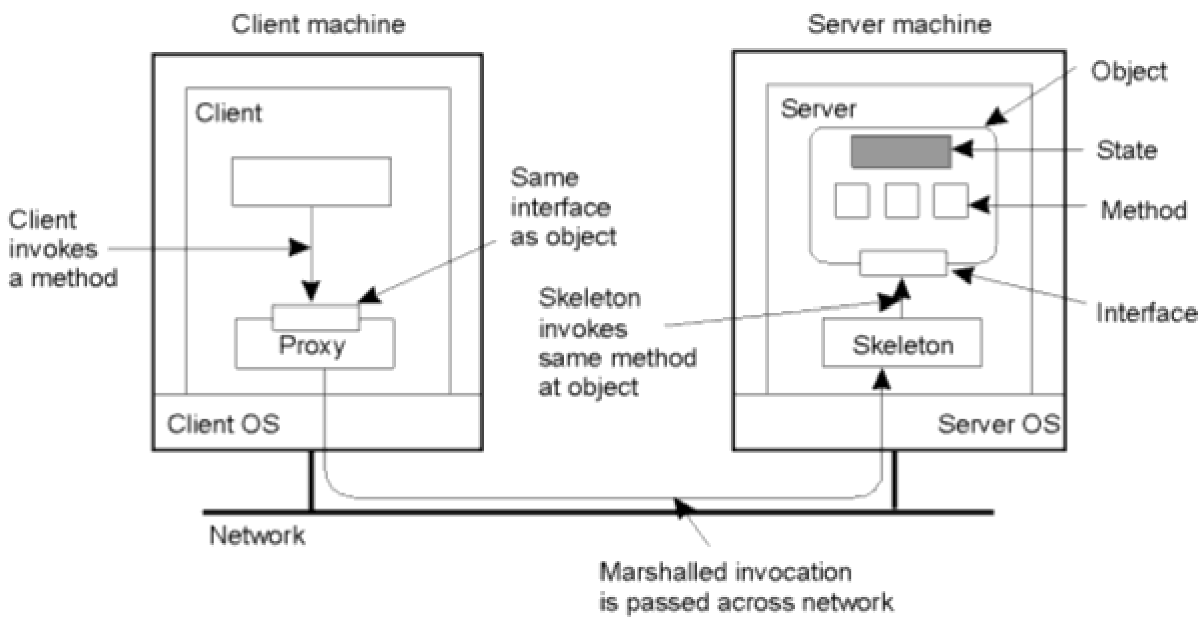
Remote method invocation

- Same idea as RPC, different programming constructs
  - The aim is to obtain the advantages of OOP also in the distributed setting
- Important difference: remote object *references* can be passed around
  - Need to maintain the aliasing relationship
- Shares many of the core concepts and mechanisms with RPC
  - Sometimes built on top of an RPC layer

Interface definition language

Politecnico
di Milano

- In RPC, the IDL separates the interface from the implementation
  - To handle platform/language heterogeneity
- Such separation is one of the basic OO principles
  - It becomes natural to place the object interface on one host, and the implementation on another
- The IDLs for distributed objects are much richer
  - Inheritance, exception handling, …



Java RMI

Politecnico di Milano

- The simplest among modern mainstream distributed object systems
  - Focuses only on remote method invocation
    - Kind of RPC for objects…
  - More advanced services provided by other components of the Java family (es., Jini, JNDI, …)
- Part of Java since version 1.1
- Innovative and/or relevant aspects:
  - Semantics of parameter passing
  - Class downloading
  - Dynamic proxies

Politecnico di Milano

Interfaces and remote objects

- The Java interface concept acquires a new role with distribution
  - Used as an IDL specification, without an IDL compiler
- A *remote object* (i.e., one whose methods can be invoked remotely) must implement an interface that extends `java.rmi.Remote`
  - And whose methods must throw the `java.rmi.RemoteException`
- Example:

```
import java.rmi.*;
public interface AccountServer extends Remote {
  Account getAccount(int num) throws RemoteException;
}
```

Account must be remote or serializable (more on this later)

Exporting remote objects

- Implementing a remote interface is not sufficient: to accept remote calls a remote object must be *exported*
  - Basically, to listen on a socket for incoming calls
- Done automatically in the constructor, if the remote object subclasses from `java.rmi.server.RemoteObject` (typically through `UnicastRemoteObject`)
- Alternately, done with the static method `UnicastRemoteObject.exportObject`
  - E.g., when there is a need to inherit from an application class
  - Returns the stub
- In any case, all the constructors of the remote object must throw a `RemoteException`
- The object can be *unexported*, too

Politecnico di Milano

Obtaining a remote reference

- There are essentially two ways:
  - Through parameter passing, as an input parameter or result value (just like in Java)
  - By expliciting querying a simple lookup service called `rmiregistry`
- In both cases, the remote reference is an object containing the client stub (proxy)
  - Implements the same interface as the remote object
  - Instance of `RemoteStub`
  - Proxies are serializable (i.e., can be passed around)
  - Possible because there is only one language …
- Once acquired, the remote reference is indistinguishable from a local one
  - The client can invoke any of the methods in the remote interface of the target object

Explicit reference lookup

- The default lookup service, `rmiregistry`, provides for dynamic service binding
  - Maintains the association between a symbolic name and an object bound on the server side
  - The service is not distributed
  - Implemented through a separate process
  - Due to security concerns it is not possible to bind an object on a registry executing on a different host
- RMI clients can:
  - Obtain a remote object reference (proxy), given the symbolic name
  - Ask for the list of available names
- RMI servers bind their remote objects (a proxy is stored in registry)
- The class `java.rmi.Naming` is used to interact with the registry
  - Main methods: `lookup, list, bind, rebind, unbind`
  - Other classes are available to create the registry from the application instead of command-line, and to enable alternative implementations

Example: The server

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class AccountServerImpl extends UnicastRemoteObject
 implements AccountServer {
 private Map<Integer,Account> accounts;
 …
 public static void main(String[] args) {
  …
  try {
   Naming.rebind("//localhost:"+ port + "/AccountServer",
                 new AccountServerImpl());
  } catch(java.net.MalformedURLException ex) { … }
    catch(RemoteException ex) { … }
 }

 public AccountServerImpl() throws RemoteException { … }

 public Account getAccount(int num) throws RemoteException {
    return accounts.get(num);
 }
}
```

defaults to 1099

Example: The client

Politecnico di Milano

```
import java.rmi.*;

public class AccountClient {
  public static void main(String args[]) {
    ...
    try {
      accountServer = (AccountServer) Naming.lookup("//" + host + ":" +
                      port +  "/AccountServer");
    } catch(java.net.MalformedURLException ex) { … }
      catch(NotBoundException ex) { … }
      catch(RemoteException ex) { … }
    ...
    Account a = null;
    try {
      a = accountServer.getAccount(current);
    } catch (RemoteException ex) { … }
    ...
  }
}
```

Apart from the try/catch block, the remote method invocation looks just like a local one

Parameter passing

- The semantics of invocation is different in the local and in the remote case
  - As a result of a precise design choice!
- Given a method invocation `m(obj)`:
  - If `obj` is a remote object, the usual by-reference semantics is preserved: if `m` modifies the state of `obj`, the latter is accessed directly through the network
  - Otherwise, `obj` is passed by copy (`obj` must be serializable); modifications in `m` are visible only on the copy and do not trigger network communication
- The same holds for methods with multiple input parameters, as well as for result values
- Trades ease of programming for communication efficiency
- Still unsatisfactory: the passing modality can be decided (to a large extent) only statically, and not per-invocation or per-instance

Exercise

- Implement, using RMI a `RemoteDateServer` similar to the one implemented using RPC
  - You can pass around longs (the `System.currentTimeMillis`) or `Date` objects
  - How are dates passed? By copy or by reference?
- Implement a client that access the `RemoteDateServer` above and measures the latency of the RMI call and the clock drifts between the client and the server

RMI and concurrency

Politecnico di Milano

- A frequently asked question is whether RMI performs a remote method invocation within a separate thread or not and, consequently, whether synchronization is necessary
- From the RMI specification (§ 3.2):
  "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 remote object invocations to threads. Since remote method invocations on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe."

Proxies, dispatchers, skeletons

- The `rmic` compiler WAS used to generate proxy, skeleton, and dispatcher for a remote object
  - It operates directly on the class file
- The proxy makes invocation transparent to clients
  - Handles connection and marshaling
  - Implements the same interface ➔ preserves type compatibility
  - *One proxy per process! (not per reference)*
- The dispatcher handles connections on the server side, and forwards a method invocation request to the skeleton
  - Deals with socket reuse and thread pooling
- The skeleton deals with marshalling and forwards the call to the actual server object
- However…
  - Since Java 2 skeletons are no longer needed
    - Uses reflection to transfer an actual `Method` instance
  - Since J2SE 5.0 stubs are no longer needed (no compilation through `rmic`)
    - Based on dynamic proxies, automatically generated classes implementing a set of interfaces specified at run-time

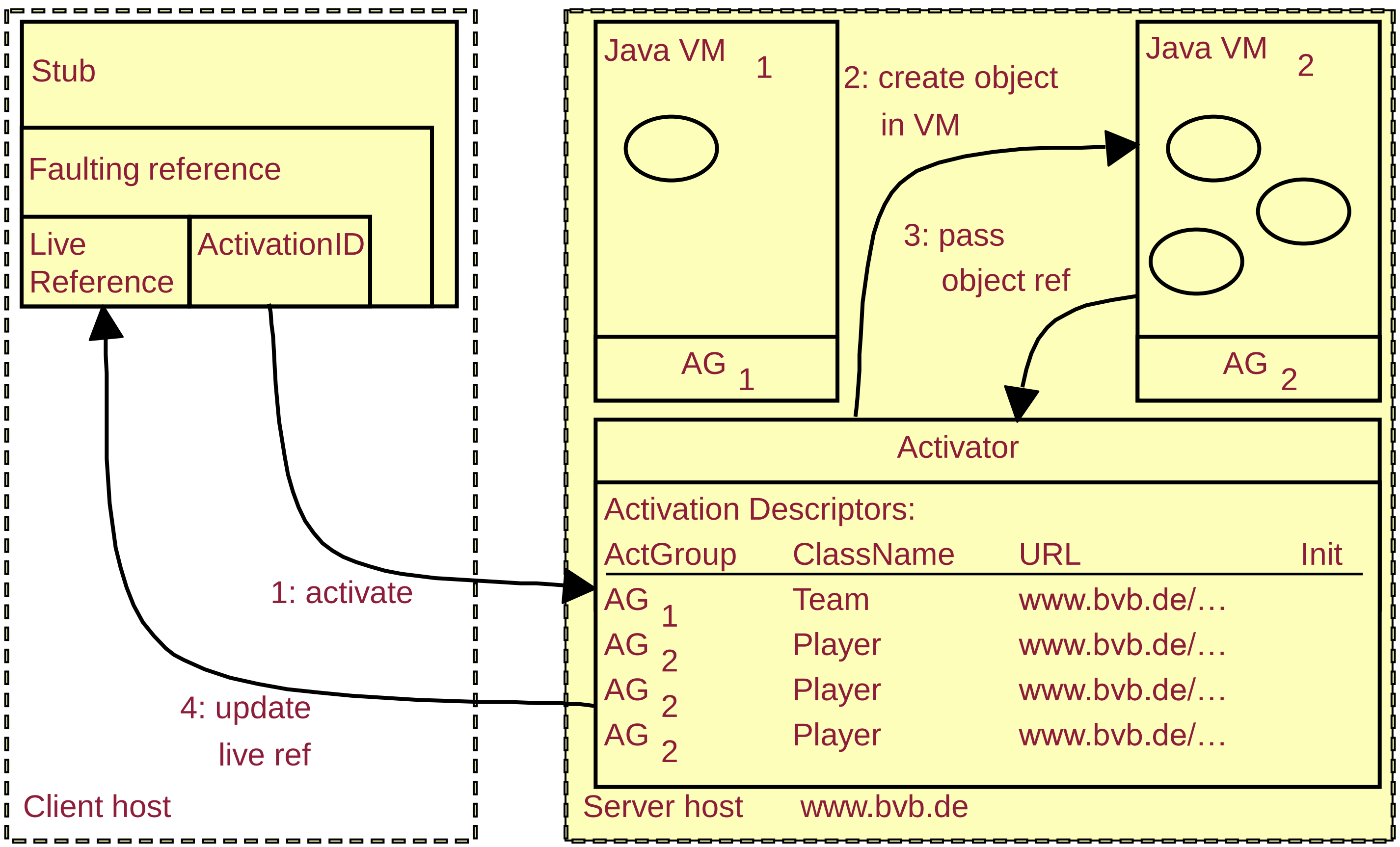
Exercise

- Reimplement the client/server chat using RMI

Dynamic object activation

- Remote objects can be created on demand, to save resources
  - Using the services provided by the classes in the `java.rmi.activation` package
- In practice, an activatable object must extend the `Activatable` class (a subtype of `RemoteObject`)
- An *activator* is responsible for the activation of remote objects
  - `rmid` is the default activator daemon
- All the activatable objects that are part of the same `ActivationGroup` are instantiated into the same JVM
  - At invocation time the activator checks if the requested `ActivationGroup` exists
  - If the `ActivationGroup` exists the activator delegates instantiation and invocation of the remote object to it
  - Otherwise the activator launches a new JVM, instantiates the `ActivationGroup` into the new JVM, and repeats the operations above

Dynamic object activation



Example: An activatable server

```
import java.util.*; import java.rmi.*; import java.rmi.activation.*;

public class AccountServerImpl1 extends Activatable implements AccountServer {
  private Map<Integer,Account> accounts;
  public static void main(String[] args) {
    ...
    try {
      ActivationGroupDesc myGroupDesc = new ActivationGroupDesc(null,null);
      ActivationGroupID agi = ActivationGroup.getSystem().registerGroup(myGroupDesc);
      ActivationDesc myDesc = new ActivationDesc(agi, “AccountServerImpl1",
                              "file:/d:/Home/Java/classes/", null);
      AccountServer server = (AccountServer) Activatable.register(myDesc);
      Naming.rebind(“AccountServer1", server);
    } catch(Exception ex) { ex.printStackTrace(); System.exit(-1); }
  }

  public AccountServerImpl1(ActivationID id, MarshalledObject data)
    throws RemoteException {
    super(id,0); // port 0 means first available port
  }

  public Account getAccount(int num) throws RemoteException {
    return accounts.get(num);
  }
}
```

Politecnico di Milano

Code downloading

- Problem: Consider an invocation `m(T a)`
  - We can expect that the bytecode for `T` is co-located with the remote object
  - But, due to polymorphism, the client may send a subtype of `T`
    - Always the case when `T` is an interface
  - Same problem regardless of whether `a` is a remote object or not
    - Finding the stub class vs. the actual class
- RMI is designed as an open system: cannot assume all possible types are preloaded everywhere
- Solution: applications can annotate types with codebase information
  - I.e., specify where to find the bytecode for a given type name
  - Typically a Web server
- Implemented by redefining the class loader and the serialization mechanism
- Simplifies deployment of stubs (in the old version of Java requiring stubs to be pre-generated)

Politecnico di Milano

Some considerations

- RMI does not fully mask distribution
  - Semantics of parameter passing
  - Remote exceptions
  - Remote interfaces
- This is a result of a precise choice, to distinguish local from remote interaction
  - J. Waldo, G. Wyant, A. Wollrath, S. Kendall. "A Note on Distributed Computing", 1994. Sun techrep republished in "Mobile Object Systems", Springer LNCS 1222, 1997.
- RMI is less powerful than other distributed object systems:
  - Meant to provide a fundamental building block instead of a complete solution
  - Other Java components build upon it (e.g., Jini, J2EE, …)