Algorithms and Programming IV
Communication Paradigms in Distributed Systems

Summer Term 2023 | 19.06.2023
Claudia Müller-Birn, Barry Linnert
Our topics today

Recap

Architectural Styles

- Layered architectures
- Service-oriented architectures
- Publish-subscribe architectures

Communication Paradigms

- Interprocess Communication
  - API for Internet Protocols
  - UDP Datagram Communication
  - TCP Stream Communication
RECAP
Recap: Distributed System Model
A CLASSIFICATION OF DISTRIBUTED SYSTEMS
Recap: Architectural Model

<table>
<thead>
<tr>
<th>Architectural elements</th>
<th>Communication paradigm</th>
<th>Roles and responsibilities</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communicating entities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processes</td>
<td>Inter-process communication</td>
<td>Architectural styles</td>
<td>Multiple server</td>
</tr>
<tr>
<td>Objects</td>
<td>UDP sockets</td>
<td>Client-server</td>
<td>Proxy/Cache</td>
</tr>
<tr>
<td>Components</td>
<td>TCP sockets</td>
<td>Peer-to-peer</td>
<td>Mobile code</td>
</tr>
<tr>
<td>Web Services</td>
<td>Multi-cast</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indirect communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remote invocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Architectural Styles

- ALP IV: Concepts of Non-Sequential and Distributed Programming | Summer Term 2023
Communication Paradigms in Distributed Systems

ARCHITECTURAL STYLES
Basic Idea

A **style** is formulated in terms of

- (replaceable) components with well-defined interfaces
- the way that components are connected to each other
- the data exchanged between components
- how these components and connectors are jointly configured into a system.

**Connector**

- A mechanism that mediates communication, coordination, or cooperation among components. *Example*: facilities for (remote) procedure call, messaging, or streaming.
Architectural Styles in Distributed Systems

- Layered architectures
- Service-oriented architectures
- Publish-subscribe architectures
Layered architecture - Different layered organizations

Request/Response
downcall

Layer N
Layer N-1
Layer 2
Layer 1

One-way call

Layer N
Layer N-1
Layer N-2
Layer N-3

Layer N
Layer N-1
Handle
Layer N-2
Upcall


ALP IV: Concepts of Non-Sequential and Distributed Programming | Summer Term 2023
Example: Communication Protocols
Application Layering

Traditional three-layered view

- The **application-interface** layer contains units for interfacing to users or external applications
- The **processing layer** contains the functions of an application, i.e., without specific data
- The **data layer** contains the data that a client wants to manipulate through the application components

**Observation:** This layering is found in many distributed information systems, using traditional database technology and accompanying applications.
Example: A Simple Search Engine

- **User interface**
  - User-interface level
  - HTML page containing list
- **Query generator**
  - Processing level
  - Ranked list of page titles
- **HTML generator**
  - Database queries
  - Database with Web pages
- **Ranking algorithm**
  - Web page titles with meta-information

Service-oriented Architectures

• A layered architectural style’s significant drawback is the often strong dependency between different layers.
• Such direct dependencies have led to an architectural style reflecting a more loose organization into a collection of separate, independent entities.
• Each entity encapsulates a service (can be called services, objects, or microservices).
• Each service is executed as a separate process (or thread).
Object-based style

Essence
Components are objects, connected to each other through procedure calls. Objects may be placed on different machines; calls can thus execute across a network.

Encapsulation
Objects are said to encapsulate data and offer methods on that data without revealing the internal implementation.
RESTful architectures

Essence

View a distributed system as a collection of resources, individually managed by components. Resources may be added, removed, retrieved, and modified by (remote) applications.

1. Resources are identified through a single naming scheme
2. All services offer the same interface
3. Messages sent to or from a service are fully self-described
4. After executing an operation at a service, that component forgets everything about the caller

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>Create a new resource</td>
</tr>
<tr>
<td>GET</td>
<td>Retrieve the state of a resource in some representation</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a resource</td>
</tr>
<tr>
<td>POST</td>
<td>Modify a resource by transferring a new state</td>
</tr>
</tbody>
</table>
Example: Amazon’s Simple Storage Service

Essence

- Objects (i.e., files) are placed into buckets (i.e., directories). Buckets cannot be placed into buckets. Operations on ObjectName in bucket BucketName require the following identifier:

  http://BucketName.s3.amazonaws.com/ObjectName

Typical operations (all operations are carried out by sending HTTP requests):

- Create a bucket/object: PUT, along with the URI
- Listing objects: GET on a bucket name
- Reading an object: GET on a full URI
On interfaces

Issue

• Many people like RESTful approaches because the interface to a service is so simple. The catch is that much needs to be done in the parameter space.

Amazon S3 SOAP interface

<table>
<thead>
<tr>
<th>Bucket operations</th>
<th>Object operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListAllMyBuckets</td>
<td>PutObjectInline</td>
</tr>
<tr>
<td>CreateBucket</td>
<td>PutObject</td>
</tr>
<tr>
<td>DeleteBucket</td>
<td>CopyObject</td>
</tr>
<tr>
<td>ListBucket</td>
<td>GetObject</td>
</tr>
<tr>
<td>GetBucketAccessControlPolicy</td>
<td>GetObjectExtended</td>
</tr>
<tr>
<td>SetBucketAccessControlPolicy</td>
<td>DeleteObject</td>
</tr>
<tr>
<td>GetBucketLoggingStatus</td>
<td>GetObjectAccessControlPolicy</td>
</tr>
<tr>
<td>SetBucketLoggingStatus</td>
<td>SetObjectAccessControlPolicy</td>
</tr>
</tbody>
</table>
Insert: Simple Object Access Protocol (SOAP)

SOAP is designed to enable both client-server and asynchronous interaction over the Internet. It defines a scheme for using XML to represent the contents of request and reply messages and a scheme for the communication of documents.

It is used for information exchange and RPC, usually (but not necessarily) over HTTP.

(Very) basic SOAP architecture:
On interfaces (cont.)

Simplifications

- Assume an interface bucket offering an operation create, requiring an input string such as mybucket, for creating a bucket “mybucket.”

  SOAP
  
  ```groovy
  import bucket
  bucket.create("mybucket")
  ```

  RESTful
  
  ```groovy
  PUT "https://mybucket.s3.amazonaws.com/
  ```

Conclusions

- Are there any to draw?
Publish-Subscribe Architectures

Temporal and referential coupling

<table>
<thead>
<tr>
<th></th>
<th>Temporally coupled</th>
<th>Temporally decoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referentially coupled</td>
<td>Direct</td>
<td>Mailbox</td>
</tr>
<tr>
<td>Referentially decoupled</td>
<td>Event-based</td>
<td>Shared data space</td>
</tr>
</tbody>
</table>

Event-based: communicating parties need to be both online

Shared data space: data to communicate can be stored
## Recap: Architectural Model

### Architectural Elements

<table>
<thead>
<tr>
<th>Communicating entities</th>
<th>Communication paradigm</th>
<th>Roles and responsibilities</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes</td>
<td>Inter-process communication</td>
<td>Architectural styles</td>
<td>Multiple server</td>
</tr>
<tr>
<td>Objects</td>
<td>UDP sockets</td>
<td>Client-server</td>
<td>Proxy/Cache</td>
</tr>
<tr>
<td>Components</td>
<td>TCP sockets</td>
<td>Peer-to-peer</td>
<td>Mobile code</td>
</tr>
<tr>
<td>Web Services</td>
<td>Multicast</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indirect communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remote invocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Architectural Styles

- Client-server
- Peer-to-peer
- Mobile code
- Multiple server
- Proxy/Cache
- Multicast
Architectural elements

COMMUNICATION PARADIGM
### An Architectural Model of Distributed Systems

<table>
<thead>
<tr>
<th>Architectural elements</th>
<th>Communication paradigm</th>
<th>Roles and responsibilities</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communicating entities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication paradigm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>How do they communicate, or, more specifically, what communication paradigm is used?</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ALP IV: Concepts of Non-Sequential and Distributed Programming | Summer Term 2023**
Types of Communication Paradigms

- Interprocess communication
- Remote invocation
- Indirect communication
## Types of Communication Paradigms

<table>
<thead>
<tr>
<th>Architectural elements</th>
<th>Communication paradigm</th>
<th>Roles and responsibilities</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communicating entities</td>
<td>Interprocess communication</td>
<td>Architectural styles</td>
<td>Multiple server</td>
</tr>
<tr>
<td>Processes</td>
<td>UDP sockets</td>
<td>Client-server</td>
<td>Proxy/Cache</td>
</tr>
<tr>
<td>Objects</td>
<td>TCP sockets</td>
<td>Peer-to-peer</td>
<td>Mobile code</td>
</tr>
<tr>
<td>Components</td>
<td>Multicast</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Communication Paradigm

INTERPROCESS COMMUNICATION
Interprocess Communication

Interprocess Communication (IPC) mechanisms provide a low-level support to enable processes from different address spaces to connect and exchange information.

A process is an object of the operating system through which applications gain secure access to computer resources. Individual processes are isolated from each other for this purpose.

IPC is based on the exchange of messages (= bit sequences).
- A message is sent by the one process (the sender).
- It is received by another process (the receiver).
Motivation for IPC

process \( p \)

\( \text{send} \ m \)

Communication channel

Outgoing message buffer

process \( q \)

\( \text{receive} \)

Incoming message buffer
Protocols

- Protocol refers to a set of **rules and formats** to be used for communication between processes in order to perform a given task.

outlined:

- Specification of the sequence of messages that must be exchanged.
- Specification of the format of the data in the messages.
Protocol layers in the ISO Open Systems Interconnection (OSI) model

Connection-oriented communication: sending and receiving processes synchronize at every message = send and receive are blocking operations

Connectionless communication: send and receive operations are non-blocking
## Layers ISO Model vs. TCP/IP Model

<table>
<thead>
<tr>
<th>ISO Model</th>
<th>TCP/IP Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Presentation</td>
<td>Transport</td>
</tr>
<tr>
<td>Session</td>
<td>Internet</td>
</tr>
<tr>
<td>Transport</td>
<td>Network Access</td>
</tr>
<tr>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>Data link</td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td></td>
</tr>
</tbody>
</table>
TCP/IP Model

HTTP    DNS    SNMP

TCP      UDP

IP (IPv4 / IPv6)

Ethernet

Application  Transport  Internet  Network Access
UDP vs. TCP

UDP (User Datagram Protocol)
• UDP differs from the IP service only in the additional specification of the ports: a message is sent as a datagram through the network without the arrival at the destination port being guaranteed (connectionless service).

TCP (Transmission Control Protocol)
• TCP establishes a virtual connection between a client port and a provider port and thus two opposing, reliable, sequence-true (FIFO) byte streams (connection-oriented service).
Sockets

On approach to realize interprocess communication consists of transmitting a message between a socket in one process to a socket in another process.

Internet address = 138.37.94.248

Internet address = 138.37.88.249
Interprocess communication

API FOR INTERNET PROTOCOLS
Java API: package java.net

Java provides class InetAddress that represents Internet addresses.

Method static InetAddress getByName(String host)
Can throw an UnknownHostException

Example

```java
System.out.println(InetAddress.getByName("www.fu-berlin.de"));
www.fu-berlin.de/160.45.170.10
System.out.println(InetAddress.getByName("localhost"));
localhost/127.0.0.1
System.out.println(InetAddress.getLocalHost());
wing.local/192.168.183.35
```
API for Internet protocols

UDP DATAGRAM COMMUNICATION
UDP Sockets

1. Client creates socket bound to a local port

2. Server binds its socket to a server port

3. Client/Server send and receive datagrams

4. Ports and sockets are closed

Slide adapted from Peter Löhr/Robert Tolksdorf
Using UDP for Applications

Advantage of UDP datagrams is that they do not suffer from overheads associated with guaranteed message delivery.

**Example 1: Domain Name System**

- DNS primarily uses UDP on port number 53 to serve requests
- DNS queries consist of a single UDP request from the client followed by a single UDP reply from the server

**Example 2: VOIP**

- No reason to re-transmit packets with bad speech data
- Speech data must be processed at the same rate as it is sent - there is no time to retransmit packets with errors
UDP datagram communication

JAVA API FOR UDP DIAGRAMS
Java API for UDP diagrams

Datagram communication is provided by two classes
DatagramPacket and DatagramSocket

DatagramPacket
- Constructor that makes an instance out of an array of bytes comprising a message
- Constructor for use when receiving a message, message can be retrieved by the method `getData`

DatagramSocket
- Constructor that takes port number as argument for use by processes
- No-argument constructor for choosing a free local port
Example: UDP Echo Server

```java
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        // ...
    }
}
```
... while (true) {
    DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
    serverSocket.receive(receivePacket);
    String sentence = new String(receivePacket.getData());
    InetAddress IPAddress = receivePacket.getAddress();
    int port = receivePacket.getPort();
    String capitalizedSentence = sentence.toUpperCase();
    sendData = capitalizedSentence.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
    serverSocket.send(sendPacket);
}
Example: Java Client (UDP)

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

class UDPClient {
    public static void main(String args []) throws Exception {
        BufferedReader inFromUser = new BufferedReader(new
        InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress
        .getByName("localhost");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
    }
}
```
Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from server

sendData = sentence.getBytes();
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
API for Internet protocols

TCP STREAM COMMUNICATION
TCP Sockets Communication

1. Server bind port

2. Server is ready and listening

3. Server is waiting for request, client sends request, server accepts

4. Client and server are connected - bidirectional!

5. Connection is closed
Use of TCP for Applications

Many frequently used services run over TCP connections with reserved port numbers.

- **HTTP** [RFC 2068]: The Hypertext Transfer Protocol is used for communication between web browser and web server.
- **FTP** [RFC 959]: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.
- **Telnet** [RFC 854]: Telnet provides access by means of a terminal session to a remote computer.
- **SMTP** [RFC 821]: The Simple Mail Transfer Protocol is used to send mail between computer.

Java API for TCP streams

Java interface provides two classes `ServerSocket` and `Socket`.

**ServerSocket**
- Class is intended to be used by server to create a socket at a server port for listening for connect requests from clients.

**Socket**
- Class is for use by a pair of processes with a connection
- The client uses a constructor to create a socket, specifying the DNS hostname and port of a server
TCP Echo Server

```java
public class EchoServer {
    public static void main(String args[]) throws IOException {
        ServerSocket listen = new ServerSocket(1234);
        while (true) {
            Socket socket = listen.accept();
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintStream out = new PrintStream(socket.getOutputStream());
            //...
```java
while (true) {
    String message = in.readLine();
    if (message == null) {
        break;
    }
    String answer = message.replace('i', 'o');
    out.println(answer);
}
in.close();
out.close();
socket.close();
System.out.println("Socket closed.");
}
```
```java
public class Client {

    public static void main(String args[]) throws IOException {

        Socket socket = new Socket("localhost", 1234);
        PrintStream out = new PrintStream(socket.getOutputStream());

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));

        // ...
while (true) {
    String message = keyboard.readLine();

    if (message == null)
        break;

    out.println(message);
    String answer = in.readLine();
    System.out.println("echo: "+ answer);
}

in.close();
out.close();
socket.close();
}
Notice!

However, the service `echo` is quite limited in that it cannot have several sessions at the same time. If you want to use the service, you may have to wait until an active user closes the session.

What might be a solution?
Example EchoServer Extended

```java
public class EchoServerExtended extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintStream out;

    public EchoServerExtended(Socket socket) throws IOException{
        this.socket = socket;
        this.in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        this.out = new PrintStream(socket.getOutputStream());
    }
```
Summary…

Sockets only provide the basic mechanisms, there is still work to be done, for example the implementation of more complex system models such as Request-Reply (for client-server) or group communication.

Above all, however, the necessity of homogeneous data representation in heterogeneous environments is a major issue.

These are the basic techniques for more complex middleware such as RPC, Java RMI. We will talk about it next 😊
Open Topics in IPC – we discuss them in the next lecture

- External data representation and marshalling
- Multicast communication
- Network virtualization: Overlay networks
References

Main resources for this lecture:

Algorithms and Programming IV
Remote Invocation: Remote Procedure Calls

Summer Term 2023 | 21.06.2023
Claudia Müller-Birn, Barry Linnert