Distributed Programming

Dan Cosma

About

What's in a name?

Programarea rețelelor de calculatoare

Computer Network Programming

Programarea aplicațiilor distribuite

Distributed Programming

Purpose

Introduction to Distributed Software Systems

→ learning to program in a networked environment
→ understanding and using the main techniques and technologies

Introduction

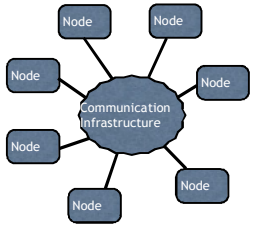
Distributed System

- ”a collection of independent computers that appears to its users as a single, coherent system” [TS01]
- an arbitrary number of processing elements running at different locations, interconnected by a communication system [Wu99]

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. “Distributed Systems:
Principles and Paradigms.” Prentice Hall, 2001.
[Wu99] Jie Wu. “Distributed Systems Design.” CRC Press LLC, 1999.

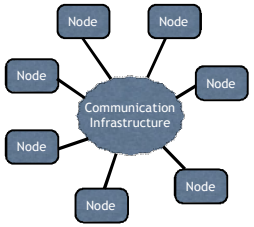
Distributed Systems

- Multiple processing units running in nodes situated at different locations
- Communication is done via an infrastructure
- The architecture is heterogenous



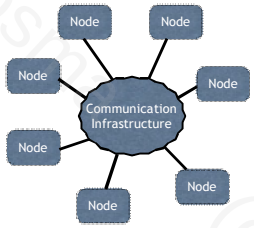
Nodes

- System components running in nodes:
  - are independent programs that have dual functionality:
    - local
    - network-aware
  - can be written in various languages / can run on different platforms



Communication Infrastructure

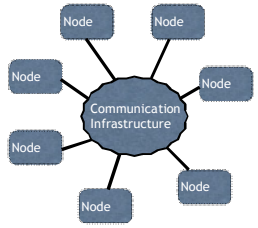
- Handles the data transmission and event notification over the network
- Is available through libraries, language constructs or platform-specific services



9

Communication Infrastructure

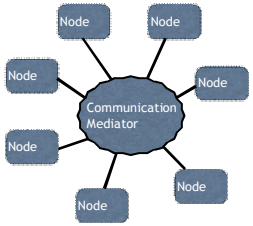
- Is built to hide the communication details, at different levels of abstraction
- Is directly related to technological concerns



10

Communication Infrastructure

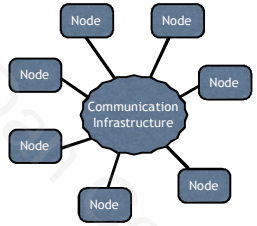
- We call it Communication Mediator



11

It's a SYSTEM

- The components, however loosely coupled, work together for a common goal, and represent parts of the same system



12

Do We Need Distributed Software ?

Do We Need Distributed Software ?

- The real world is distributed
  - Organizations span over multiple locations
  - People communicate over the internet even in their spare time
  - The world depends on computer networks

What types of systems are needed?

- Applications that seamlessly integrate our workplaces, homes, ourselves so that we
  - find information easily
  - work productively
  - live in sync with the others
- ... without being involved in the details

Distributed applications

- Answer real-world concerns
- Connect communities
- Widely used: basically all modern applications are network-aware, and the ability to communicate over the network is essential for their functionality

[1] OASIS (Organization for the Advancement of Structured Information Standards) Reference Model for Service Oriented Architecture 1.0

Therefore...

Therefore...

Learning how to develop distributed programs is important

Communication

- The main concern in network-aware applications
- To find the common language between the various components of the application, the engineer must
  - understand their differences
  - find common platform-related communication infrastructures
  - describe a communication policy (and protocol)
  - adapt the components where necessary

  (preferably write them after designing the policy)

19

Communication

- Usually, an application is homogenous, as it is developed on a single platform/technology
- When components must run on different environments:
  - the constraints imposed by the environments are vital when finding the carrier for the "common language"
  - compromises must be made

  (e.g. mobile clients may impose performance constraints, and limit the choice to "simple" technologies such as direct socket connections or simple HTTP exchanges)

20

Communication

- The communication policy
  - describes the layout of the communication, around a communication protocol
- Communication protocol
  - the set of rules, formats, and conventions that govern the communication between components
  - an essential part in the system design
  - ensures long-term compatibility of components
  (long-term, referring both to the system's run time and the system's evolution)

21

Types of Distributed Applications

A Loose Classification of Distributed Applications

- Classic communication applications: FTP, Web browsing, remote shell, remote desktop, etc.
- Web applications
- File and information sharing
- Distributed databases
- Enterprise applications
- Cloud Computing
- Multi-agent distributed systems

23

"Classic" Applications

- Provide single, specific features
- Rely on standardized protocols
- Dedicated clients and servers
- Components (client, server) can be implemented by distinct vendors
- Widely used, widely ported

24

Web Applications

- Use the HTTP protocol
- The client is always the generic Web browser
- Traditionally, the functionality is almost entirely server-side
- The browser sends HTTP requests, the server generates HTML pages as response
- Session management is complex (HTTP is a bit primitive)
- Client-side functionality requires "imaginative" workarounds (now available as dedicated frameworks)

25

Social networks

- Mainly web applications
- Usually provide dedicated mobile clients (phones, tablets, other useful gadgets)
- Part of the "Cloud" services
- Exceedingly popular

26

File/information sharing

- Usually peer-to-peer
- Widely distributed over the network
- Robust software architectures (node failure impact is limited)
- Minimal centralization, usually for locating peers
- Excellent for large data distribution (e.g. Linux)
  - greatly reduces the load on traditional file servers
- Well-suited for multimedia streaming
  - eliminates the server bottleneck and reduces communication costs

27

Distributed Databases

- Distributed database = a database that uses storage located on multiple computers
- It is seen as a single database, but the data can span over multiple locations
- Uses replication and duplication to maintain consistency
  - replication: changes propagate to all nodes
  - duplication: data in a "master" node is copied in all other nodes

28

Enterprise Applications

- Enterprise application = software used in organizations, usually large
- Types:
  - built in-house
  - custom-made by another party or outsourced
  - Software as a Service (SaaS) accessed via the internet

29

Cloud Computing

- = software services that do not require the user's involvement in the deployment, configuration and hosting
- Based on the utility computing model
  = computing services are like public utilities, similar to the traditional ones (gas, water, electricity, etc.)
- Cloud computing providers expose their services online
- Users need minimal resources to connect as clients

30

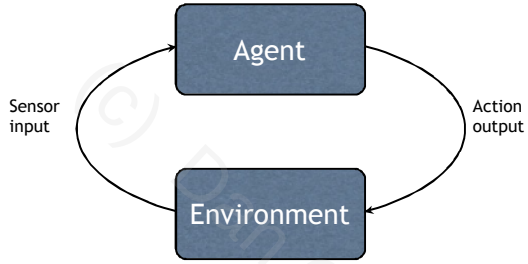
Distributed Agents

- Agent (Michael Woolridge [1]):
  = "a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design requirements"

[1] Gerhard Weiss - editor: Multiagent Systems - A Modern Approach
to Distributed Artificial Intelligence, The MIT Press, 1999.

31

Agent



Source: M. Wooldrige: Intelligent Agents, from the book edited by Gerhard Weiss: Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence, The MIT Press, 1999, page 29.

32

Examples of Agents

- Control systems, e.g. a temperature monitor in an air conditioning system
- E-mail notification agents
- "Spam" filters
- Synchronization agents (e.g. between a phone and a computer)

33

Intelligent Agents

- A software agent which, besides being autonomous, features
  - reactivity
  the ability of perceiving and responding to environment changes
  - proactivity
  the ability to take the initiative in order to fulfill their goals
  - social ability
  the ability to communicate with other software agents

34

Distributed Agents

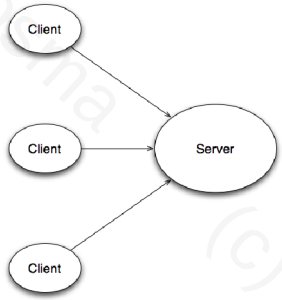
- Agents, with various degrees of intelligence, which
  - communicate to other agents other via the network,
  and/or
  - are able to migrate from one location to another (mobile agents)
- Multi-agent distributed systems: complex distributed systems made of multiple distributed agents that work for a common goal

35

Distributed Architectures

Client-server

- The most common architecture
- Server: provides a set of services
- Client: uses the services
- The client initiates the communication
- Usually clients are lighter than servers

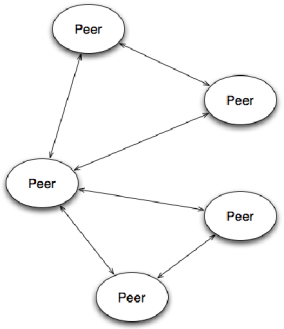


Client-server

- Advantages
  - the core functionality is in one place (server)
  - easy to implement and manage
  - easy to control, evolve
  - Allows for thin clients
- Disadvantages
  - hard to scale up
  - clients depend on the server interface
  - centralization, bottleneck, server too critical

Peer-to-peer

- The components are balanced – they can play both client and server roles
- The runtime layout of the system is flexible: peers can join the network, or can leave it at any time
- Peers communicate to each other as needed (no fixed channels)
- Decentralization is key
- Multiple communication paths provide redundancy

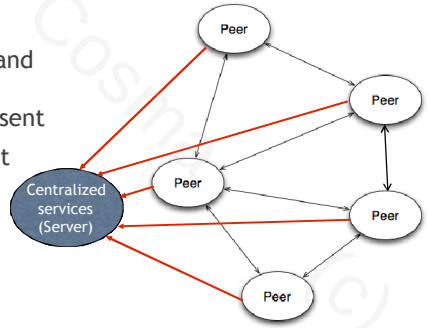


Peer-to-peer

- Advantages
  - flexible, scalable
  - decentralized
  - survives (partial) component failure
- Disadvantages
  - complex architecture
  - not easy to manage
  - hard to discover the running peers

Mixed architectures

- Both client-server and peer-to-peer subsystems are present
- Example: BitTorrent



May solve the peer-to-peer problems (e.g. discovery, coordination) with a minimal compromise in centralization

41

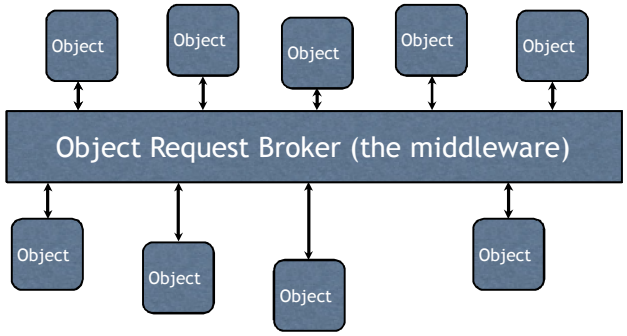
Distributed Objects

- Fundamental architectural components: objects
- No a-priori distinction between clients and servers
- Objects define interfaces and provide services
- Objects use each other's services
- Objects are distributed over the network and communicate through specific middleware*

* Not all middleware systems are related to distributed objects, other types of middleware exist

42

Distributed Objects



43

Distributed Objects

- Advantages
  - open architecture: objects can be added as needed
  - scalability
  - provides a very good framework for interoperability
  - provides the possibility of dynamic reconfiguration (e.g. object migration)
- Disadvantages
  - complex infrastructures
  - comunication overhead due to complex protocol stacks
  - dependency on (sometimes) proprietary middleware

44

Service-Oriented Architectures

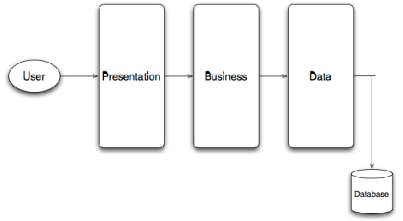
- Designed around the concept of providing services
  - Service = “an act or a performance offered by one party to another” (Lovelock et al., 1996)
  - Web Service = “a standard representation for some computational or informational resource that can be used by other programs” [1]
- Applications are built as collections of independent components that communicate by publishing and/or using services
- One same component (service) can be used or reused in multiple applications

For references, see Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006
[1] Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006

45

Three Tier

- Largely used in modern enterprise systems
- Presentation: interacts with the user
- Logic (Business): the main system functionality (e.g. algorithms)
- Data: models the data used by Business



46

Application Protocols

47

Communications Protocols

- Formal description of the communication between hardware or software components
- The usual information carrier is the message
- Describe
  - the message format (syntax, semantics)
  - the message exchange rules
  - synchronization, coordination during the message exchange

48

Software Communication

- Protocols established between software components
- Developed in the early stages of the design
- Transported by communication mediators specific to the application platform

49

Importance

- Protocols represent the common language for the communicating components
- Essential for providing
  - component integration
  - transport for system commands
  - adequate component coupling
  - efficiency in sending key system data
  - error handling and recovery

50

Usage scenarios

- Programs communicating over the network: TCP/IP, RMI, etc.
  the technology itself is not relevant here
- Software components in an application: pipes, IPC, etc.
- Operating systems components
- Kernel-level communication

51

Responsibilities

- Representation
  - represent the abstract data and the concepts in communication
- Authentication
  - provide mechanisms for ensuring the communication parties are genuine
- Authorization
  - mechanisms to make sure the parties are allowed to communicate
- Coordination
  - commands, rules of communication, acknowledgment of receipt, etc.
- Error handling

52

Protocol Layering

- Breaking a complex protocol into several simpler ones
- Layers represent functionalities: each solve a particular problem
- Layers communicate to each other

53

Example of Layering

- TCP/IP stack:
  - Application: communication specific to applications
  - Transport: end-to-end communication, including error control, flow control and application-level addressing (ports)
  - Internet: route packets over the network
  - Link: send packets between hosts, over the local network

54

Application Protocol

- Application-specific communication protocol
- Connects software components or applications
- Two approaches:
  - proprietary - built in-house for custom, specific applications
  - standardized - public specifications for others to use

55

Public application protocols

- Thoroughly specified in public documents (e.g. RFC's)
- Vendors or organizations can build components by only knowing the protocol: interoperability
- Examples:
  - FTP - File Transfer Protocol
  - SSH - Secure Shell protocol
  - SMTP - Simple Mail Transfer Protocol
  - HTTP - Hypertext Transfer Protocol
  - BitTorrent protocol
  - ...

56

Need a Protocol?

1. Find and existing one that fits (at least partially) your goals

2. Define a data exchange model over an existing protocol (e.g. over HTTP or SMTP)

3. Design a protocol from scratch

For details, see RFC 3117

57

Option 1.

Find and existing one that fits (at least partially) your goals

- Not so easy to find
- Even if found, is this really a better way?

Example: need to push or pull files, synchronously or asynchronously:

~ should we use FTP?

~ do we like all its aspects?
(authentication, negotiation, command ports, etc.)

We have to evaluate the costs of modifying the protocol:

is it really cheaper than developing it from scratch?

For details, see RFC 3117

58

Option 2.

Define a data exchange model over an existing protocol (e.g. over HTTP or SMTP)

Advantages:

- inherit all the infrastructure for transporting the data
(e.g. HTTP servers, proxies, authentication mechanisms, etc.)
- reuse the already existing tools (monitoring, development)

Disadvantages:

- protocols have limitations
(e.g. HTTP isn't flexible enough to support server-side asynchronous behavior)
- little room for extensibility

For details, see RFC 3117

59

Option 3.

Design a protocol from scratch

Now, that's an interesting idea! ;)

60

Designing a Protocol

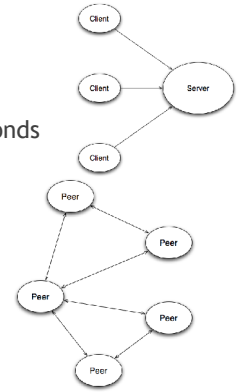
1. Choose the patterns of communication and data transmission
2. Establish the design goals
3. Choose the message format "philosophy"
4. Design the message structure: format, fields, types of messages, etc.
5. Design the communication rules (sequences)

Steps 4 and 5 go together

61

Patterns of communication

- Client - server
  one party initiates the communication, the other responds
- Peer-to-peer
  any party may initiate the communication

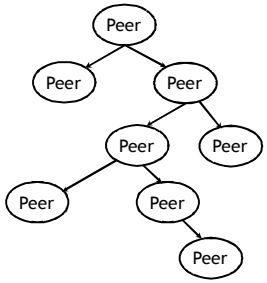


See Robin Sharp, *Principles of protocol design*, Springer, 2008

62

Patterns of communication

- Hierarchical communication
  many parties, organized in a hierarchy, and communicate only via the branches of the tree



See Robin Sharp, *Principles of protocol design*, Springer, 2008

63

Patterns of transmissions

- One-to-one
  only two parties involved in communication at a time
- Multicast
  one or more parties may transmit data to multiple parties at a time
- Broadcast
  one or more parties may transmit data to all parties at the same time

Each or all of these patterns may be used at different stages in the communication

64

Design Goals

- Define the framework for communication
  - Should the communication be fast?
  - Do we need reliable exchanges? (E.g. confirmations and such)
  - How important is the authentication of parties?
  - Is the transferred data confidential? What degree of authorization is needed?
  - How many types of parties are involved? Can they all communicate to each other?
  - Are there bandwidth or connection availability limitations?
  - Do we need to maintain communication channels? Are connectionless models more suitable, instead?
  - Do we need complex error handling?
  - ...

65

Design Goals

- A communication protocol should be:
  - simple
    - ~ don't make easy tasks hard to do
    - ~ don't provide two ways for doing the same thing
  - scalable
    - ~ estimate the number of clients per server (or peers communicating)
    - ~ design the protocol so that it balances the responsibilities (e.g. shifts the communication balance to the clients, to free the servers which are already full of responsibilities)
  - efficient
    - ~ minimize the command overhead
    - ~ minimize the data traffic
  - extensible
    - ~ make room for further extensions
    - ~ don't overdo it, though

66

Message formats

- Two approaches:
  - Text-oriented protocols
  - Protocols using binary messages

67

Text-Oriented

- All messages are readable character strings
- Advantages
  - human readable, easy to understand and monitor
  - flexible, easy to extend (if properly designed)
  - easy to test, even with "standard" clients (telnet?)
- Disadvantages
  - human readable, easy to read by unauthorized persons (without encryption)
  - may become complex, harder to parse in code
  - may make the messages unjustifiably large

68

Binary messages

- Messages are blocks of structured binary data
- Advantages
  - Better ways of structuring the data
  - Suitable for large or complex data transfers
  - Messages are as small as possible
- Disadvantages
  - Hard to read, debug or test
  - Need to consider the data representation conventions on hosts and network (e.g. the "endianness": little-endian vs. big-endian)

69

Designing the Message

- A very important aspect in protocol design
- Influences all the characteristics of the communication: scalability, efficiency, simplicity, extensibility
- The design involves two aspects:
  a) types of messages
  b) message structure

70

Types of Messages

- One message type for each distinct aspect of the communication
- Three categories of messages:
  - commands
  - data transfer
  - control

  Each category may include several message types

71

Command Messages

- Define the stages of the dialogue between the parties
- Address various communication aspects:
  - communication initiation or ending
  - describe the communication stage (e.g. authentication, status request, data transfer)
  - status changes (e.g. requests for switching to the data transfer mode)
  - resource changes (e.g. requests for new communication channels)
  - ...

72

Data transfer

- Messages that carry data over the network
- They are usually sent as a responses to specific commands
- Data is usually fragmented in multiple messages
- Besides the actual data, may describe:
  - the type of the binary data format
  - clues for the layout of the structured data (when the structure is flexible/dynamic)
  - data size, offset or sequence information
  - type of the data block: last / intermediary

73

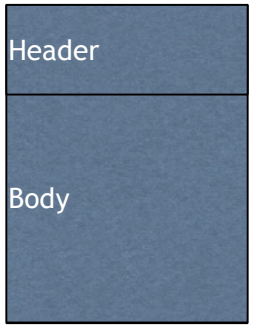
Control Messages

- Control the dialogue between the parties
- Address various communication aspects:
  - coordination (e.g. receipt confirmation, retry requests)
  - cancellation or interruption
  - availability checks
  - ...

74

Message Structure

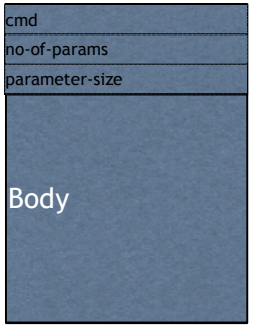
- Header: contains structured fields describing the actual data in the message:
  - message type
  - command
  - body size
  - recipient information
  - sequence information
  - retransmission count
  - etc.
- Body: the actual data to be transmitted:
  - the command parameters
  - the data payload



75

Message Structure

- The header structure must be well-known by the receiving party
- The header may contain clues helping the recipient to understand the rest of the message and the details regarding the data in the body (e.g. size, data format or encryption, etc.)
- Headers usually have a fixed size, while the body size may be variable (within limits)



76

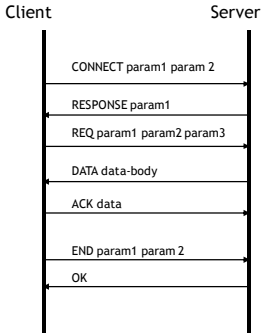
Communication Rules

- Along with the messages, this is the other essential part of the protocol
- Describe the sequences of commands, data and control messages, at each and all the stages in the communication, for all parties in the system
- Should be clearly and thoroughly specified, through detailed descriptions of each communication scenario (for each possible case of peer interaction)

77

Communication Rules

Sequence diagrams are more than useful...

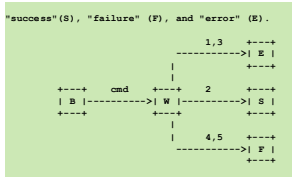


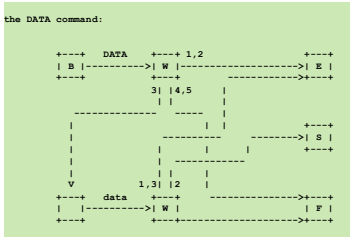
An imaginary diagram for a non-existent protocol

78

Communication Rules

...but *state diagrams* are almost mandatory





State diagrams from the SMTP specification (1982)
[source: RFC 821]

79

Documenting the Design

- The protocol specification must be available for all interested parties, as a specific document
- The specification must be
  - clear, easy to understand
  - comprehensive (complete)
  - non-ambiguous
  - maintainable (for versioning and such)

By only having the specification, parties must be able to thoroughly implement the software components involved in communication

80

Specification Content

- Introduction
  - purpose of the protocol, domain, environment, prerequisites
- The communication model
  - parties involved, relations, roles, general description of the dialogue flow between components, etc.
- Communication steps or procedures
  - description of each stage, procedure or aspect of communication
- Message description
  - syntax and semantics for all types of messages (commands, headers, codes, etc.)
- Sequence of commands and replies
  - the detailed description of the communication rules, including state diagrams, sequence diagrams, and comprehensive explanations for the procedures

81

Example: SMTP

- "Simple Mail Transfer Protocol"
- One of the oldest protocols still in widespread use (~1980)
- Designed for transporting outgoing e-mail
- Uses TCP port 25 (traditionally)
  Port 587 is also used for user agent connections ("e-mail clients").

82

RFC 821 Contents

TABLE OF CONTENTS

1. INTRODUCTION 1

2. THE SMTP MODEL 2

3. THE SMTP PROCEDURE 4

3.1. Mail 4
3.2. Forwarding 7
3.3. Verifying and Expanding 8
3.4. Sending and Mailing 11
3.5. Opening and Closing 13
3.6. Relaying 14
3.7. Domains 17
3.8. Changing Roles 18

4. THE SMTP SPECIFICATIONS 19

4.1. SMTP Commands 19
4.1.1. Command Semantics 19
4.1.2. Command Syntax 27
4.2. SMTP Replies 34
4.2.1. Reply Codes by Function Group 35
4.2.2. Reply Codes in Numeric Order 36
4.3. Sequencing of Commands and Replies 37
4.4. State Diagrams 39
4.5. Details 41
4.5.1. Minimum Implementation 41
4.5.2. Transparency 41
4.5.3. Sizes 42

APPENDIX A: TCP 44
APPENDIX B: NCP 45
APPENDIX C: NITS 46
APPENDIX D: X.25 47
APPENDIX E: Theory of Reply Codes 48
APPENDIX F: Scenarios 51

GLOSSARY 64

REFERENCES 67

83

The SMTP Model

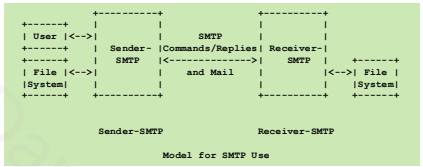
Sender-SMTP establishes a two-way communication with a Receiver-SMTP.

Receiver-SMTP may be the ultimate destination or an intermediary.

Sender sends a MAIL command. If Receiver can accept mail, responds OK.

Sender and Receiver negotiate recipients (RCPT).

Sender sends the e-mail (DATA) terminated with a specific sequence. Receiver confirms.



[source: RFC 821]

84

Main SMTP commands

`HELO <SP> <domain> <CRLF>`

Sender opens a connection. <domain> is the hostname of the sender.

`QUIT <CRLF>`

Sender closes the connection.

`MAIL <SP> FROM:<reverse-path> <CRLF>`

Starts the transaction. <reverse-path> is the source mailbox.

`RCPT <SP> TO:<forward-path> <CRLF>`

Specifies a recipient. Multiple RCPT commands are accepted. The receiver can accept it as a local destination, can reject it, or can forward it to another server.

`DATA <CRLF>`

Sends the mail content (text). Ends with <CRLF>.<CRLF> (a dot on a new line)

[source: RFC 821]

85

Example SMTP Conversation

```
R: 220 BBN-UNIX.ARPA Simple Mail Transfer Service Ready
S: HELO USC-ISIF.ARPA
R: 250 BBN-UNIX.ARPA
S: MAIL FROM:<Smith@Alpha.ARPA>
R: 250 OK
S: RCPT TO:<Jones@Beta.ARPA>
R: 250 OK
S: RCPT TO:<Green@Beta.ARPA>
R: 550 No such user here
S: RCPT TO:<Brown@Beta.ARPA>
R: 250 OK
S: DATA
R: 354 Start mail input; end with <CRLF>.<CRLF>
S: FROM: Mr. Smith <Smith@Alpha.ARPA>
S: TO: you (it could be an e-mail address here, a list, whatever)
S: SUBJECT: A very important message
S: Blah blah blah...
S: ...etc. etc. etc.
S: .
R: 250 OK
S: QUIT
R: 221 BBN-UNIX.ARPA Service closing transmission channel
```

[source: RFC 821, adapted]

86

Example of Forwarding

```
Example of Forwarding

     Either

     S: RCPT TO:<Postel@USC-ISI.ARPA>
     R: 251 User not local; will forward to <Postel@USC-ISIF.ARPA>

     Or

     S: RCPT TO:<Paul@USC-ISIB.ARPA>
     R: 551 User not local; please try <Mockapetris@USC-ISIF.ARPA>
```

[source: RFC 821]

87

Distributed Technologies – an overview

A selection of technologies

- Protocol stacks (TCP/IP)
  - data channels carrying information
- Remote procedure/method calls (RPC, RMI)
  - high-level language-specific constructs
- Message-oriented infrastructures (JMS)
  - third-party services transporting structured messages
- Application servers (JSP server, EJB container)
  - sophisticated environments managing the applications

89

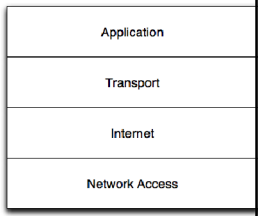
Protocol Stacks

- Describe facilities included in the modern operating systems
- Support network communication at the application level
- Dependent on a layered model describing the various types of concerns addressed
- Each layer defines a communication protocol

90

The TCP/IP protocol stack

- Network access layer: transmission of the data as datagrams to a remote host
- Internet layer: defines the network as interconnected subnetworks, deals with routing. Defines the IP address
- Transport layer: communication channels, error control, sequence of data arrival, etc.
- Application layer: protocols used by the application -- e.g. FTP, HTTP, SSH, etc.
- (usually) The main primitive for programs: the socket



Communication through primitives such as ”sockets”

91

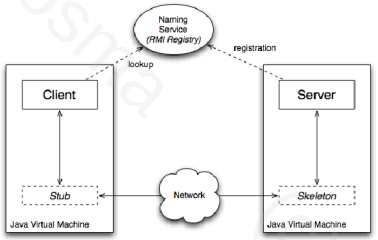
Remote invocation

- A communication method where pieces of software talk to each other over the network by means of higher-level constructs, such as function or method calls
- Software infrastructures abstract the actual data transmission, so that the developer writes the distributed code in a manner very similar to writing local applications
- Data is sent as function or method parameters, results are retrieved as returned values
- Examples:
  - Remote Procedure Call (UNIX)
  - Remote Method Invocation (Java)
  - Remote Invocation (CORBA)

92

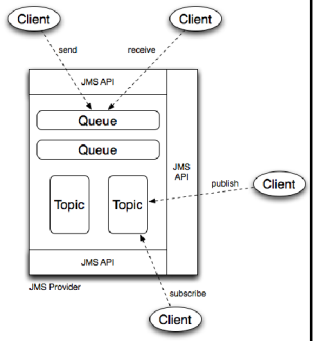
Remote Method Invocation (Java)

- Uses specific language constructs
- Hides the communication by providing natural ways of remote communication



Communication:
- objects publish methods available remotely
- specific connection / registration API calls

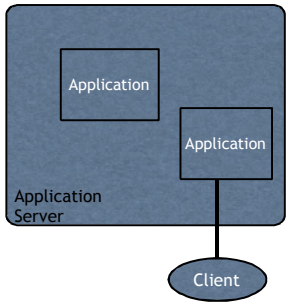
Messaging systems



94

Application Servers

- Provide an environment for running the application
- Applications run inside the application server (hence it is sometimes called container)
- Applications are provided complex features (transactions, persistency, distribution, etc.)
- Constraints: applications are strictly limited to specific rules



Java Remote Method Invocation

Java Remote Method Invocation (RMI)

A technology native to the standard Java platform

Provides support for network communication

Uses language-specific mechanisms

97

Main concept

Objects can be made accessible
through the network

>> a subset of their methods are *published* for other to use

>> the published methods represent the
remote services the respective class provides

98

Remote Interfaces

The published functionality is gathered in specific interfaces called *remote interfaces*

Remote interfaces must implement `java.rmi.Remote`

A remote interface declares methods that will be accessible through the network

A class may implement as many remote interfaces as necessary

99

RemoteException

When communicating over the network, specific errors may occur

>> All methods in a remote interface must throw
`java.rmi.RemoteException`

100

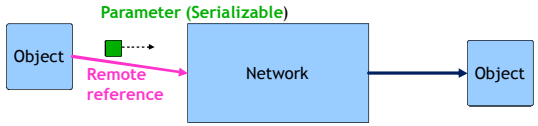
Classes with Remote Methods

To publish methods over the network, a class must:

- Implement a remote interface
- Explicitly export the remote functionality:
  >> extend java.rmi.UnicastRemoteObject
  or
  >> call UnicastRemoteObject.*exportObject()*

101

Remote References and Serializable Objects

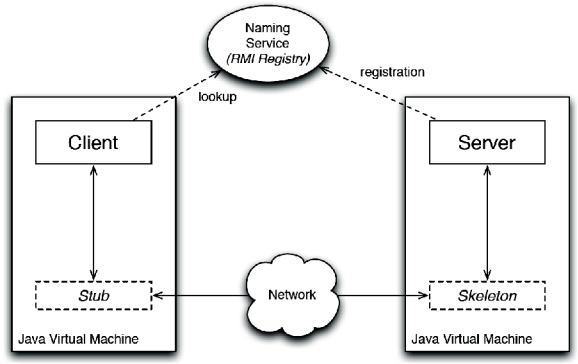


An object that uses a remote object receives a so-called *remote reference* to the latter

Parameters or results that must be transported over the network must be *Serializable* objects

102

The RMI Architecture



103

An RMI Example

Client-server application

- The server provides the current date and time
- Clients may request the date, the time, or both

https://sites.google.com/site/aplicatiidistribuite/

104

An RMI Example

Packages:
- sprc.rmiex.server
  - the server
- sprc.rmiex.client
  - the client
- sprc.rmiex.server.pub
  - classes used both by the client and the server

Directories:
- src/client
- src/server

105

The Remote Interface

```
package sprc.rmiex.server.pub;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface TimeServer extends Remote
{
    public String getTime(Format f)
        throws RemoteException;
}
```

106

The Parameter Class

```
package sprc.rmiex.server.pub;

import java.io.Serializable;

public class Format implements Serializable
{
    public boolean showDate = false;
    public boolean showTime = true;

    public Format(boolean showDate,
                  boolean showTime)
    {
        this.showDate = showDate;
        this.showTime = showTime;
    }
}
```

107

The Server Class

```
package sprc.rmiex.server;

import java.rmi.*;
import java.rmi.server.*;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import sprc.rmiex.server.pub.Format;
import sprc.rmiex.server.pub.TimeServer;

public class TimeServerImpl
        extends UnicastRemoteObject
        implements TimeServer
{
public TimeServerImpl() throws RemoteException
{
  super();
}
```

108

```
public String getTime(Format f)
{
  //creare calendar
  Calendar cal = new GregorianCalendar();
  //initializarea calendarului cu data si ora curenta
  cal.setTime(new Date());
  String raspuns = "Acum: ";

  if(f.showDate)
    raspuns = raspuns
      + cal.get(Calendar.YEAR)
      + "-" + cal.get(Calendar.MONTH)+ "-"
      + cal.get(Calendar.DAY_OF_MONTH) + ",";

  if(f.showTime)
    raspuns = raspuns
      + cal.get(Calendar.HOUR_OF_DAY) + ":"
      + cal.get(Calendar.MINUTE) + ":"
      + cal.get(Calendar.SECOND);

  return raspuns;
}
```

The Server Class


```
public static void main(String[] args)
{
  //Instalarea managerului de securitate
  //pentru RMI
  if (System.getSecurityManager() == null)
    {
      System.setSecurityManager(
           new RMISecurityManager());
    }

  //Inregistrarea serverului la serviciul de nume
  try {
    TimeServer server = new TimeServerImpl();
    Naming.rebind("//localhost/TimeServer", server);
    System.out.println("TimeServer started.");
  } catch (Exception e) {
    System.err.println("[TimeServer] Error:" +
                       e.getMessage());
  }
}
}
```

The Server Class

The Client

```
package sprc.rmiex.client;

import java.rmi.Naming;
import java.rmi.RMISecurityManager;

import sprc.rmiex.server.pub.Format;
import sprc.rmiex.server.pub.TimeServer;

public class Client {

public static void main(String[] args)
{
  if (System.getSecurityManager() == null)
  {
    System.setSecurityManager(
         new RMISecurityManager());
  }
```


The Client

```
try{
  TimeServer server = (TimeServer)
     Naming.lookup("//localhost/TimeServer");
  Format f = new Format(true, true);

  String s = server.getTime(f);
  System.out.println("Serverul a raspuns: "+ s);
} catch (Exception e) {
    System.err.println("[Client] Eroare:" +
         e.getMessage());

  }
}
}
```

Looking Up for the Server

```
TimeServer server = (TimeServer) Naming.lookup("//
    localhost/TimeServer");
```

is equivalent with:

```
Registry reg = LocateRegistry.getRegistry("
    localhost");
TimeServer server = (TimeServer) reg.lookup("
    TimeServer");
```

113

Compiling the Programs

in src/server:

```
javac -d ../../bin sprc/rmiex/server/pub/Format.java
javac -d ../../bin sprc/rmiex/server/pub/TimeServer.java
javac -d ../../bin sprc/rmiex/server/TimeServerImpl.java
```

in src/client:

```
javac -d ../../bin -classpath ../../bin/
            sprc/rmiex/client/Client.java
```

If JDK < 1.5, in the bin directory:

```
rmic sprc.rmiex.server.TimeServerImpl
```

114

Compiling the Programs

A java.policy file must be created in the bin directory:

```
grant {
   permission java.net.SocketPermission "*:1024-65535",
     "connect,accept";
   permission java.net.SocketPermission "*:80",
     "connect";
};
```

Running the server (from the bin directory):

```
java -Djava.security.policy=java.policy
     -Djava.rmi.server.codebase=file:/d:\prj\rmiex\bin/
     sprc.rmiex.server.TimeServerImpl
```

Running the client (from the bin directory):

```
java -Djava.security.policy=java.policy
     sprc.rmiex.client.Client
```

115

Design Patterns and RMI

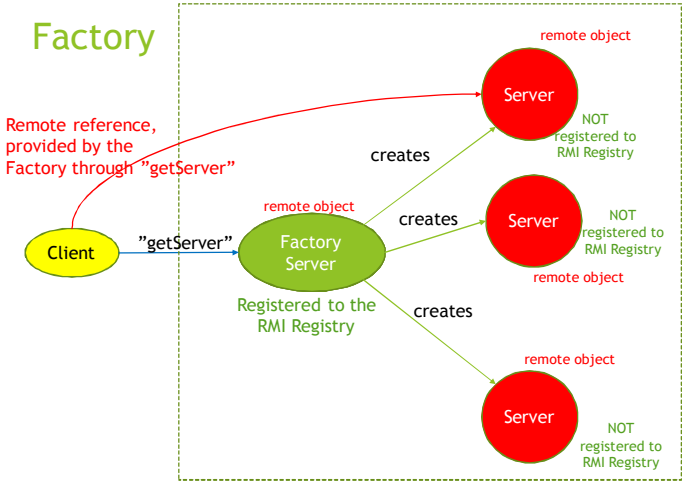
116

Object-Oriented Programming

- RMI provides a means of developing distributed applications by directly using language-specific constructs
- Remotely accessible objects can be created and their references passed between objects in the application
- Therefore: the usual OO design patterns can be easily applied
- Common patterns:
  - Factory
  - Observer

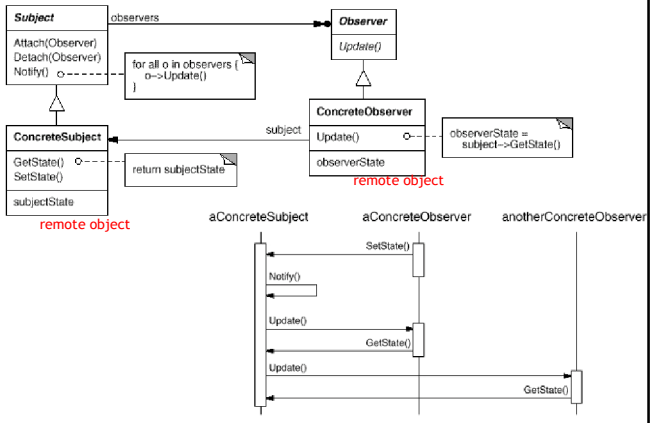
117

Factory



118

Observer



source: Design Patterns

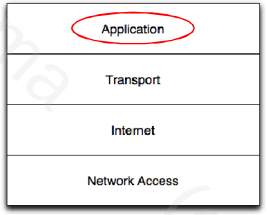
119

Application-level TCP/IP communication

120

The TCP/IP protocol stack

- Applications use the transport-level protocols to communicate
- Two useful protocols:
  - TCP
  - UDP



Communication is done through primitives known as "sockets"

Connection-oriented and connectionless

Two ways of communicating over the network:

- A semi-permanent communication channel is established at the beginning of the communication. Subsequent sending does not need to specify destination addresses:
  connection-oriented communication
- No established link is created; at each send, the parties must specify the destination address:
  connectionless communication

TCP – Transmission Control Protocol

- Connection-oriented
  - a bi-directional connection is created
- Reliable
  - message acknowledgement
  - retransmission
  - timeout
- Ordered
  - messages are received in the same sequence as transmitted

UDP – User Datagram Protocol

- Connectionless
  - transmission does not verify the readiness of the receiver
- Unreliable
  - messages can be lost
  - no retransmission, no timeout control
- Unordered
  - messages are not necessarily received in the same sequence as transmitted

TCP/IP Addressing

125

Address

A construct used for locating and/or identifying an hardware or software entity in a network

126

Address spaces

- The totality of addresses generated using a certain specified pattern
- Linear address space
  = no hierarchical information is contained within the address
  Example: MAC addresses
- Hierachical address space
  = addresses contain information that place the locations in hierarchies
  Example: postal addresses

127

Addressing at the Network Level (IP)

IP Address

- IPv4
  - 32 bits
  - usually represented in the “dotted decimal” form: 193.226.12.13
- IPv6
  - 128 bits

128

Address classes

- The IPv4 address space was partitioned in several classes, of which the most important are:
- Class A:
  - first 8 bits represent the network address, 24 bits: host address
  - first bit is 0
  ⇨ 128 networks, each with 2^{24} hosts
- Class B:
  - first 16 bits represent the network address, 16 bits: host address
  - first two bits are 10
  ⇨ 16384 (2^{14}) networks, each with 65536 (2^{16}) hosts
- Class C:
  - first 24 bits represent the network address, 8 bits: host address
  - first two bits are 110
  ⇨ 2 097 152 (2^{21}) networks, each with 256 (2^{8}) hosts

129

Reserved spaces

- 0.0.0.0-0.255.255.255 – reserved
- 10.0.0.0-10.255.255.255 – private addresses, according to RFC 19184
- 127.0.0.0-127.255.255.255 – loopback addresses, internal to the TCP/IP stack;
- 172.16.0.0-172.31.255.255 – private addresses, according to RFC 1918
- 192.168.0.0-192.168.255.255 – private addresses, according to RFC 1918

130

Address classes – too rigid

- The address classes proved to be impractical, especially in what regards classes A and B
- Reason: too many host addresses to be managed in a same single network
- Some organizations may receive a too large address space, they will never fully use
- Solution: sub-netting
  = Dividing the network, and consequently its network address, in several sub-networks

131

Network Mask

- Each network address is associated a (sub-)network mask
- Network mask:
  - the first n bits depict the network address, and are set to 1; all the rest are 0
  Therefore the network masks for the address classes are:
  - Class A: 255.0.0.0
  - Class B: 255.255.0.0
  - Class C: 255.255.255.0

132

Network mask

- Using the network mask:
  To find out whether an address belongs to a certain network, apply a bitwise AND operation between the address and the network mask. All addresses that give equal results belong to the same network
- Example -- a class C network:
  193.226.12.13 & 255.255.255.0 = 193.226.12.0
  193.226.12.235 & 255.255.255.0 = 193.226.12.0

  the same network address

  11000001111000100000110000001101 &
  11111111111111111111110000000000
  11000001111000100000110000000000

Describing networks addresses

- The addresses belonging to a network can be easily described as a pair containing
  - the network address
  - the network mask

Examples:

193.226.12.13 / 255.255.255.0, or 193.226.12.13/24*
- an IP address

172.16.0.0 / 255.240.0.0, or 172.16.0.0/12
- a network address

*CIDR ("Classless Inter-Domain Routing") notation, the number after the slash counts the non-zero bits at the beginning of the network mask

Sub-netting

How sub-netting is done:

"Borrow" adjacent bits from the (most significant part of the) host address, and use it for the sub-network address

- Borrowing 1 bit will create 2 sub-networks
- Borrowing 2 bits will create 4 sub-networks
- …

Sub-netting example

- A class C network, with the network address 192.168.1.0 (netmask: 255.255.255.0)
- A new netmask: 255.255.255.192
  192 = 11000000 ➞ we "borrowed" 2 bits

Therefore, we have created 4 sub-networks within the above class C network:

- 192.168.1.0,
- 192.168.1.64,
- 192.168.1.128,
- 192.168.1.192

```
  0 = 00000000
 64 = 01000000
128 = 10000000
192 = 11000000
```

Where are IP addresses used?

An IP address is associated, at the network level, with a network interface

- A computer may have several IP addresses
- Addresses may be associated with physical interfaces, as well as with logical ones

Local host address:
- 127.0.0.1, reserved, as part of the loopback class of addresses (127.0.0.0/8)

137

Ports

- A port is a number depicting, at the transport layer, a communication endpoint in a computer
- Uniquely identifies, within a computer, a process or application that is able to communicate through the network
- Used in conjunction with the IP address The TCP/IP ports are 16-bit numbers
- Different transport protocols may use the same port number without interfering with each other (e.g., TCP and UDP can simultaneously use a given port k)

138

Ports

- Ports 0-1023 are typically privileged (only super-user processes can register them)
- Ports 1024-65535 can be freely used by applications
- Only one process on a same OS can use a given port at a time
- Some ports are “well known”, usually (but not mandatorily) assigned to common application-level protocols. Examples:
  - 80 – HTTP
  - 22 – SSH
  - 20, 21 – FTP
  - 23 – Telnet
  - 25 – SMTP
  - 110 – POP3
  - 53 – DNS
  - 143 – IMAP

139

IP and Port

Uniquely identify a communication endpoint (associated with a software component) over the network

140

Names, not numbers!

- To simplify usage, IP addresses can be translated to and from names belonging to a hierarchical namespace
- The translation is done using the DNS (Domain Name Service) protocol
- A DNS server registers the correspondence between names at a certain level in the hierarchy, and the corresponding addresses. DNS servers can be queried when needed
- DNS servers form a distributed system, and are organized in a hierarchical structure:
  - root-level servers: provide the addresses for the authoritative top-level domain (TLD) server
  - servers for the TLD: resolve the names within the assigned TLD: .com, .net, ,ro, …
  - subdomains are resolved in the same way, by a hierarchy of servers

141

Communicating through TCP/IP at the application level

142

Network Socket

= A primitive depicting a communication endpoint over a network

Defined by:
- the transport protocol used (TCP, UDP)
- the socket address (IP and port)

Network socket types:
- Stream sockets (use the TCP protocol)
- Datagram sockets (use the UDP protocol)
- Raw IP sockets (bypass the transport layer and expose the IP packet headers to the application)

Note: non-network sockets do exist, but they are outside the scope of this course

143

Sockets and Programming

Programs can create and use sockets via a socket API
- available on virtually all modern software platforms
- accessible from most programming languages
- A client-server model is commonly used

Internet sockets are usually based on the Berkeley (BSD) Sockets standard

The BSD Sockets API (written in C) defines:
- Network sockets
- UNIX Domain sockets (outside the scope of this presentation)

Other languages provide BSD sockets, usually by wrapping the C-language BSD Sockets API

144

BSD Sockets

C headers:
- <sys/socket.h>
  - Core functionality
- <netinet/in.h>
  - internet, address and protocol families
- <sys/un.h>
  - Local address family, not used on networks
- <arpa/inet.h>
  - Functions for manipulating IP addresses
- <netdb.h>
  - Functions name resolution (e.g. DNS)

145

BSD Sockets – main functions

int socket(int domain, int type, int protocol);
Client and server side. Creates a socket. Arguments:
- domain: AF_INET (IPv4), AF_INET6 (IPv6), AF_UNIX (non-network)
- type: SOCK_STREAM (TCP), SOCK_DGRAM (UDP), etc.
- protocol: IPPROTO_TCP, IP_PROTO_UDP

Returns: a new file descriptor representing the socket, or -1 on error

int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
Client and server side. Binds a socket to an address (IP and port). Mandatory for server-side sockets. Clients usually don't call this function except for the rare cases when a specific port is needed at the client end (e.g. because of a firewall restriction on outgoing ports)

int listen(int sockfd, int backlog);
Server side. Prepares a socket for incoming connections. Necessary only for stream (TCP) sockets.

146

BSD Sockets – main functions

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
Server side. Waits for an incoming connection. Returns a new socket when the connection is established. Only required for stream sockets (TCP).
The new socket:
- will be bound to the same port
- represents the communication endpoint with the client that initiated the connection

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
Client side. Connects to a server specified by IP address and port. If the client-side socket wasn't already bound to a port, an *ephemeral port* will be created and bound for this endpoint.
Used mainly for connection-oriented protocols (TCP)
For connectionless protocols, connect only sets the *default* destination address (to use with send(), as opposed to sendto())

147

Helper functions

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, int len, int type);

Server and client side. Support functions for translating to/from host names using DNS or other resolving methods (such as local /etc/hosts files)

148

Sending data

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Sends data through a connected socket. If the protocol is connectionless, the default destination is used, as set with connect().

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Send data to a specific address. If the socket is connection-oriented (TCP), the given destination address is ignored.

149

Receiving data

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Receives data from a connection-oriented socket.

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Receives data, usually from a socket. If not null, the source address is filled-in with the address of the sender.

150

Types of servers

- Iterative servers
  - Can serve only one client at a time
- Concurrent servers
  - Can serve multiple clients at a time
  - Use concurrency-specific primitives (e.g., processes, threads)

151

Example of iterative server

```
int sockfd, newsockfd;
if ((sockfd=socket(...)) < 0)  {
    printf ("error ..."); exit(1);
  }

if (bind(sockfd,...) < 0)  {
    printf ("error ..."); exit(1);
  }

if (listen(sockfd,5) < 0)  {
    printf ("error ..."); exit(1);
  }
```

```
for (;;)
{
 newsockfd = accept(sockfd, ...);
 if (newsockfd < 0)  {
     printf ("error ..."); exit(1);
   }

 process(newsockfd);
     /*handle the client*/

 close (newsockfd);
}
```

152

Example of concurrent server

```
int sockfd, newsockfd;

if ((sockfd=socket(...)) < 0){
     printf ("error ...");
     exit(1);
  }

if (bind(sockfd,...) < 0){
     printf ("error ...");
     exit(1);
  }

if (listen(sockfd,5) < 0){
     printf ("error ...");
     exit(1);}
  }

for (;;)
{
 newsockfd=accept(sockfd, ...);

if (newsockfd < 0) {
     printf ("error ...");
     exit(1);
   }

if (fork()==0)   {
     close(sockfd);
     process(newsockfd);
       /*handle the client*/
     exit(0);
   }

close (newsockfd);
}
```

153

More programming examples…

…in the lab support documentation

154

Deploying a classic distributed system

A case study

155

Installing a customized e-mail delivery system

156

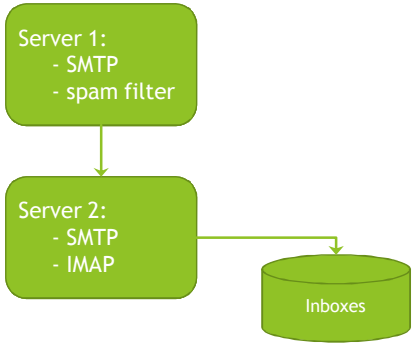
Purposes

A system fitted for small to medium-sized groups

- Receive external e-mail messages through SMTP
- Send local messages to the outside (SMTP)
- Filter messages that may constitute spam
- Provide standard inboxes
- Provide e-mail aliases and lists
- User-side message download (IMAP)

157

An architecture



158

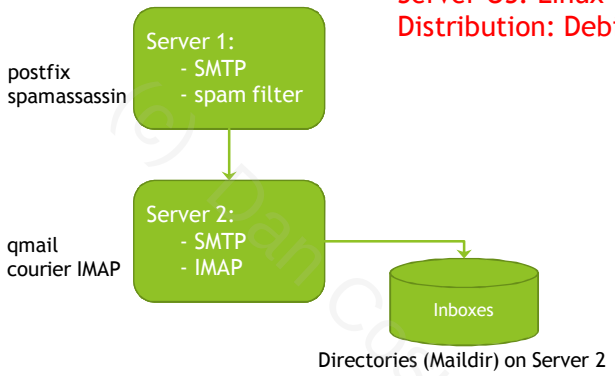
Software

- Many OS-es can be chosen
- There are several choices for the mail servers and the filtering software
- Choose from the variants best suited for the host OS/distribution

159

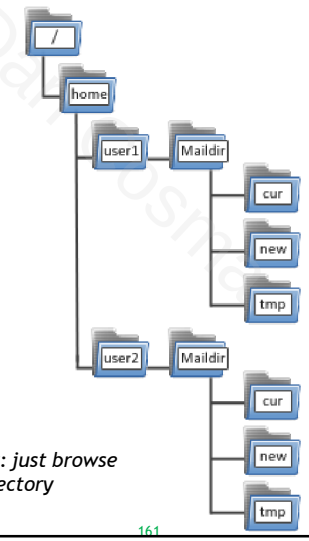
An architecture

Server OS: Linux
Distribution: Debian



160

Maldir

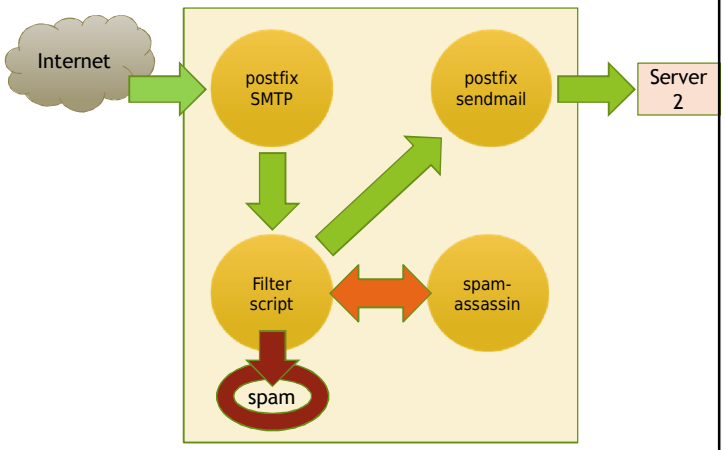


A note about privacy: just browse your own Maildir directory

Image: Wikipedia, attribution: GVdSteen at nl.wikipedia

161

Server 1 – process view



162

Postfix → filter script

/etc/postfix/master.cf:

[...]

```
# Original postfix-sendmail connection
# spamassassin unix -     n       n       -       -       pipe
#        user=spamd argv=/usr/bin/spamc -f –e
#        /usr/sbin/sendmail -oi -f ${sender} ${recipient}

# modified to directly remove some spam
spamassassin unix -     n       n       -       -       pipe
        user=spamd argv=/usr/local/bin/spamfilter.sh -oi –f
${sender} ${recipient}
```

163

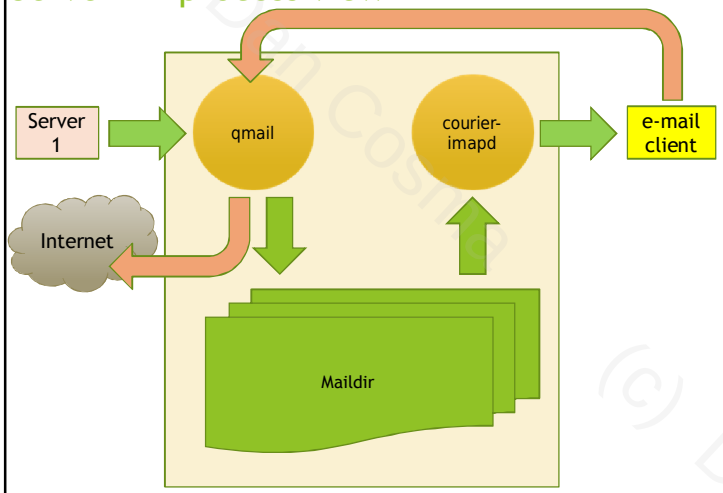
The filter script

/usr/local/bin/spamfilter.sh:

```
#!/bin/bash
# pipes mail to spamassassin and removes mails with high score
# source: linuxquestions.org, and many other places on the internet
SENDMAIL="/usr/sbin/sendmail"
EGREP=/bin/egrep
TMPFILE=/tmp/spamfilter.$$
SIDELINE_DIR=/var/spamfilter
# Number of *'s in X-Spam-level header needed to remove message:
SPAMLIMIT=10
# Clean up when done or when aborting.
trap "rm -f $TMPFILE" 0 1 2 3 15
# Pipe message to spamc and store in $TMPFILE
cat | /usr/bin/spamc | sed 's/^\.$/../' > $TMPFILE
# Are there more than $SPAMLIMIT stars in X-Spam-Level header?
if $EGREP -q "^X-Spam-Level: \*{$SPAMLIMIT,}" < $TMPFILE
then
  mv $TMPFILE $SIDELINE_DIR/`date +%Y-%m-%d_%R`-$$
else
$SENDMAIL "$@" < $TMPFILE
fi
#Postfix returns the exit status of the Postfix sendmail command.
exit $?
```

164

Server 2 – process view



165

qmail

- Modular design

Modules	Function
qmail-smtpd	accepts/rejects messages via SMTP
qmail-inject	injects messages locally
qmail-rspawn/qmail-remote	handles remote deliveries
qmail-lspawn/qmail-local	handles local deliveries
qmail-send	processes the queue
qmail-clean	cleans the queue

Source: http://www.lifewithqmail.org

166

qmail

- Main directories
  - /var/qmail/control
    - configuration files
  - /var/qmail/queue
    - mail queues
  - /var/qmail/alias
    - aliases (alternate mailbox names, lists, etc.)

167

/var/qmail/control

Configuration files

- badmailfrom – source (”From”) adresses to be rejected (blacklisted)
- defaultdelivery – the default destination for incoming e-mails

```
./Maildir/
```

- defaultdomain – default domain for this server

```
cs.upt.ro
```

- me – the hostname of this server

```
bigfoot.cs.upt.ro
```

168

/var/qmail/control

Configuration files

- locals - domains this server delivers locally

```
mail.cs.utt.ro
cs.utt.ro
aspc.cs.utt.ro
bigfoot.cs.upt.ro
mail.cs.upt.ro
cs.upt.ro
```

- defaulthost - the default host managed by this server

```
bigfoot.cs.upt.ro
```

- rcpthosts - domains this server is allowed to manage (accept mail for)

```
mail.cs.utt.ro
cs.utt.ro
aspc.cs.utt.ro
bigfoot.cs.upt.ro
mail.cs.upt.ro
cs.upt.ro
aspc.cs.upt.ro
```

- . . .

169

/var/qmail/queue

A directory structure containing the qmail message queues.

Examples:

- bounce - store the delivery errors
- mess - messages being sent
- info - envelope sender addresses
- remote - local envelope recipient addresses for messages being sent
- local - local envelope recipient addresses for messages being sent
- lock - lock files

170

/var/qmail/alias

A directory structure containing various aliases in the ".qmail" format, implementing alternate e-mail names, e-mail lists, and even complex commands.

Examples:

- .qmail-dan:cosma

```
danc
```

- .qmail-staff

```
dan.cosma
petru.mihancea
marius.minea
[...]
```

- .qmail-filtered:address

```
|/var/qmail/alias/access_filter
```

171

User .qmail files

On qmail systems, users can create a .qmail file in their home directories, containing ".qmail" format specifications for forwarding, filtering, etc.

Example:

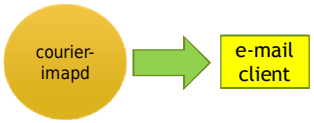
- /home/myuser/.qmail

```
./Maildir/
mygmailaddress@gmail.com
```

172

Retrieving e-mail

- E-mail clients can be configured to get the e-mails through various protocols
  - POP3 – Post Office Protocol: receives complete e-mails and usually deletes them from the server
  - IMAP – Internet Message Access Protocol: clients usually leave the messages on the server; IMAP servers provide structured mail storage (folders)



- Courier IMAP is able to use Maildir folders for storage

173

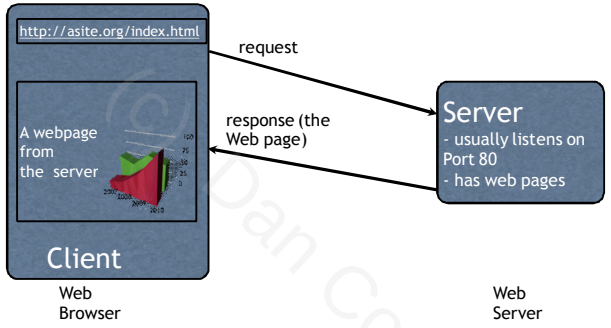
Java Server Pages

174

JSP – a technology for developing Web Applications in Java

175

The "traditional" Web



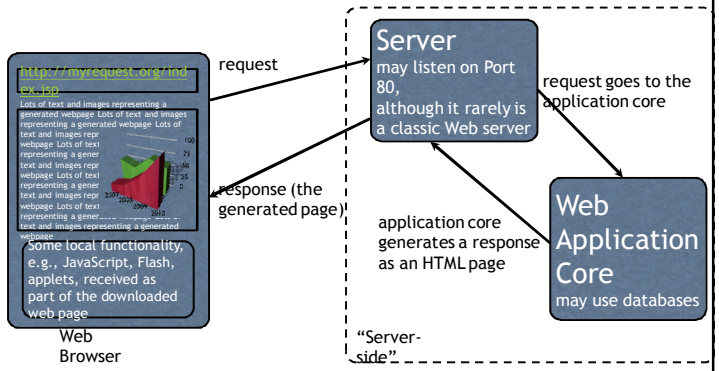
176

Web Applications

- Use the HTTP protocol
- The client is always the generic Web browser
- Traditionally, the functionality is almost entirely server-side
- The browser sends HTTP requests, the server generates HTML pages as response
- Session management is complex (HTTP is a bit primitive)
- Client-side functionality requires “imaginative” workarounds (now available as dedicated frameworks)

177

The anatomy of a Web Application



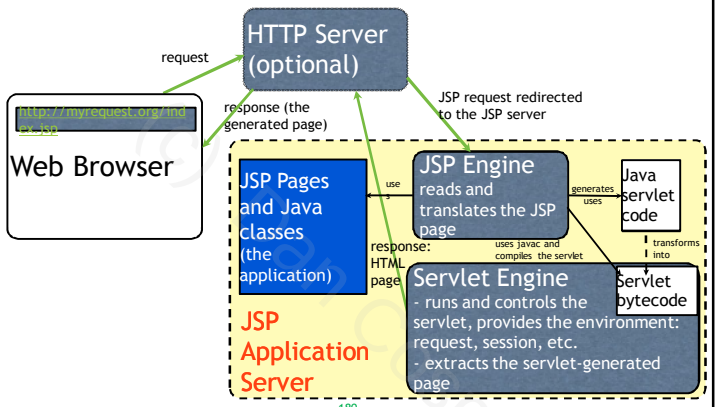
178

JSP

- Java Server Pages
- Technology that builds on Java Servlets
- The container provides support for Web applications
- Easy and quick development of dynamic Web sites

179

A JSP Application



180

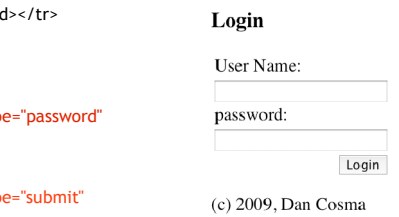
An Example

- Managing a simple login page in JSP
- Excerpt from a slightly more complex example application [1]

[1] Dan C. Cosma, Programarea aplicatiilor distribuite, Editura de Vest, Timisoara, 2009, ISBN 978-973-36-0501-0

The Main "JSP"

```
<h3>Login</h3>
<form name="login" action="loginAction.jsp" method="post">
<table>
        <tr>      <td>User Name:</td></tr>
        <tr> <td><input type="text"
name="userName"/></td> </tr>
    <tr><td>password:</td> </tr>
        <tr><td>
                        <input type="password"
name="password"/>
        </td></tr>
        <tr><td align="right">
                        <input type="submit"
name="add" value="Login"/>
        </td></tr>
</table>
</form>
(c) 2009, Dan Cosma
```



The "Action" JSP

```
<%@ page language="java"
import="java.lang.*,java.util.*" %>
<%
String userName = request.getParameter("userName");
String password = request.getParameter("password");
if(userName == null)
  { %>
   <p> Please login first.
  <%
  }
else
  { %>
   Congratulations <%=userName%>! You are logged in with the password: <%=password%>.
<%
  } %>
<p> <hr> <p>
<form name="goMain" action="index.jsp" method="post">
  <input type="submit" name="go" value="Return to main menu"/>
  <!-- Forwarding the username -->
  <input type="hidden" name="userName"
value="<%=request.getParameter("userName")%>"/>
</form>
(c) 2009, Dan Cosma
```


Comments

- JSP pages mix Java with HTML
  This may lead to unmaintainable code (hard to read and understand)
- JSP pages should be small, and contain as little Java as possible
- Use separate classes for the main Java functionality
  As the JSP will in fact transform into a Java class (the servlet), you can call other classes from within the JSP
- Use a layered model to separate functionalities/concerns
  In fact, design the program using all the "general" design best practices you are already familiar with

Deployment

- JSPs along with the helper classes are packaged in a standard format (E.g. a *.war* archive with a well-known structure)
- The package is deployed to the container
  This operation is dependent on the chosen JSP application server variant
- The deployment should follow the rules specified by the application server
- The deployment process should be automated
  You should also avoid deploying the application using IDE-specific plugins. The customer doesn't have to install Eclipse to make your program work.
  Moreover, you don't want to depend too much on a plugin that does "magical" things behind the scene. They will certainly fail you at a point.

185

Message-Oriented Infrastructures (Message-Oriented Middleware)

Java Message Service

186

Messaging system

- A peer-to-peer facility enabling clients to send and receive messages to each other
- The messages are sent to an agent that intermediates the communication
- A messaging system enables loosely coupled communication between the components (senders and receivers)

Note: messaging systems are NOT e-mail or chat applications! They deal with the communication between software components

187

Messages

- The applications communicate by passing messages to each other
- A message is a structured data entity that basically consists of
  - a header
  - properties (optional, JMS)
  - body

188

Messaging domains

- Describe the messaging exchange model
  - Point-to-point
  - Publish-subscribe

189

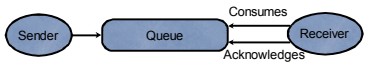
Point-to-point

- The destination of the messages is clearly specified
- Gravitates around the concept of message queues
- Senders send messages to a specific queue, thus specifying the intended receiver
- Receivers monitor their respective queues and consume the messages

190

Point-to-point

- A message is consumed only by one receiver
- The sender does not wait for the receiver
- The receiver acknowledges the successful processing of the message

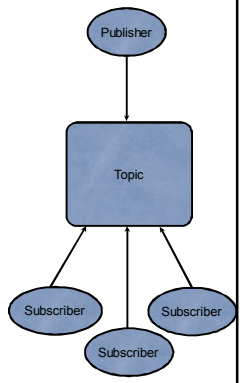


Messages can be consumed both synchronously and asynchronously

191

Publish-subscribe

- The message is sent to a shared resource by a publisher client (the equivalent of a sender)
- Multiple receivers, called subscribers *may* consume the message
- Subscribers specify the messages they are interested in, by describing message filters
- The message consumption can be done both synchronously and asynchronously



192

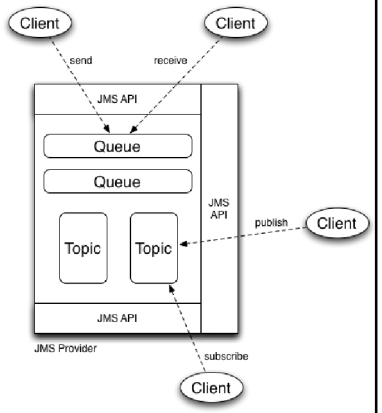
Java Message Service (JMS)

- A specification that enables the implementation of message services in the Java environment
- Unifies the messaging functionality under a single, consistent specification
- JMS in not a service in itself, it is only adhered to by particular implementations
- The implementations (the actual services) are called JMS providers

193

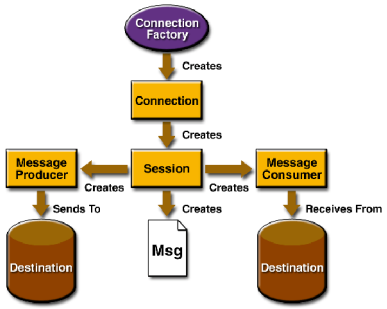
One architecture, two messaging domains

- A JMS provider provides two types of resources ("destinations")
  - message queues
  - topics



194

JMS API Programming Model



source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

195

Administered objects

- Connection Factories and Destinations
- Are managed administratively, rather than programatically
- The administrative details vary from vendor to vendor (providers)
- The access to the resources is done through portable interfaces
  -> clients are easily adapted to different providers

196

Connection Factories

- Create a connection with a JMS provider
- Two types defined in J2EE:
  - QueueConnectionFactory
  - TopicConnectionFactory

Creating and connecting to the factories

```
$ j2eeadmin -addJmsFactory jndi_name queue
$ j2eeadmin -addJmsFactory jndi_name topic
```

```
Context ctx = new InitialContext(); //get the JNDI context; searches
        //the classpath for a vendor-specific jndi.properties file

QueueConnectionFactory queueConnectionFactory =
  (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");

TopicConnectionFactory topicConnectionFactory =
  (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

JMS Destinations

- A Destination specifies the target/source of the messages: queues or topics
- Destinations are created through administration:
  j2eeadmin -addJmsDestination queue_name queue
  j2eeadmin -addJmsDestination topic_name topic
- Clients can connect using the standard API:
  Queue myQueue = (Queue) ctx.lookup("MyQueue");
  Topic myTopic = (Topic) ctx.lookup("MyTopic");

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Connections

- Represent the connection with the JMS provider
- Two types: QueueConnection, TopicConnection

```
QueueConnection queueConnection =
  queueConnectionFactory.createQueueConnection();
...
queueConnection.close();

TopicConnection topicConnection =
  topicConnectionFactory.createTopicConnection();
...topicConnection.close();
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Sessions

- A session represents a single-threaded context that produces or consumes messages
- Provides support for transactions
- Serializes the execution of message listeners
- Two types: QueueSession, TopicSession

```
TopicSession  topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);//non-transacted, automatic
                          //message acknowledgement
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

201

Message Producers

- Produce messages that are sent to a Destination
- Two types: QueueSender, TopicPublisher

```
QueueSender queueSender = queueSession.createSender(myQueue);
TopicPublisher topicPublisher = topicSession.createPublisher(myTopic);

...
queueSender.send(message);
...
topicPublisher.publish(message);
...
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

202

Message Consumers

- An object capable of receiving messages
- Two types: QueueReceiver, TopicSubscriber
- The message consumption can be done:
  - synchronously
  - asynchronously
- Topic subscribers can be made durable (can receive messages that occurred when they were inactive)

203

Synchronous message consumption

```
QueueReceiver queueReceiver =
queueSession.createReceiver(myQueue);

TopicSubscriber topicSubscriber =
topicSession.createSubscriber(myTopic);

queueConnection.start();Message m =
queueReceiver.receive();topicConnection.start();Message m =
topicSubscriber.receive(1000); // time out after a second
```

- Messages are not delivered until the connection is started

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

204

Asynchronous message consumption

- To receive messages asynchronously, the application can define message listeners
- A listener implements the MessageListener interface:

```
public interface MessageListener {
   public void onMessage(Message message);
}
```

- The listener is associated with a consumer

```
TopicListener topicListener = new
TopicListener();topicSubscriber.setMessageListener(topicListener);
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

205

Message Selectors

- Can be used for filtering the messages that arrive to a consumer
- The filtering is done by the JMS provider, not by the application
- The selectors are specified as statements in a subset of SQL92 conditional expression syntax
- Selectors can be passed as arguments to the createReceiver, createSubscriber, and createDurableSubscriber methods

206

Messages

- A message consists of: header, properties, body
- There are 5 types of messages defined by the API:
  - TextMessage: the body is a text (e.g. XML)
  - MapMessage: a set of name/value pairs
  - BytesMessage: a stream of bytes
  - StreamMessage: a stream of primitive Java values, filled and read sequentially
  - ObjectMessage: a Serializable object

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

207

Cloud Computing

208

Cloud Computing

- = software services that do not require the user's involvement in the deployment, configuration and hosting
- Based on the *utility computing* model
  = computing services are like public utilities, similar to the traditional ones (gas, water, electricity, etc.)
- Cloud computing providers expose their services online
- Users need minimal resources to connect as clients

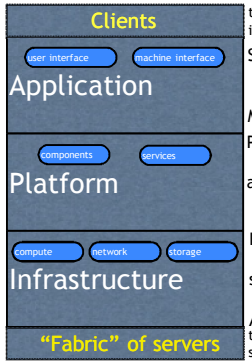
209

Cloud Computing

- The services are usually available via
  - Web applications
  - Web Services
  - APIs
- Examples of services:
  - storage
  - e-mail, communication, social networking
  - office tools
  - virtual servers
- Services are completely controlled by the vendor, and clients pay per usage
  - resembles the mainframe-based model

210

The Cloud Computing Stack



the hardware or software that completely relies on cloud services and is essentially useless without them

Software as a Service (SaaS)
- the application, provided as a service
- examples: Google Apps, Office Web Apps, Apple MobileMe

Platform as a Service (PaaS)
- the computing platform provided as a service, facilitates application deployment, complete development of services
- examples: Microsoft Azure Services Platform, Amazon Web Services, Google App Engine

Infrastructure as a Service (IaaS)
- the computer infrastructure, e.g., virtual servers, storage space
- examples: Amazon Elastic Compute Cloud (EC2), Amazon Elastic Block Store, Amazon Machine Image

the hardware and software infrastructure built to deliver the cloud services: processing units, specialized cloud operating systems, etc.

211

sources: Wikipedia, samj.net

Cloud Computing

- Advantages:
  - users can easily expand or modify the resources, as their needs change
  - reduced costs for user (debatable)
  - reduced maintenance costs for providers:
    the services are in one place, easy to support or improve
  - location independence
    all services are reachable wherever you are
  - *multi-tenancy* allows for efficient resource sharing among users
    multi-tenancy: a principle in software architecture where a single *instance* of the software runs on a server, serving multiple client organizations (tenants)

212

source: Wikipedia

Cloud Computing

- Issues:
  - Privacy concerns
    all data and business logic is stored on the vendor's data centers
  - Security concerns
    although the security can be addressed in a centralized and efficient way by the vendor, the users essentially lose control on their sensitive data
  - Reliability depends on the vendor
    however, major vendors use multiple redundancy to address this issue
  - Availability concerns
    some providers did cease to exist due to various reasons

213

Service-oriented systems

214

Service-Oriented Architectures

- Designed around the concept of providing *services*
- Service = *"an act or a performance offered by one party to another"* (Lovelock et al., 1996)
- Web Service = *"a standard representation for some computational or informational resource that can be used by other programs"* [1]

For references, see Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006
[1] Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006

215

Service

- Services can be provided for various users or organizations
- An application can use various services, from distinct providers
- The act of providing the service is independent of the application that uses the service (Turner et al., 2003)

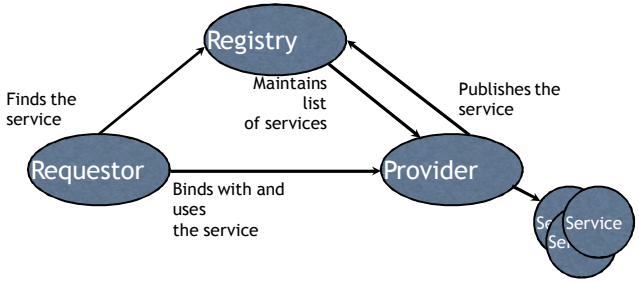
For references, see Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006

216

Service Interface

- To provide a service, an organization must define and publish a *service interface*
- *Service interface* = a definition that specifies
  - the information provided by the service
  - the methods for accessing the data
  - the parameters needed when using the service
- The interface must be expressed so that
  - the purpose of the service is clear
  - the service usage is unambiguous
  - the results and side effects are clearly described

Service-Oriented Interaction



Adapted from Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006

Standards

- Service orientation can provide interoperability, but only if built around standards
- Web Services standards:
  - SOAP (Simple Object Access Protocol)
    >> defines how structured objects are exchanged (including their structure)
    >> relies on other application protocols such as HTTP
  - WSDL (Web Services Description Language)
    >> XML-based, defines how web service interfaces are represented
  - UDDI (Universal Description, Discovery and Integration)
    >> platform-independent XML-based registry for Web Services

Service-Oriented Architectures

- Differences from Distributed Objects
  - services can be offered by any provider, even third-party
  - service specification is public, any authorized user can use the service without negotiating with the provider
  - services can be created or discovered dynamically
  - applications can choose dynamically one of several similar services
  - the application can change/choose what types of services it uses at different stages in its runtime evolution or when the environment changes

Adapted from Ian Sommerville, Software Engineering, 8th edition, Addison-Wesley, 2006

Web Services

221

Web Services

- Definition (W3C):

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Source: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

222

What Is a Web Service?

- A component of a distributed application:
  - self contained and self described
  - accessible through the network
  - communicating using standardized protocols
  - discoverable by other parties through various methods
  - data exchange format is usually XML

223

Web Services

- Types of Web services:
  - arbitrary Web services, in which the service may expose an arbitrary set of operations
  - REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations

Source: http://www.w3.org/TR/ws-arch/#relwwwrest

224

Technologies

- SOAP - the communication protocol
- WSDL - the service description
- UDDI - the service discovery
- XML, JSON - data format

225

SOAP

- The current development of SOAP defines two acronym expansions:
  - Simple Object Access Protocol - a message represents a remote method invocation (using the SOAP RPC representation)
  - Service-Oriented Architecture Protocol - the message represents the information passed to/from a service in a loosely-coupled, message-based, service architecture

226

SOAP

- Defines the communication protocol and XML data formats for exchanging messages
- The structure of a SOAP message:
  - Envelope - identifies the XML as a SOAP message
  - Header - application-specific
  - Body - invocation or response information
  - Fault - errors, status information

227

WSDL

- Web Services Description Language
  - XML-based
  - Describes the Web service as a collection of operations (methods) exposed publicly
- The WSDL 2.0 document elements:
  - Service - the container
  - Endpoint - service location (e.g. an URL)
  - Binding - specifies the interface and the SOAP binding style (Document/RPC)
  - Interface - defines the service, its operations and messages
  - Operation - the exposed methods
  - Types - the type of the data

228

"Classic" Web Services

- Operations are specified freely, as application-specific constructs (e.g. "getCustomerData")
- Services may be located through UDDI nodes
- Services are described through WSDL
- SOAP is used as RPC or as message-orientation support

229

REST

- Representational State Transfer
- An architectural style suited for the Web
- Introduced by Roy Fielding in 2000
  http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

230

REST

- The REST philosophy:
  - Design a Web service focusing on system *resources*
  - A resource is identified by an URI (Uniform Resource Identifier)
  - A representation of a resource is a document capturing the state of the resource
  - Clients and servers exchange resource representations
  - Client requests to the servers are made when a transition to a new state is required
  - REST services are stateless

231

REST

- Defines a set of architectural principles for Web services:
  - Use HTTP methods as they were designed
  - The service is stateless
  - Resources are organized in a directory-like structure of URIs
  - Transfer XML, JSON (JavaScript Object Notation), or both

Source: https://www.ibm.com/developerworks/webservices/library/ws-restful/

232

Use HTTP Methods

- REST applications directly map their operations on the standard HTTP methods:
  - To create a resource on the server: POST
  - To retrieve a resource: GET
  - To change the state of a resource: PUT
  - To delete a resource: DELETE
- Example: instead of GET /adduser?name=Robert HTTP/1.1
  use

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>  <name>Robert</name> </user>
```

Source: https://www.ibm.com/developerworks/webservices/library/ws-restful/

233

REST is Stateless

- The service does not store state information
- When making a request, the client presents the server with all the necessary state information so that the request can be fulfilled
- Improves scalability, simplifies the design

Source: https://www.ibm.com/developerworks/webservices/library/ws-restful/

234

REST: URIs as Directories

- Resources should be represented analogous to a directory structure
- The URIs should be as simple and intuitive as possible, should be lowercase-only
- Examples:
  http://www.myservice.org/discussion/topics/computers
  http://www.myservice.org/discussion/topics/science/threads
  http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}

Source: https://www.ibm.com/developerworks/webservices/library/ws-restful/

235

REST: Data Transfer

- The representations of the resources represent the resource state and attributes (a “snapshot” in time for that specific resource)
- Clients receive the representations upon request
- They can use XML, JSON or other structured formats

Source: https://www.ibm.com/developerworks/webservices/library/ws-restful/

236