Slides for Chapter 4: Interprocess Communication

From Coulouris, Dollimore, Kindberg and Blair
Distributed Systems: Concepts and Design
Edition 5, © Addison-Wesley 2012

Text extensions to slides © David E. Bakken, 2012-2020
Next 2 Chapters: Communication Aspects of MW

• Chap 4: IPC
  • API for Internet protocols
  • External data representation and marshalling
  • Multicast communications
  • Network virtualization: overlay networks
  • Case Study: MPI

• Chapter 5: Remote invocation paradigms
  • Request-reply protocols
  • Remote procedure call
  • Remove method invocation
  • Case study: Java RMI
API for Internet Protocols [4.2]

4.2.1: How to implement send/receive of Sec 2.3.2
4.2.2: Sockets
4.2.3: UDP in Java
4.2.4: TCP in Java
Figure 4.1
Middleware layers

- APIs for Internet protocols
  - UDP: **message passing** abstraction (incl. multicast)
  - TCP: **stream processing**
Characteristics of IPC [4.2.1]

• Message passing supported by send+receive to endpoints ("message destination" in text)

• **Synchronous** and **asynchronous** communication
  
  • Associate a queue with each endpoint
    
    • Senders add msg to remote queue; Receivers remote msg from local queue
    
    • Send and receive both have synch. (blocking) and asynch. (non-blocking)
    
    • Non-blocking receive not always supported cause more complex for app

• Can combine in nice ways
  
  • SR Language from U. Arizona 1990s, [http://www.cs.arizona.edu/sr/](http://www.cs.arizona.edu/sr/)

<table>
<thead>
<tr>
<th></th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocking wait</td>
</tr>
<tr>
<td><strong>Sender</strong></td>
<td></td>
</tr>
<tr>
<td>Asynch call</td>
<td>Asynch. Msg passing</td>
</tr>
<tr>
<td>(non-blocking)</td>
<td></td>
</tr>
<tr>
<td>Synch call</td>
<td>Rendezvous</td>
</tr>
<tr>
<td>(blocks)</td>
<td></td>
</tr>
</tbody>
</table>

© 2012 David E. Bakken
Characteristics of IPC (cont.)

• Message destinations
  • Endpoint on internet is (IP address, (remote) port #)
  • Local port created for both sides to access
  • One receiving process per port
  • Process can use multiple ports to receive messages
  • Transparency support: look up endpoint from name (Sec 5.4.2)

• Reliability
  • Reliable: msg delivered despite “reasonable” packets lost
  • Integrity support: msgs must be uncorrupted and no dups
    • More “above” too to verify sender etc

• Ordering
  • Some apps require sender order (FIFO)
  • Layer above UDP (if multicast), or with TCP
Figure 4.2
Sockets and ports

- **Socket abstraction**: hide steps in lower-level protocols etc
  - To receive: socket bound to local port
  - Can use same socket to both send and receive
  - “large number” (!!) of ports, $2^{16}$
  - Anyone can send to a socket if know endpoint
  - Can not have multiple receivers (except for multicast ..)
Java API for IP Addresses

- **Class InetAddress (both IPv4 and IPv6)**
- `InetAddress aComputer = InetAddress.getByName("foo.bar.edu")`
- Can throw `UnknownHostException`
- **Can throw** `UnknownHostException`
UDP datagram communication [4.2.3]

- Sent without ACKs or retries

- **Issues**
  - Message size: receiver gives fixed-sized buffer, if msg big trunc.
  - Blocking: receive blocks (unless timeout set), send rtns quickly
    - Receiver can get msg later if not blocked (queued)
  - Timeouts: can set, choosing really hard
  - Receive from any: no origin (sender) specified, endpoint in header

- **Failure model:** omissions, ordering

- **Uses of UDP**
  - DNS
  - VOIP and other video/audio
  - Higher-level multicast with stronger properties
Java API for UDP datagrams

- **DatagramPacket class**
  - Sending constructor takes array of bytes, length, endpoint
  - Another constructor for receiving msgs: array of bytes, length
  - `DatagramPacket.getData`: receiver gets buffer

- **DatagramSocket class**
  - Constructor: port (also no-port: choose any free local port)
  - `Send()` and `receive()`
  - `setSoTimeout`: for receive, if times out throws `InterruptedIOException`
  - `Connect`: connect to remote endpoint
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket(m, m.length(), aHost, serverPort);
            aSocket.send(request); // send message to the remote endpoint
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply); // blocking wait for reply
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException e) {System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {System.out.println("IO: " + e.getMessage());
        } finally {if(aSocket != null) aSocket.close();}
    }
}
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){  // typical infinite server waiting loop: get request, send reply
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: "+ e.getMessage());
        }catch (IOException e) {System.out.println("IO: "+ e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
}
TCP stream communication [4.2.4]

• Hides many details behind socket abstraction
  • Message sizes: sender chooses how much data to read or write
    • Underlying impl decides when to send packets
    • Can flush/synch to force a send (why need?)

• Lost packets
• Flow control
• Packet duplication and ordering
• Message destinations (once socket set up)
  • Server creates listening socket
  • Client calls connect(...)
  • Server calls accept(...)

© 2012 David E. Bakken
TCP stream communication (cont)

• API
  • Assumes for setup one side client other server
  • After that bidirectional with no distinction (both input and output stream in socket)
  • Listening socket maintains queue of connect requests

• Issues with TCP (and stream communication)
  • Matching data items: need to agree on data types (UDP too)
  • Blocking: like UDP
  • Threads: server usually forks new process for each client (why?)
TCP stream communication (cont)

• Failure model
  • Checksums detects and rejects corrupted packets
  • Sequence numbers: detect and reject duplicate packets
  • If too many lost packets, socket declared to be closed
    • I.e., not (very) reliable communication
  • Uses of TCP (lots): HTTP, FTP, Telnet, SMTP
Java API for TCP streams

• **ServerSocket** class: for listening to connect requests
  • Method **accept** gets connect request or blocks if none queued
    • Returns a **Socket** object to communication with the client

• **Socket**: for pair to communicate with
  • Client uses constructor to create a given endpoint
  • Can throw **UnknownHostException** or **IOException**
  • **getInputStream** and **getOutputStream** to access streams
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream());
            DataOutputStream out =
                new DataOutputStream(s.getOutputStream());
            out.writeUTF(args[0]);        // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
    }finally {if(s!=null) try {s.close();}catch (IOException e){System.out.println("close:"+e.getMessage());}}
}
Figure 4.6
TCP server makes a connection for each client and then echoes the client’s request

```java
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch (IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}
```

// this figure continues on the next slide
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out = new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}
        }
    public void run()
    {
        try {
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());
        } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}
        }
    }
}
External data representation and marshalling [4.3]

• Procedures/methods called with params, msgs take buffers
• Marshalling does this translation, unmarshalling reverses
• External data representations describe how
  • Endian-ness, ASCII or Unicode text, etc
• Two main techniques
  • Neutral format
  • Sender’s format (“receiver makes right”)
Approaches to marshalling and external data representation

1. CORBA’s common data rep. (CDR): structs, primitives
2. Java serialization: object or tree of objects
3. XML: textual description of data

• Comparisons
  • CDR, Java: middleware layer, XML more for hand coding in app
  • CDR, Java: binary form, XML text
  • CDR: only values (sort of), Java, XML: type info
  • XML larger, more error prone than automatic marshalling by middleware compiler

• Other possibilities (more “lightweight”)
  • Google protocol buffers (20.4.1): describe stored & transmitted data
  • JavaScript Object Notation (JSON)
CORBA’s Common Data Representation (CDR) [4.3.1]

- All 15 primitive types: short, long, boolean, ... any
  - Defn’s for both big- and little-endian (sent in sender’s order; tag)
  - Other types straightforward: IEEE floats, chars agreed between client and server
- Constructed/composite types (Fig 4.7, next)
  - Primitive types that make them up added in a byte sequence in a given order
- Marshalling generated automatically from IDL
Figure 4.7
CORBA CDR for constructed types

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also can have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>
The flattened form represents a `Person` struct with value: `{‘Smith’, ‘London’, 1984}`
Java Object Serialization [4.3.2]

• Java class equivalent to CORBA Person struct:

```java
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;

    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }

    // followed by methods for accessing the instance vars
}
```

© 2012 David E. Bakken
Java Serialization (cont.)

- **Handles**: serialized references to other objects
- Reflection: used by serialization to find class name of object to be serialized, when deserialized to create class
- (Read rest of details in text, not covering)
### Figure 4.9
Indication of Java serialized form

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>h0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String name:</td>
</tr>
<tr>
<td>1984</td>
<td>5 Smith</td>
<td>6 London</td>
</tr>
</tbody>
</table>

**Explanation**

- **Class name, version number**
- **Number, type and name of instance variables**
- **Values of instance variables**

The true serialized form contains additional type markers; h0 and h1 are handles.
Extensible Markup Language (XML) [4.3.3]

• Markup language: encodes both text and its structure
  • HTML: for web pages
  • XML: for structured documents for web
  • Both: derived from (very complex) SGML

• XML namespaces: provided for scoping names (avoid name collisions)

• XML schemas: define elements and attributes for a doc, nesting, order and number of elements, etc

• Overview in this lecture, read details in text
  • Gory details not testable (don’t memorize minutia)
  • But should really have intuition into difference from CORBA and Java: purpose, why design decisions made, efficiency, readability, other comparisons
<person id="123456789">
  <name>Smith</name>
  <place>London</place>
  <year>1984</year>
  <!-- a comment -->
</person>
Figure 4.11 Illustration of the use of a namespace in the Person structure

```xml
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place>
    <pers:year> 1984 </pers:year>
</person>
```
<xsd:schema xmlns:xsd = "URL of XML schema definitions">
  <xsd:element name = "person" type = "personType"/>
  <xsd:complexType name = "personType">
    <xsd:sequence>
      <xsd:element name = "name" type = "xs:string"/>
      <xsd:element name = "place" type = "xs:string"/>
      <xsd:element name = "year" type = "xs:positiveInteger"/>
    </xsd:sequence>
    <xsd:attribute name = "id" type = "xs:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
Remote object references [4.3.4]

• (Only applies to CORBA & Java: distributed object model)
• NOT XML
• Remote object reference: identifier valid thru a DS
• Generated so unique over space and time
  • Lots of processes in a DS!
  • Must not reuse
Figure 4.13
Representation of a remote object reference

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
</tr>
<tr>
<td>interface of remote object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CORBA Object References (not in textbook)

• Object reference
  • Opaque handle for client to use
  • Identifies exactly one CORBA object
  • IOR == “Interoperable Object Reference”

• References may be passed among processes on different hosts
  • As parameters, return values, or “stringified”
  • ORB will marshall
  • ORB on receiver side unmarshalling will
    • create a proxy
    • return a pointer to it
  • Basically functions as a remote “pointer” that works across heterogeneity in language, OS, net, vendor, …
CORBA Object References (cont.)

Object Key
- Opaque to client
- ORB-specific

Object ID
- Can be created by user or POA

Components: Optional data
- e.g., alternate endpoint, info for security policies, etc
Multicast Communication [4.4]

• Point-to-point communications not great for process groups
• Multicast: 1:many communications, many uses
  • Fault-tolerance based on replicated services
  • Discovering services in spontaneous networking
  • Better performance through replicated data (multicast updates)
  • Propagation of event notices (Facebook, implement pub-sub)
IP Multicast (IPMC) [4.4.1]

- One implementation of multicast, using UDP not TCP
  - Use normal sockets to join (receiving) group
- Multicast routers: can send to multiple LANs (use its mcast)
- Multicast addresses
  - Permanent: assigned by IANA, exist even if no members
  - Temporary: come and go dynamically
- Failure model:
  - Same as UDP (omission)
  - Some group members may receive, others not
  - AKA **unreliable multicast** (reliable multicast in Chapter 15, for 562)
Java API for IP Multicast

• **Class** MulticastSocket
  • **Subclass of** DatagramSocket
  • `joinGroup(...)`
  • `leaveGroup(...)`
import java.net.*;
import java.io.*;
public class MulticastPeer{
  public static void main(String args[]){
    // args give message contents & destination multicast group (e.g. "228.5.6.7")
    MulticastSocket s =null;
    try {
      InetAddress group = InetAddress.getByName(args[1]);
      s = new MulticastSocket(6789);
      s.joinGroup(group);
      byte [] m = args[0].getBytes();
      DatagramPacket messageOut =
          new DatagramPacket(m, m.length, group, 6789);
      s.send(messageOut);
    }
  }

  // this figure continued on the next slide
// get 3 messages from others in group
byte[] buffer = new byte[1000];
for(int i=0; i< 3; i++) {
    DatagramPacket messageIn =
        new DatagramPacket(buffer, buffer.length);
    s.receive(messageIn);
    System.out.println("Received:" + new String(messageIn.getData()));
}
s.leaveGroup(group);
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally {if(s != null) s.close();}
Reliability and ordering of multicast [4.4.2]
• IPMC: dropped msgs, partial delivery to group, no ordering
• Effects on different apps?
  • Fault-tolerance based on replicated services
    • E.g., keep replicas with same state, multicast requests
    • What happens with failures above?
  • Discovering services in spontaneous networking
    • What happens with failures above?
• Better performance through replicated data (multicast updates)
  • What happens with failures above?
• Propagation of event notices (Facebook, implement pub-sub)
  • What happens with failures above?
Network virtualization: overlay networks [4.4]

- Some applications need (much) more advanced delivery services than Internet protocols provide
  - End-to-end argument says to not push functions down here
- Network virtualization: construct many different virtual networks over Internet
  - Support specific services needed
  - Answers end-to-end argument: app-specific virtual network
Overlay networks [4.5.1]

• Overlay network: virtual network consisting of topology of virtual nodes and virtual links (above underlay network’s)
  • Tailor services to specific app (e.g., multimedia content)
  • More efficient in some network environments (ad hoc)
  • Add more features: multicast, secure communications, …

• Can redefine addressing, protocols, routing

• Advantages
  • Add services without having to change (and standardize) underlay
  • Encourage experimentation
  • Can exist with other overlays (same kind or different)
Overlay networks (cont)

• Disadvantages:
  • extra level of indirection
    • Placement (overlay→underlay) is key for efficiency
  • add to complexity

• Examples in book
  • Skype next
  • Chap 10: P2P file sharing, distributed hash tables
  • Chap 19: mobile/ubiquitous: ad hoc and disruption-tolerant
  • Chap 20: multimedia streaming
### Figure 4.15
Types of overlay

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tailored for application needs</td>
<td>Distributed hash tables</td>
<td>One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).</td>
</tr>
<tr>
<td>Peer-to-peer file sharing</td>
<td>Overlay structures</td>
<td>Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.</td>
</tr>
<tr>
<td>Content distribution networks</td>
<td>Overlays that subsume a range</td>
<td>Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [<a href="http://www.kontiki.com">www.kontiki.com</a>].</td>
</tr>
</tbody>
</table>
### Types of overlay

<table>
<thead>
<tr>
<th>Tailored for network style</th>
<th>Wireless ad hoc networks</th>
<th>Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disruption-tolerant networks</td>
<td>Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.</td>
<td></td>
</tr>
<tr>
<td>Offering additional features</td>
<td>Multicast</td>
<td>One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobson, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].</td>
</tr>
<tr>
<td>Resilience</td>
<td>Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].</td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>Overlay networks that offer enhanced security over the underlying IP network, including virtual private networks, for example, as discussed in Section 3.4.8.</td>
<td></td>
</tr>
</tbody>
</table>
Skype [4.5.2]

- P2P VOIP overlay network (instant msgs, video, telephony)
- Addresses: skype username or phone number
- Original architecture: P2P
  - Ordinary user machines “hosts”
  - Well enabled/connected hosts: super node
- Authenticate users over well-known login server, gives them super node
- Supernodes goal: search for users efficiently
- Direct voice connection between two parties
  - Signalling: TCP
  - Call: UDP or TCP (latter only to circumvent firewalls)
- Codecs key: optimized for ≥ 32 kbps
Figure 4.16
Skype overlay architecture (pre-cloud)
Case Study: MPI [4.6]

• Started in grid computing (and influenced it)
• NOT Covering: required for 564 (not for 464)