ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Network programming

Concurrency and distribution

The notions of concurrency and distribution are logically distinct:

- concurrency refers to time (parallel execution)
- distribution refers to space (remote execution)

In practice, programming distributed systems often requires to deal with concurrency too (remote location implies possible parallelism).

In this presentation, we focus on models of network communication that are central to distributed system programming.

From concurrent to distributed systems

	Multiprocessor	Multicomputer	Distributed system
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared excluding maybe disks	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

From: A. S. Tanenbaum, *Modern operating systems*, 3rd edition, 2009.

Models of distributed systems

There are many different models of distributed computing

- Client/server (e.g., TCP)
- Object-oriented middlewares (e.g., RMI, CORBA)
- Web services
- Document-based (e.g., the WWW)
- File-system based (e.g., NFS, Samba)
- Tuple spaces (e.g., Linda)
- Publish/subscribe (e.g., IBM Websphere MQ)
- Map/reduce (e.g., Hadoop)
- Grids

Challenges of network programming

- Heterogeneity
- Openness
- Security
- Scalability and load balancing
- Failure handling (partial failures)
- Concurrency
- Programming abstractions (transparency)

(list adapted from G. Cugola)

A distributed system is a system where I can't get my work done because a computer has failed that I've never even heard of.

-- Leslie Lamport

Outline

- Client/server programming with sockets
  - TCP
  - UDP
  - IP Multicast
- The Remote Procedure Call (RPC) model
  - Java's RMI
  - C# (and .NET)'s WCF: remote objects as web services

ETH Zürich
Chair of Software Engineering

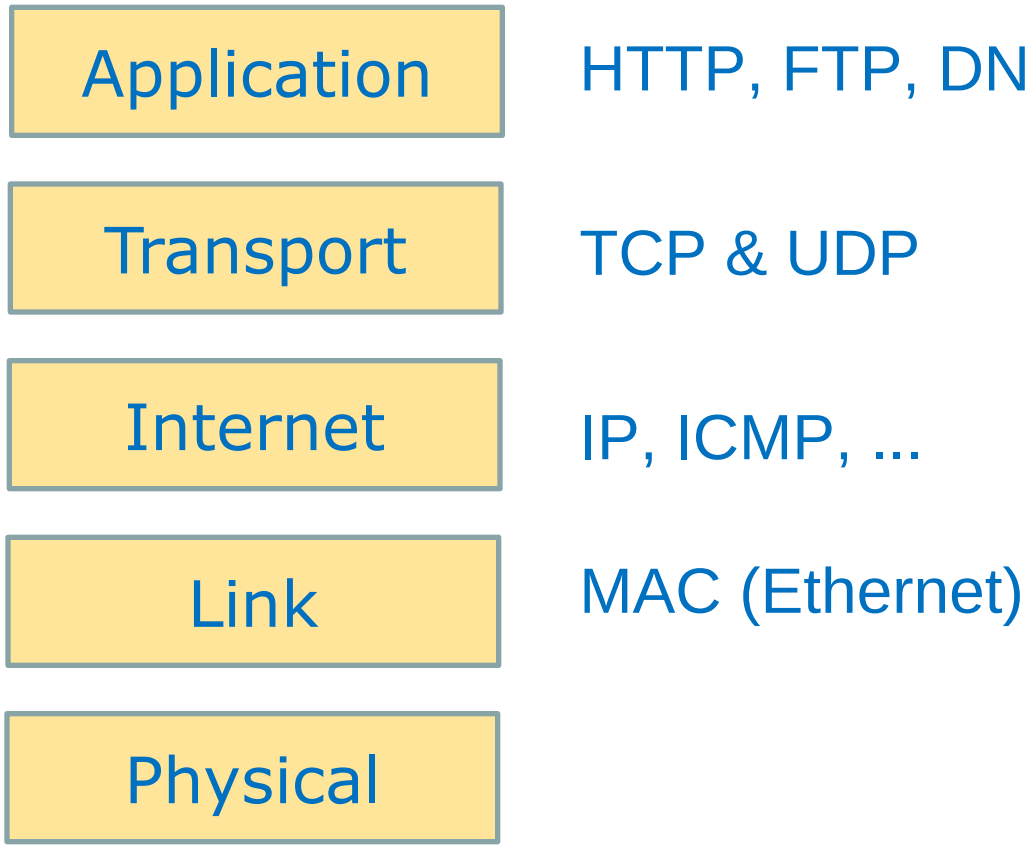
Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Client/server programming with TCP/IP sockets

The Internet protocol stack

The IP protocol belongs to a hierarchy of communication protocols commonly known as TCP/IP stack



TCP and UDP

- End-to-end communication between nodes
- Each node is identified by an IP address and a port number

TCP (Transmission Control Protocol) is the main connection-oriented (also: stream) internet protocol

- handshaking: a stable connection is established
- packet order is reconstructed
- reliable packet delivery (best effort)

UDP (User Datagram Protocol) is the main connectionless (also: datagram) internet protocol

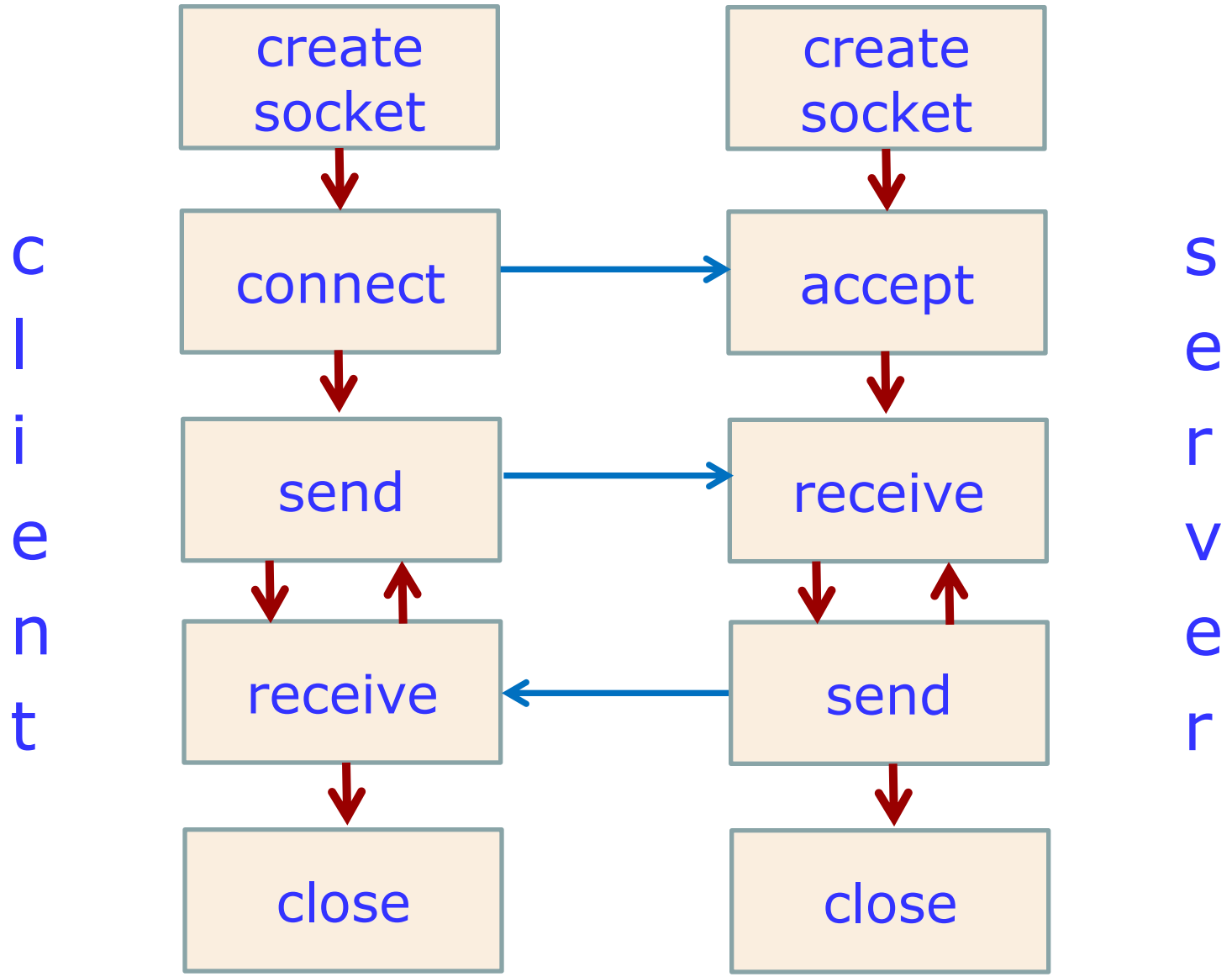
- stateless
- no acks, no retransmission
- unreliable but simple

Socket communication

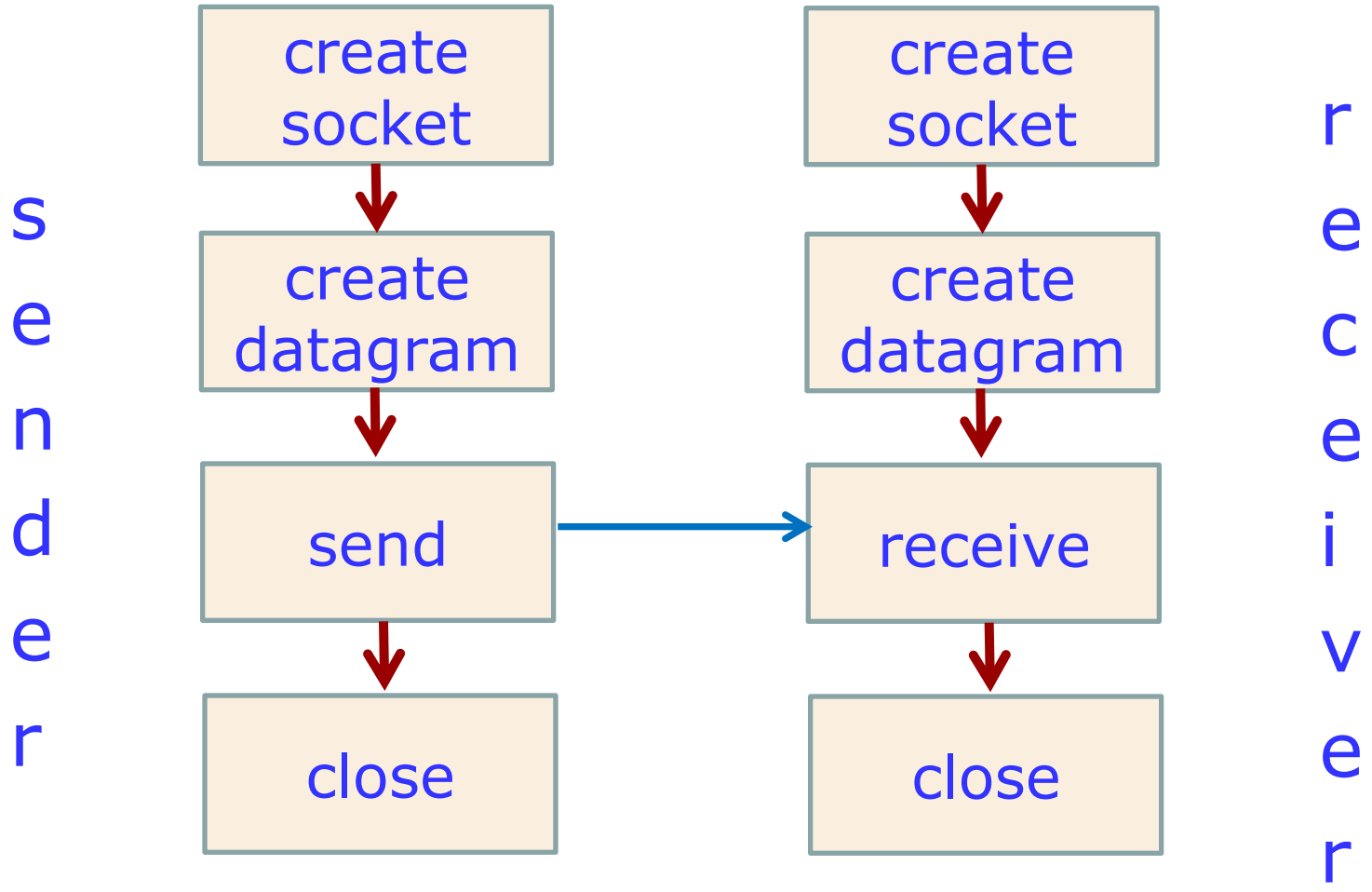
- Sockets are abstractions for distributed inter-process communication
- Each socket instance provides a logical end point for a communication flow
- The original socket API design dates back to Unix BSD (circa 1982)

Sockets can support different communication protocols.
We focus on Internet sockets, based on the IP protocol

Stream (TCP) socket communication



Datagram (UDP) socket communication



Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Sockets in Java

Stream sockets API in Java (java.net)

`class Socket`

- On client side: used to connect to server
- On server side: used to manage a connection with a single client

`class ServerSocket`

- Only on server side: used to listen to any number of client connection attempts

Main API of `class Socket`

- **`new Socket(String host, int port)`**
  Connect to host with name **`host`** (transparent name resolution) on port **`port`**

- **`InputStream getInputStream()`**
  Return a stream to read from the attached connection

- **`OutputStream getOutputStream()`**
  Return a stream to write to the attached connection

- **`void close()`**
  Close the attached connection (implicit with try with resources)

Main API of `class ServerSocket`

- **`new ServerSocket(int port)`**
  Create a socket that listens on port **`port`**

- **`Socket accept()`**
  Accept a connection (on the port where it's listening) and return a **`Socket`** object to communicate on that connection.
  (Block until a connection is established. You will use threading to implement non-blocking behavior.)

- **`void close()`**
  Dispose the server socket (implicit with try with resources)

TCP Client in Java: example (1/2)

```
String host = "localhost";
int port = 100;

try (
      // Create the socket and connect
      Socket s = new Socket(host, port);
      // Setup output stream to send data to server
      PrintWriter out = new
            PrintWriter(s.getOutputStream(), true);
      // Setup input stream to receive data from server
      BufferedReader in = new BufferedReader(new
                  InputStreamReader(s.getInputStream()));
   ) {
        System.out.println("Client: connected");
```

TCP Client in Java: example (2/2)

```
try ( /* See previous slide */ )
  {
    System.out.println("Client: connected");
    // Send a series of messages
    out.println("First message");
    out.println("Second message");
    out.println("Last message");
    // Receive one message
    String msg = in.readLine();
    if (msg != null)
      System.out.println("Client receives: " + msg);
  } catch (IOException e) {
      System.err.println("Connection problem");
  }
```

TCP Server in Java: example (1/2)

```
int port = 100;
try ( // Create the server socket
      ServerSocket servSock = new ServerSocket(port);
     ) {
      // Accept consecutive client connections
      while (true) {
            try ( // Accept a client
            Socket s = servSock.accept();
            // Output stream to send data to client
            PrintWriter out = new
               PrintWriter(s.getOutputStream(), true);
            // Input stream to receive data from client
            BufferedReader in = new BufferedReader(new
               InputStreamReader(s.getInputStream()));
            ) {
               System.out.println("Server: accepted");
```

TCP Server in Java: example (2/2)

```
try ( /* See previous slide */ ) {
      // Read three messages from client
      for (int i = 0; i < 3; i++) {
        String msg = in.readLine();
        System.out.println("Server received: " + msg);
      }
      // Send closing message to client
      out.println("All done.");
} catch (IOException e) {
      System.err.println("Accept problem");
  }
} // while (true)
} // outermost try (...)
 catch (IOException e) {
      System.err.println("Setup problem");
}
```

Datagram sockets API in Java (java.net)

`class DatagramSocket`

- To send datagrams to their recipients using UDP
- To receive datagrams using UDP while listening on a certain port

`class DatagramPacket`

- Wrap a message (payload) and its recipient (address & port)

Main API of `class DatagramSocket`

- **`new DatagramSocket()`**
  Setup a socket to send datagrams (binding on any port)

- **`new DatagramSocket(int port)`**
  Setup a socket to receive datagrams on port **`port`**

- **`void send(DatagramPacket p)`**
  Send datagram **`p`** through socket

- **`void receive(DatagramPacket p)`**
  Receive datagram through socket and store it in **`p`**.
  (Block until a datagram is received.)

- **`void close()`**
  Tear down socket

Main API of `class DatagramPacket`

- **`new DatagramPacket(byte[] b, int len)`**
  Setup a packet to receive datagrams of length **`len`**

- **`new DatagramPacket(byte[] b, int len, InetAddress a, int port)`**
  Setup a packet to send datagram with payload **`b`** of length **`len`** to address **`a`** on port **`port`**

- **`byte[] getData()`**
  Datagram payload as byte array
- **`int getLength()`**
  Datagram length
- **`int getOffset()`**
  Datagram offset

UDP sender in Java: example

```
InetAddress addr = InetAddress.getByName("localhost");
int port = 100;
String msg = "That's all folks."
// Create socket
DatagramSocket s = new DatagramSocket();
// Create datagram
DatagramPacket p = new
      DatagramPacket(   msg.getBytes(), msg.length(),
                        addr, port  );
// Send datagram through socket
s.send(p);
// The socket could be reused for other datagrams
s.close();
```

UDP receiver in Java: example

```
int port = 100;
// Create socket
DatagramSocket s = new DatagramSocket(port);
// Create datagram (to store received message)
DatagramPacket p = new DatagramPacket(new byte[17], 17);
// Receive datagram
s.receive(p);
// Convert payload to string
String msg = new
      String(p.getData(), p.getOffset(), p.getLength());
System.out.println("Received message: " + msg);
// The socket could reused for other datagrams
s.close();
```

ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Sockets in C#

Stream sockets API in C# (System.Net)

`class Socket`

- On client side: used to connect to server
- On server side: used to manage a connection with a single client
- More flexible than Java's: supports all kinds of sockets (also datagram and non-Internet)
- More complicated than Java's: more verbose client code, different alternative usages for the same functionality.

`class TcpListener`

- Only on server side: used to listen to any number of client connection attempts
- Encapsulate functionalities also available through **`Socket`**

Main API of `class Socket`

- `new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)`
  Initialize socket for IPv4 TCP stream communication.
- `void Connect(string host, int port)`
  Connect to host with name `host` (transparent name resolution) on port `port`
- `int Receive(byte[] buf)`
  Receive message in `buf` and return number of bytes received
- `void Send(byte[] buf)`
  Send message `buf` (encoded as byte array)
- `void Close()`
  Close the attached connection

Main API of `class TcpListener`

- **`new TcpListener(IPAddress addr, int port)`**
  Create a socket that listens at address **`addr`** on port **`port`**

- **`void Start()`**
  Begin listening on socket

- **`Socket AcceptSocket()`**
  Accept a connection (where it's listening) and return a **`Socket`** object to communicate on that connection. (Block until a connection is established. Use threading or **`Async`** method variants to have non-blocking behavior.)

- **`void Stop()`**
  Dispose the server socket.

TCP Client in C#: example (1/2)

```
string host = "localhost";
int port = 100;

try {
      // Create IPv4 stream TCP socket
      Socket s = new Socket(AddressFamily.InterNetwork,
                  SocketType.Stream, ProtocolType.Tcp);
      // Connect to server
      s.Connect(host, port);
      Console.WriteLine("Client: connected");
      // Prepare first payload
      byte[] msg = Encoding.ASCII.GetBytes(
                                    "First message");
```

TCP Client in C#: example (2/2)

```
    // Send first message
    s.Send(msg);
    // Omitted: prepare and send 2nd and 3rd messages
    ...
    // Receive one message
    byte[] buf = new byte[128];
    int recv = s.Receive(buf);
    if (recv > 0)
      Console.WriteLine("Client receives: " +
                        Encoding.ASCII.GetString(buf));
    // Dispose socket (better: in finally)
    s.Close();
} catch (SocketException) {
    Console.Error.WriteLine("Connection problem");
}
```

TCP Server in C#: example (1/2)

```
int port = 100;
IPAddress addr = IPAddress.Parse("127.0.0.1");

try {
    // Create TCP listener socket
    TcpListener servSock = new TcpListener(addr, port);
    // Start listening
    servSock.Start()
    // Accept consecutive client connections
    while (true) {
        try { // Accept a client
                Socket s = servSock.AcceptSocket();
```

TCP Server in C#: example (2/2)

```
        // Read three messages from client
        for (int i = 0; i < 3; i++) {
          byte[] buf = new byte[128];
          int recv = s.Receive(buf);
          Console.WriteLine("Server received: " +
                            Encoding.ASCII.GetString(buf));
        }
        // Send closing message to client
        s.Send(Encoding.ASCII.GetBytes("All done."));
} catch (SocketException) {
        Console.Error.WriteLine("Accept problem");
  }
} // while (true)
} // outermost try (...)
// Omitted: dispose resources, catch exceptions
...
```

Datagram sockets API in C# (System.Net)

`class Socket`

- To send datagrams to their recipients using UDP
- To receive datagrams using UDP while listening on a certain port

`class UdpClient`

- Only on receiver side: used to listen to and receive datagrams
- Encapsulate functionalities also available through **`Socket`**

`class IPEndPoint`

- Encapsulate IP address and port information
- Used to initialize **`UdpClient`**

API of `class Socket` for UDP

- **`new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp)`**
  Initialize socket for IPv4 UDP datagram communication.

- **`void SendTo(byte[] buf, IPEndPoint ep)`**
  Send message **`buf`** (encoded as byte array) to remote **`ep`**.

Main API of `class UdpClient`

- **`new UdpClient(int port)`**
  Create a socket that listens on port **`port`**

- **`byte[] Receive(ref IPEndPoint ep)`**
  Receive datagram through socket and return it.
  Store in **`ep`** information about the remote sender.
  (Block until a datagram is received.)

- **`void Close()`**
  Tear down listener

Main API of `class IPEndPoint`

- `new IPEndPoint(IPAddress.Any, 0);`
  Setup a placeholder endpoint (“any” address), to be overwritten when receiving datagrams.

- `new IPEndPoint(IPAddress addr, int port);`
  Setup an endpoint corresponding to **`addr`** and **`port`**, to send datagrams to.

To get an **`IPAddress`** from symbolic host name:

```
Dns.GetHostEntry(host).AddressList[0];
```

Note that **`GetHostEntry`** returns in general several IP addresses for a host; here we pick the first one.

UDP sender in C#: example

```
IPAddress addr =
        Dns.GetHostEntry("localhost").AddressList[0];
int port = 100;
byte[] buf =
        Encoding.ASCII.GetBytes("That's all folks.");
// Create socket
Socket s = new Socket(AddressFamily.InterNetwork,
                      SocketType.Dgram, ProtocolType.Udp);
// Create destination endpoint
IPEndPoint endpoint = new IPEndPoint(addr, port);
// Send datagram through socket to endpoint
s.SendTo(buf, endpoint);
// The socket could be reused for other datagrams
s.Close();
```

UDP receiver in C#: example

```
int port = 100;
// Create UDP listener socket
UdpClient s = new UdpClient(port);
// Create placeholder endpoint (to store sender info)
IPEndPoint ep = new IPEndPoint(IPAddress.Any, 0);
// Receive datagram
byte[] buf = s.Receive(ref ep);
// Convert payload to string
string msg =
         Encoding.ASCII.GetString(buf, 0, buf.Length);
Console.WriteLine("Received message: " + msg);
// The socket could reused for other datagrams
s.Close();
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

The Remote Procedure Call (RPC) Model

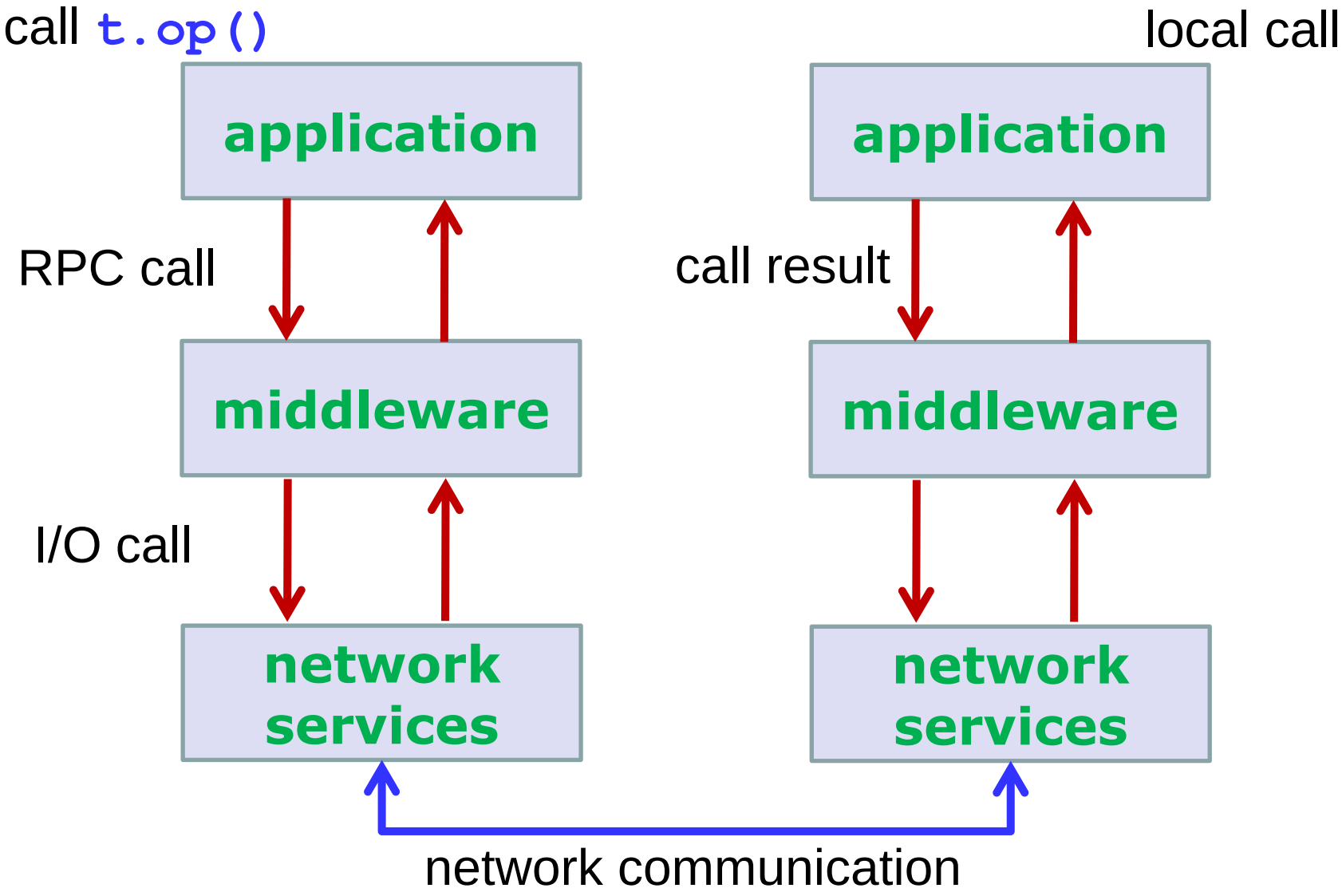
Remote Procedure Call

Client/server communication (using TCP or other protocols) takes place at a lower level of abstraction than application programming. Thus, programming applications with a lot of communication is complex, as it requires to deal with I/O primitives directly.

The Remote Procedure Call (RPC) mechanism raises the level of abstraction by transparently supporting procedure calls that are executed on remote processes. Thus, application programmers write usual code, whereas the RPC infrastructures takes care of distribution.

First implementation for Sun Unix in the early 1980s. That model is still widely used over the Internet (ONC RPC).

RPC Communication



RPC: interfaces and abstraction

An interface between the application code and the operations that can be invoked remotely ensures that the RPC mechanism is transparent and doesn't break the abstraction.

- An Interface Definition Language (IDL) is used to define an RPC interface: signatures of available operations
- Stubs for every node are automatically generated from IDL definitions:
  - map IDL onto the application language
- Different nodes may even use different runtimes and languages, as long as the middleware takes care of conversions
  - handle serialization and marshaling (data representation conversion) to make parameter passing transparent

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

RPC in Java: Remote Method Invocation (RMI)

Remote Method Invocation in Java

Remote Method Invocation (RMI) is Java's standard mechanism for RPC

- Limited to communication between applications written in Java or other languages that run on the JVM
- Since RMI is essentially mono-language, its IDL reuses Java's **`interface`** mechanism
- Addresses specific requirements of the object-oriented model (semantics of argument passing, object creation, polymorphism)

RMI offers a limited set of basic functionalities. The Java framework offers more advanced services (built on top of RMI) in other components such as Jini.

RMI interfaces and implementations

A remote object is an object that is accessible (whose methods are callable) remotely.

Remote objects are:

- instances of classes implementing an **`interface`** that extends **`java.rmi.Remote`** and
- exported (i.e., registered with the middleware) by calling **`UnicastRemoteObject.exportObject`**
  - exporting is implicit if the remote object's class extends **`java.rmi.server.UnicastRemoteObject`**

Methods of remote objects:

- may throw **`java.rmi.RemoteException`**
- have arguments of serializable or remote class types

RMI interfaces: example

```
import java.rmi.*;

public interface RCellInterface extends Remote {
      // Operations
      public void setVal(int val)
                  throws RemoteException;
      public int getVal()
                  throws RemoteException;
            // argument of remote class type
      public void setOther(RCellInterface rc)
                  throws RemoteException;
}
```

RMI implementation: example

```
public class RCell
 extends UnicastRemoteObject implements RCellInterface {

      public RCell() throws RemoteException {
            setVal(1);
            // implicitly exported because
            // extending UnicastRemoteObject
      }

      // omitted setVal and getVal: setter and getter

      // Set rc to -value of this
      public void setOther(RCellInterface rc)
                        throws RemoteException {
            rc.setVal(-getVal());
      }
}
```

RMI remote objects, references, proxies

Remote objects are accessed in Java applications through remote references (references pointing to remote objects)

Remote references are passed around just like local references (passed as arguments, returned as results)

The application code cannot distinguish between references to local and to remote objects

- with the exception of argument passing: see later

In the runtime environment, remote references point to proxy objects for the remote objects

- proxies are generated automatically as instances of **`RemoteStub`**

The RMI registry

Using the **`rmiregistry`** service, RMI server applications can offer remote references to clients

- The registry maps symbolic names to remote objects hosted by the server
- The registry runs local to the server, and only server-side applications can register objects in it

Clients can query the registry and obtain remote references through **`class java.rmi.Naming`**

- **`Remote lookup(String name)`**
  Return a remote reference for symbolic name **`name`**
- **`String[] list(String name)`**
  List all symbolic names registered on registry **`name`**
- **`void bind(String name, Remote obj)`**
  Register remote object **`obj`** under symbolic **`name`**

RMI server with registry: example

A server that registers a remote object of class **`RCell`** under symbolic name **`RCinst`**:

```
try {
      // Start registry
      // (alternatively from command line)
      LocateRegistry.createRegistry(port);
      // Create instance of RCell
      RCellInterface rc = new RCell();
      // Register rc under name "RCinst"
      Naming.rebind("//localhost:" + port +
                                    "/RCinst", rc);
}
  catch (RemoteException e) { ... }
```

RMI argument passing semantics

In a remote call `o.m(a)`, where `o` is a remote object:

- if `a` is a reference to a remote object, the actual argument is passed by reference (usual Java semantics)
- if `a` is a reference to a local (plain Java) object, the actual argument is passed by copy
  - the object pointed by `a` is deep-copied as in serialization
  - `m` works on the copy only, the original object is unchanged

The same applies to returned values (local objects are copied)

Advantages: simpler implementation and less communication

Disadvantages: breaks abstraction (distribution is not completely hidden), still not very flexible

RMI client application: example

A client that gets a remote object under symbolic name **`RCinst`** and calls **`setOther()`** on it.

```
try {
  // Get access to registry
  LocateRegistry.getRegistry(host, port);
  // Query for remote instance of RCell
  RCellInterface rc = (RCellInterface)
            Naming.lookup("//" + host + ":" + port +
                                        "/RCinst");
  // Remote call with remote argument
  rc.setOther(rc); // Just like a local call!
  // If rc stored 1 before the call, it now stores...
  ...
```

Remote call with local argument: example

```
public interface RCellInterface extends Remote {
      // argument of serializable class type
      public void setOther(LCell c)
                  throws RemoteException;
      // LCell is like RCell but without remoting
}
```

In the client:

```
LCell c = new LCell(); c.setVal(2);
// Remote call with local argument
rc.setOther(c); // Argument c passed by copy!
// c stored 2 before the call, and it now stores...
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

RPC in C#: Services of Windows Communication Foundation (WCF)

Remote objects in C# using services

The Windows Communication Foundation (WCF) is .NET's framework for distribution.

WCF follows the (web) service model of distribution.

Web services are:

- A software system designed to support interoperable machine to machine interaction over a network -- W3C
- An API accessible over a network, executed on a remote system hosting the requested service

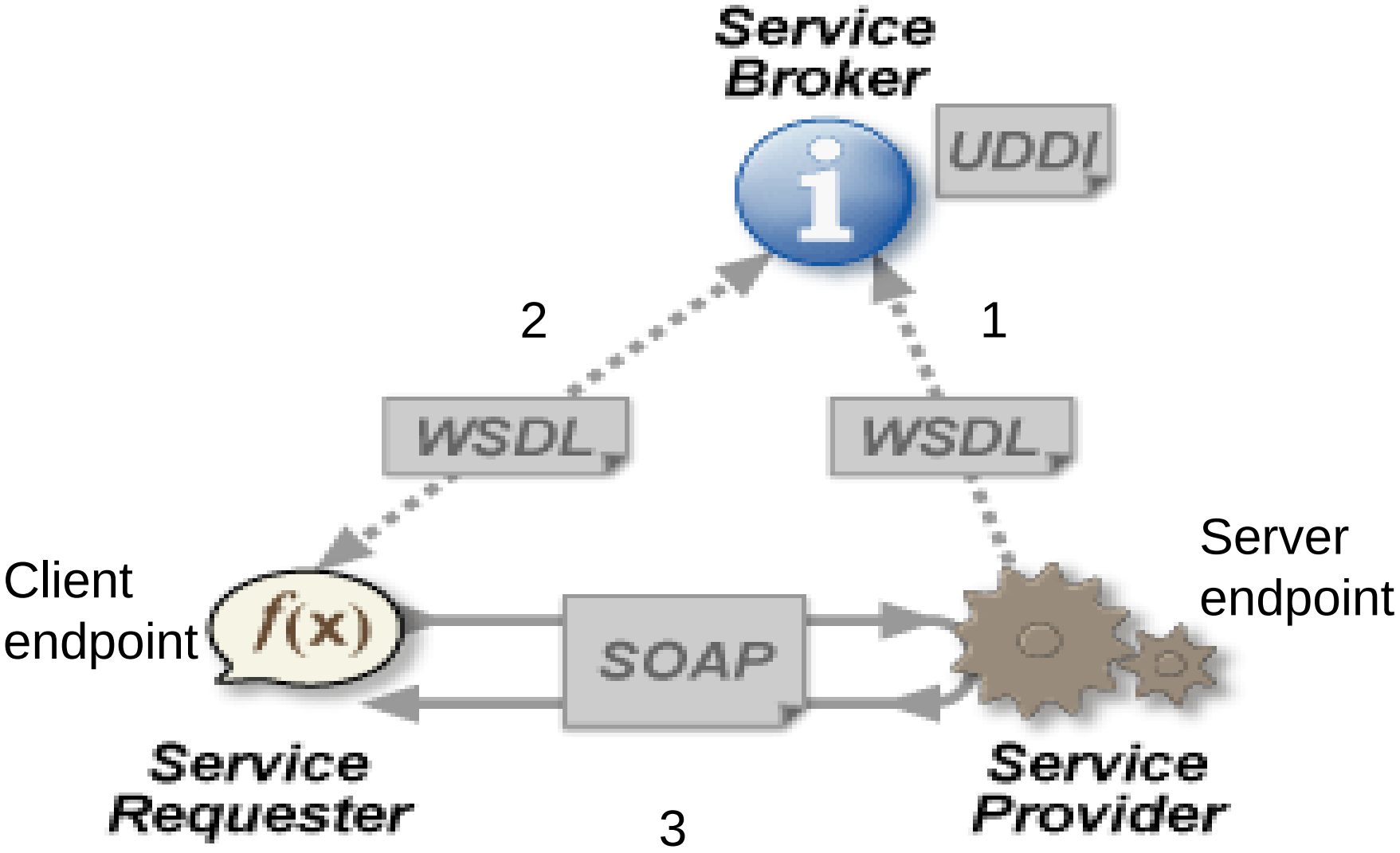
Services and distributed objects

Concretely, web services provide features similar to distributed objects and middleware, with some important differences:

- they target heterogeneous languages and platforms, and high decoupling
- they rely on a specific set of standard protocols
  - WSDL, UDDI, SOAP
- they support higher-level models than RPC
  - SOA and RESTFUL

We're now presenting the essential WCF API for services that provides similar functionalities as RPC

A high-level view of service technology



Source: http://en.wikipedia.org/wiki/Web_services

Fundamental concepts of WCF

Endpoint: access point where services are made available.

Each endpoint is identified by:

- an address (typically a URL), which identifies the access point
- a binding, which specifies the underlying communication protocol to be used to access the service

A specific service available at a endpoint is further identified by:

- a contract, which defines the interface of the service (and possibly other elements of its specification, such as functional or performance guarantees)
- typically distinct data contract and operation contract (→ arguments passed by copy)
- In C# defined using attributes and **`interface`**

Data contract: example

An abstract cell that consists of an integer value.

```
using System.Runtime.Serialization;

[KnownType(typeof(LCell))]
[DataContract]
public abstract class AbstractCell {
    [DataMember]
    public abstract int Val { get; set; }
}
```

Data contract implementation

A concrete implementation of the data contracts of **AbstractCell**

```
[DataContract]
public class LCell : AbstractCell {
    protected int val;

    public LCell() { val = 7; }

    public override int Val {
        get { return val; }
        set { val = value; }
    }
}
```

Operation contract: example

The interface of operations available as a service.

```
using System.ServiceModel;

[ServiceContract]
public interface IRCell {
    [OperationContract] void setVal(int val);
    [OperationContract] int getVal();
    [OperationContract] void setOther(AbstractCell rc);
}
```

Note: we have two implementations (setters/getters and properties) of very similar functionality. This is only for direct comparison with the RMI example. In practice, you'd probably keep one implementation only.

Service implementation: example

A concrete implementation of the operations of **`IRCell`** as well as the data of **`AbstractCell`**

```
[Serializable()]
public class RCell : AbstractCell, IRCell {

  public RCell() { Val = 0; }
  public override int Val { get; set; }
  public int getVal() { return Val; }
  public void setVal(int val) { Val = val; }
  public void setOther(AbstractCell rc) {
     rc.Val = this.getVal();
  }
}
```

WCF API for addresses & bindings

Bindings are defined in **`class`** **`System.ServiceModels.Binding`** and descendants

- For example, **`BasicHttpBinding()`** uses the HTTP protocol for communication

Addresses are defined using **`class System.Uri`**

- **`public Uri(string s)`** specifies a URI as a string such as **`http://localhost:8000`**

WCF API for service hosting

Services are instantiated using concrete descendants of **`class System.ServiceModels.ServiceHostBase`**

- **`public ServiceHost(Type t, Uri a)`** initializes host for service type **`t`** at address **`a`**
- **`ServiceEndpoint AddServiceEndpoint(Type t, Binding b, string n)`** creates and returns endpoint for service with contract specified as type **`t`**, using binding **`b`** and identifier **`n`**

WCF server: example

A server that hosts a service of class **`RCell`** under symbolic name **`RCinst`**:

```
try {
  BasicHttpBinding binding = new BasicHttpBinding();
  Uri url = new Uri("http://localhost:8000");
  // Create service for RCell objects at "url"
  service = new ServiceHost(typeof(RCell), url);
  // Create and register an instance of RCell
  // under name "RCinst" using contract of IRCell
  service.AddServiceEndpoint(typeof(IRCell),
                             binding, "RCinst");
  // Put service online
  service.Open();
}
...
```

WCF API for service clients

Channels to access service endpoints are created using **`class System.ServiceModels.ChannelFactory<>`** and descendants

- **`public ChannelFactory<T>(Binding b, EndpointAddress a)`** created factory for channels connecting to endpoint with address **`a`** using binding **`b`**
- **`IChannel createChannel()`** returns channel, which is then used as a local reference of class **`T`**
- **`void Close()`** tears down a factory (or closes a channel)

WCF service client: example

A client that gets a remote object under symbolic name **`RCinst`** and calls **`setOther()`** on it.

```
BasicHttpBinding binding = new BasicHttpBinding();
// Channels for service IRCell under name RCinst
var factory = new ChannelFactory<IRCell>(binding,
            new EndpointAddress(
                  "http://localhost:8080/RCinst"));
// Get reference to remote service
IRCell rc = factory.CreateChannel();
var lrc = new LCell();   lrc.Val = 4;
// Remote call (data is passed by copy!)
rc.setOther(lrc);
// lrc stored 4 before the call, and it now stores...
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Wrap up

How to choose a networking model?

- What kind of data does the application work on?
- What's the abstraction level of the application using networking?
  - position in the communication stack
- What's the scale of the application using networking?
  - typical umber of nodes
- Is the application multi-platform?
- How reliable is the network?
- How important is performance?
  - beware of premature optimization