Architecture of Distributed Systems
2011/2012

Interaction styles

Johan Lukkien

Chapter 4, 8.3 and 10.3, Tanenbaum & van Steen, *Distributed Systems, Principles and Paradigms*
Style examples

• Architectural
  – Dataflow
    • Batch sequential
    • Pipes & Filters
  – Independent components
    • Client-Server - REST
    • Publish & subscribe, broker
    • Peer to Peer
    • Service oriented
  – Shared memory
    • Blackboard, Whiteboard
    • databases, hypertext
    • Model-View-Controller
  – Virtual machines
    • Layering (also: independent components)
    • Interpreter
    • Mobile agents
    • Rule-based

• Interaction
  – procedure/method call
  – message passing
  – streaming
  – active messages
  – remote procedure call
  – remote method invocation
  – files
  – eventing

• Note: interaction styles induce some architectural elements
  – e.g RPC infra structure, client / server libraries, message queues
Interaction styles and views

• Interaction styles pertain to communication aspects in the views
  – in contrast to e.g. the subsystem part of the development view, where you find typically inclusion and dependence relationships

• However, further detailing interaction styles yields new components in all views
  – architectural elements supporting the style
  – often part of a framework for the interaction style

4+1 View Model [Kruchten 95]
Qualities of Interactions (communication)

- **Memory/storage**
  - *transient*: interaction requires sender and receiver to ‘execute’ at same time
  - *persistent*: interaction (data) remains while sender and receiver disappear

- **Units of information**:
  - *discrete*: structured units, independent and complete
  - *streaming*: basic units; no further communication structure

- **Synchronization**
  - *asynchronous*: sender/caller does not wait or block;
  - *synchronous*: caller blocks till interaction acceptance
  - *buffered*: limited difference between #calls and #responses

- **Connection**
  - Connection oriented
  - Connection-less

- **Reliability**

- **Time dependence**
  - temporal relationships typically with streaming
  - *synchronous*: bounded delay
  - *isochronous*: bound minimum and maximum delay (i.e., jitter)
Persistent communication

A sends message and continues
A stopped running

A

B

B is not running
B starts and receives message

Time

(a)

A sends message and waits until accepted
Message is stored at B's location for later delivery

A stopped running

B

B is not running
B starts and receives message

Accepted

Time

(b)
Transient communication

A sends message and continues

Message can be sent only if B is running

B receives message

Send request and wait until received

Request is received

ACK

Running, but doing something else

Process request

Send request and wait until accepted

Request is received

Accepted

Running, but doing something else

Process request

Send request and wait for reply

Request is received

Accepted

Running, but doing something else

Process request
Agenda

• Layered protocols and middleware
• (Remote) Procedure calls
• (Remote) Method invocation
• Message Oriented Communication
• Streaming
Interaction starting point

• Details of interactions and interacting entities become visible mostly in the process and logical views

• Except for regular procedure/method calls, interaction requires independently executing entities
  – processes, threads
    • Communication on same machine goes via shared memory (see picture: several processors in the platform execute the processes/threads that share the available memory)
  – processes can execute on different machines
    • Communication between machines use a protocol stack
      – also possible for processes on same machine

28-Sep-11

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
Concepts: Process, Thread

- **Process**
  - “program in execution”
    - defines a data space
      - virtualization of the memory
    - defines ownership of resources
      - concurrency transparency
    - has at least one associated thread
    - unit of deployment (distribution)
      - together with a software component
    - unit of fault containment

- **Thread**
  - unit of concurrency
    - virtualization of the processor
  - unit of scheduling
    - though in case of one thread per process, the process is often said to be the unit of scheduling
  - operates in an address space
    - i.e., in a process; several threads may share this
  - has an associated execution state
    - place where it is in the code, stack image and return addresses of function calls, values of processor registers
Layered protocols: OSI('83) reference model

- Interfaces:
  - provided to layer above
  - required from layer below

- OSI protocols:
  - are in fact hardly used

- OSI reference:
  - widely accepted

Layers, interfaces, and protocols in the OSI model.
Although we may think IP is all there was...

Protocol Stacks in Relationship to the OSI Model

<table>
<thead>
<tr>
<th>OSI Layer</th>
<th>Apple Computer</th>
<th>Banyan Systems</th>
<th>DEC DECnet</th>
<th>IBM SNA</th>
<th>Microsoft Networking</th>
<th>Novell NetWare</th>
<th>TCP/IP Internet</th>
<th>Xerox XNS</th>
<th>OSI Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Layer 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation Layer 6</td>
<td>AppleTalk Filing Protocol (AFP)</td>
<td></td>
<td>Remote Procedure Calls (RPC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Remote Procedure Calls (RPC)</td>
<td></td>
</tr>
<tr>
<td>Session Layer 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transport Layer 4</td>
<td>AppleTalk Transaction Protocol (ATP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Layer 3</td>
<td>Datagram Delivery Protocol (UDP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Link Layer 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical Layer 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Application Programs and Protocols for file transfer, electronic mail, etc.

ISO 8073
ISO 8072
ISO 8072
ISO 8072
ISO 8072
ISO 8072
ISO 8072
ISO 8072
ISO 8072
ISO 8072

Network Interface Cards: Ethernet, Token-Ring, ARCNET, StarLAN, LocalTalk, FDDI, AATL, etc.
NIC Drivers: Open DataLink Interface (ODI), Network Independent Interface Specification (NIIS)
Issues addressed by the layers

• Basic network protocols:
  – Physical:
    • sends bits on medium (i.e. standardizes the electrical, mechanical, and signalling interfaces)
  – Data link:
    • detects and corrects errors in frames; deliver frames in one-hop neighborhood
  – Network:
    • send packets from sender to receiver machines using multihop routing
  – Transport:
    • breaks messages into packets; delivery guarantees; multiplexing ports

• Higher-level protocols:
  – Session:
    • provides dialog control and synchronization facilities (checkpoints);
  – Presentation:
    • structures information and attaches meaning ("semantics": e.g. names, addresses, amount of money);
  – Application:
    • network applications, including a collection of "standard" ones (email, file transfer)
Layered protocols and message layout

A typical message as it appears on the network. Note that only link layer messages ever “exist”
Layering and middleware

- Layers >4 build services on top of transport facilities
- A general definition of such services is *middleware*
  - going beyond simple transport / message passing
    - RFC 2768 mentions: remote execution, data management, resource management, security, mobility support, ...
  - therefore, the focus shifts from the protocol definition to the programming interface
  - ‘middleware’ also refers to a layer that hides distribution aspects

Network OS:
- delivers transport service

Programming interface:
- API for this service
Middleware: the hour-glass problem

- IP/everything tendency
  - goal in the layering: connect everything
  - UDP, TCP/IP as basic interoperability

- Application needs are diverse
  - use transport layer for private purpose
  - no direct need to support standards
  - hence, difficult to define generally accepted middleware layer
  - result: middleware libraries for many application domains

- Interaction style frameworks typically come with a middleware library (e.g. CORBA, UPnP)
Middleware tradeoff

- Two interfaces:
  - application-middleware
  - network protocol between devices

- Case (a): middleware merely provides message passing
  - using just IP-convergence
  - per application: application-dependent protocol
  - no sharing of logic

- Case (b): middleware provides advanced services
  - services define the (application independent) protocol between middleware layers
  - sharing of standard functions
Agenda

• Layered protocols and middleware
• (Remote) Procedure calls
• (Remote) Method invocation
• Message Oriented Communication
• Streaming
Procedure/method call

- Called function runs on ‘thread of control’ of caller
  - synchronous, discrete, most often transient (, connection-less)
  - synchronous call (with completion return) or signal (without return)

- This is the normal ‘mode’ of operation in regular programs (processes)
  - call to functions in same process, including libraries
  - implied support: call stack

- Extended to interaction across process and machine boundaries: remote procedure call
  - visible in the process and deployment views
  - implied support: infra structure for
    - establishing the remote call and return
    - binding the RPC reference, i.e., the function to be called on the remote machine
(Remote) procedure calls

Numbers refer to MS tool description
1. combined fragment (alt)
2. interaction operand
3. boundary
4. guard

from: Control Flow Analysis of UML 2.0 Sequence Diagrams, Vahid Garousi, Lionel Briand and Yvan Labiche
Conventional procedure call

Call: read(int fd, unsigned char *buf, int nbytes)

(a) Parameter passing in a local procedure call: the stack before the call to read.
(b) The stack while the called procedure is active.
Steps in calling an RPC

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
Steps in calling an RPC

1. Client procedure calls client stub in normal way
2. *Client stub* builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to *server stub*
5. *Server stub* unpacks parameters, calls server
6. Server does work, returns result to the stub
7. *Server stub* packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to *client stub*
10. *Stub* unpacks result, returns to client
Remote procedure call

• Typical in Client-Server style
• The concept aims at
  – access transparency
    • provide a procedural interface to remote functionality
    • difference local/remote not visible
    • .....except for non-local resources
    – typically, remote data
  – portability (of existing code)
  – reduce language/OS dependence
• Concerns of
  – scalability: many clients calling
  – reliability: increased dependence
    • independent failure calls for an elaborate solution to server failure
  – performance, see picture

28-Sep-11

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
Elements of a realization

- *Marshalling* of data
- Definition of the *RPC protocol*
- How to deal with *parameters*
- The development process to *include RPCs into a program*
  - frameworks
- RPC server infrastructure, server *discovery, binding time*
  - again, a framework
- Semantics under partial failures
- Synchronization (alternatives)
Need for marshalling

- Representations of numbers, characters, and other data items on machines may differ
  - The little numbers in boxes indicate the address of each byte

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>

(a)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>I</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

(b)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>

(c)

- Original message on the Pentium (“little endian”)
- The message after receipt on the SPARC (“big endian”)
- The message after simple inversion is still wrong

Hence, information about the meaning of the bytes in a message is needed.
Protocol

- Message format (see pictures)
- Data structure representation, e.g.
  - integers in two’s complement, chars in 16-bit Unicode, etc.
  - marshalling
- Message exchange sequence
  - e.g. acknowledge
- Binding to carrier
  - TCP, UDP, ....

```
foobar( char x; float y; int z[5] )
{
    ....
}
```

(a) A procedure
(b) Possible parameter packing into a message
RPC parameter passing

- Usually, value/result semantics
  - no reference parameters, no statics (globals)
  - ...lack of transparency

- Reference parameters:
  - address space problem
  - copy/restore
    - just copy for in- and restore for out-parameters
  - remote reference mechanism
    - ...additional RPC(s) for complex data structures?
    - ...migrate code towards data rather than using RPCs?
    - what to do with concurrency?

- For RPCs to be useful remote resources are typically required
  e.g. implicit state (file server), through library calls, or global references (URI)
Example: DCE RPC framework

• Distributed Computing Environment RPC
  – original design by the *Open Software Foundation* that became the *Open Group* since
  – MS implementation in Exchange/Outlook: MS-RPC
  – supposedly part of *Call of Duty*, multiplayer

• Framework
  – tooling for supporting RPC-based client & server design
  – run-time middleware services for supporting registration and discovery
Developing with DCE RPC

• Developer specifies the used interfaces in an interface definition language (IDL).
  – uniqueness guaranteed by a generated GUID
• An IDL compiler generates
  – a common include file
  – client and server stubs (code)
• Developer fills in the code for the functions in the IDL file
• The linker combines the resulting object file with stub and libraries
DCE runtime services: discovery, registration

- Server machines run a ‘daemon’
- A directory service stores (service, server) pairs

- A server process:
  - registers itself with the daemon and obtains an endpoint (a ‘port’)  
  - registers itself together with the identity of the machine on which it runs with the directory

- A client process:
  - finds an appropriate server machine
  - finds the server endpoint through the daemon
  - calls the RPC

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
28-Sep-11
Partial failure of RPC

- Important starting point: *make a failure model*
  - describe the behavior against which resilience is required
    - part of the problem specification, viz., the environment

- Failure model:
  - request loss, duplication
  - server crash
  - server response loss, duplication
  - client crash
    - may leave orphan tasks

- Response to these failures:
  - detection via timeout and explicit acknowledgement
  - make requests *idempotent*
    - multiple issue is identical to single issue
    - timestamping, dividing time into *epochs* (to indicate server restarts)
  - define semantics explicitly
    - at most once (DCE), at least once, exactly once
  - use timeout mechanism and ‘clean start’ for removing orphans
Asynchronous RPC (1)

- No need to wait if there is no result
  - terminology: ‘synchronous signal’, one-way RPC, asynchronous RPC

(a) Traditional RPC and (b) Asynchronous RPC

(a) Traditional RPC and (b) Asynchronous RPC
Deferred RPC (2)

- Do something useful while waiting
  - two asynchronous RPCs, 2\textsuperscript{nd} without result (‘one-way’)

A client and server interacting through two asynchronous RPCs.
Agenda

- Layered protocols and middleware
- (Remote) Procedure calls
- (Remote) Method invocation
- Message Oriented Communication
- Streaming
Plumbing: proxy, adapter, stub, skeleton and broker

• Proxy:
  – a component that acts on behalf of another component, implementing the same interface
  – capable of implementing filtering policies (e.g. which requests to pass on) and sometimes caching
  – typically, ‘proxy’ refers to a client side (representing the client); a ‘reverse proxy’ is placed at a server side

• Stub:
  – (originally): an empty interface implementation
  – (RPC/RMI): client stub: transparently implements an interface for a remote object; responsible for the messaging (also called: proxy)
    server stub (or skeleton): transparently perform the calls of a client stub and handles the messaging

• Example:
  – a HTTP proxy server acts on behalf of a user and filters requests
Object Adapter or Object Wrapper:
- a component that relays calls to an object interface and manages it
  - typically implementing different management policies for the object, e.g. creation policy, multi-threading, perhaps transient/persistent
  - converting between interfaces
  - possibly state holding for that conversion

Broker
- a component that handles and translates calls (messages) between two or more parties, and that manages the binding between references and objects.
  - This binding can be dynamic, based on interface inspection.
  - This dynamic binding becomes visible by explicit invocation (instead of a transparent method call) “RMI (object ref, method name, parameters)”
Distributed objects

General organization as typically supported by a framework. The proxy can be generalized to a broker as in the CORBA case.

CORBA organization based on an IDL (simplified picture, without services)
Communication is by the IIOP, Internet Inter-ORB Protocol

picture from wikipedia
Remote object invocation and parameters

- **RMI**: very similar to RPC
- **Difference:**
  - an object represents a context (a state) together with operations on it
    - objects can thus be persistent or transient
  - clear separation between interface and implementation (~ SOA)
    - actual implementation is hidden, can be a program in any language
  - references to objects occur naturally in a program

- **Parameters,**
  - pass local objects by value/result
  - and remote objects by reference

The organization to implement "**RMI (C, obj.method (L1, R1))**"
Run time system

- Distributed objects are supported by (services of) a run-time system
  - management: creation/destruction of objects, storage
  - communication: inspection of interfaces, binding, invocation

- Java Enterprise Beans:
  - 4 types of objects
    - session beans
      - stateful or stateless
    - entity beans (persistent)
    - message-driven beans
  - JMS: Java Message Service
  - JNDI: Java Naming and Directory Interface
  - JDBC: Java DataBase Connectivity
  - RMI: Remote Method Invocation
Agenda

• Layered protocols and middleware
• (Remote) Procedure calls
• (Remote) Method invocation
• Message Oriented Communication
• Streaming
Message Oriented Communication

- RPC & RMI are rather synchronous
  - in time: wait for reply
  - in space: shared data is known
  - functionality and communication coupled

- Look for communication models with better decoupling and other qualities
  - message oriented: communication and functionality separated
    - more abstract? or more basic?
Sockets

- Sockets with TCP give a point-to-point byte oriented transport service
- The TCP service interface has some 8 functions and two roles
  - client, server
- The quality of the service:
  - transient, discrete, connection oriented, buffered
  - reliable
    - against packet loss duplicates, and reordering
    - traded for delay
- Rather basic interface
  - just send/receive
  - basis for more advanced middleware services (see: protocol triangle)
- Note: there is also an unreliable datagram service: UDP

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
Setup / communicate / destroy

- Client needs to know the transport-level address of the server
- end-to-end connection, characterized by
  \[((Client, ClientPort), (Server, ServerPort))\]

Connection setup using sockets
Message queuing systems

- **Message queuing systems**
  - **guarantee delivery**
    - no guarantee *when* and whether or not handled
  - **persistent, discrete, asynchronous**
    - limited synchronization, e.g. upon handing off to the MQ system
    - perhaps *controllable* synchronization
    - persistence gives freedom to sender and receivers to go offline
  - **simple interface**
    - note that the primitives use *queue* references, which makes them more abstract
    - in practical cases more management is possible (message and queue sizes, ordering, ....)

- **Examples:**
  - email (application)
  - IBM’s WebSphere

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>
General architecture of queuing systems

• Use (unique) *logical* names for message *queues* (top)
• The queuing layer forms an overlay
  – need to map queue names to transport addresses

• For scalability, the overlay is extended with routers (bottom)
  – scalability: why?
    • the complexity of maintaining a global overview
  – also allows for secondary processing (e.g. logging) and scalable multicasting
Message brokers

- Diverse applications
  - hence, diverse message formats
  - conversion required
    - perhaps taking application knowledge into account
    - rules for conversion needed

- Supporting a Publish & Subscribe implementation
  - rules for filtering and forwarding needed
Agenda

- Layered protocols and middleware
- (Remote) Procedure calls
- (Remote) Method invocation
- Message Oriented Communication
- Streaming
Stream-oriented communication

- **Media streaming**
  - temporal relationships between data items play a crucial role
    - e.g. audio (mono, stereo), video, audio + video;
      - 20 µsec difference in channels distorts stereo;
      - 80msec difference is acceptable for audio – video (asymmetric).
  - Synchronous: bounded end-to-end delay
  - Bounded interpacket delay (= bounded jitter): isochronous

- **Soft (firm) real-time**
  - missing a packet is a pity but no disaster
  - graceful reduction in quality

- **General issue:** *quality of service*
  - no discrete success/failure but range of qualities, decided dynamically
    - trade resources for quality
  - QoS guarantee is end-to-end concern, points along the route must cooperate
    - RSVP: resource reservation protocol