Layered Protocols

- Low-level layers
- Transport layer
- Application layer
- Middleware layer
Basic Networking Model

Drawbacks:

- Focus on message-passing only
- Often unneeded or unwanted functionality
- **Question:** Violates transparency?

02 – 2  Communication/2.1 Layered Protocols
Low-level layers

**Physical layer**: contains the specification and implementation of bits, and their transmission between sender and receiver.

**Data link layer**: prescribes the transmission of a series of bits into a frame to allow for error and flow control.

**Network layer**: describes how packets in a network of computers are to be *routed*.

**Observation**: for many distributed systems, the lowest-level interface is that of the network layer.
Transport Layer

**Important:** The transport layer provides the actual communication facilities for most distributed systems.

**Standard Internet protocols:**

- TCP: connection-oriented, reliable, stream-oriented communication
- UDP: unreliable (best-effort) datagram communication

**Note:** IP multicasting is generally considered a standard available service.
Client–Server TCP

TCP for transactions (T/TCP): A transport protocol aimed to support client–server interaction

(a) (b)
### Application Layer

**Observation:** Many application protocols are directly implemented on top of transport protocols, doing a lot of application-independent work.

<table>
<thead>
<tr>
<th></th>
<th>News</th>
<th>FTP</th>
<th>WWW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transfer</strong></td>
<td>NNTP</td>
<td>FTP</td>
<td>HTTP</td>
</tr>
<tr>
<td><strong>Encoding</strong></td>
<td>7-bit + MIME</td>
<td>7-bit text + 8-bit binary (user has to guess)</td>
<td>8-bit + content type</td>
</tr>
<tr>
<td><strong>Naming</strong></td>
<td>Newsgroup</td>
<td>Host + path</td>
<td>URL</td>
</tr>
<tr>
<td><strong>Distribution</strong></td>
<td>Push</td>
<td>Pull</td>
<td>Pull</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>Flooding</td>
<td>Caching + DNS tricks</td>
<td>Caching + DNS tricks</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>None (PGP)</td>
<td>Username + Password</td>
<td>Username + Password</td>
</tr>
</tbody>
</table>
**Middleware Layer**

**Observation:** Middleware is invented to provide common services and protocols that can be used by many different applications:

- A rich set of communication protocols, but which allow different applications to communicate
- Marshaling and unmarshaling of data, necessary for integrated systems
- Naming protocols, so that different applications can easily share resources
- Security protocols, to allow different applications to communicate in a secure way
- Scaling mechanisms, such as support for replication and caching

**Note:** what remains are truly application-specific protocols

**Question:** Such as...?
Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations
Basic RPC Operation

Observations:

- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

**Conclusion:** communication between caller & callee can be hidden by using procedure-call mechanism.
Local procedure call: \( \text{read}(\text{int} \; \text{fd}, \; \text{char}^{*} \; \text{buf}, \; \text{int} \; \text{nbytes}) \)

1: Push parameter values of the procedure on a stack
2: Call procedure
3: Use stack for local variables
4: Pop results (in parameters)

**Principle:** “communication” with local procedure is handled by copying data to/from the stack (with a few exceptions)
RPC Implementation (2/2)

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"
Parameter marshaling: There’s more than just wrapping parameters into a message:

- Client and server machines may have different data representations (think of byte ordering)
- Wrapping a parameter means transforming a value into a sequence of bytes
- Client and server have to agree on the same encoding:
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, transforming them into machine-dependent representations.
RPC: Parameter Passing (2/2)

RPC parameter passing:

- RPC assumes \textit{copy in/copy out} semantics: while procedure is executed, nothing can be assumed about parameter values (only Ada supports this model).
- RPC assumes \textit{all} data that is to be operated on is passed by parameters. Excludes passing \textit{references} to (global) data.

**Conclusion:** full access transparency cannot be realized.

**Observation:** If we introduce a \textit{remote reference} mechanism, access transparency can be enhanced:

- Remote reference offers unified access to remote data
- Remote references can be passed as parameter in RPCs
Local RPCs: Doors

**Essence:** Try to use the RPC mechanism as the only mechanism for *interprocess communication* (IPC). Doors are RPCs implemented for processes on the same machine.
Asynchronous RPCs

**Essence:** Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.

**Variation:** *deferred synchronous RPC:*

---

*Communication/2.2 Remote Procedure Call*
**RPC in Practice**

**Essence:** Let the developer concentrate on only the client- and server-specific code; let the RPC system (generators and libraries) do the rest.
Client-to-Server Binding (DCE)

Issues: (1) Client must locate server machine, and (2) locate the server.

Example: DCE uses a separate daemon for each server machine.
Remote Object Invocation

- Distributed objects
- Remote method invocation
- Parameter passing
Remote Distributed Objects (1/2)

- Data and operations **encapsulated** in an object
- Operations are implemented as **methods**, and are accessible through **interfaces**
- Object offers only its **interface** to clients
- **Object server** is responsible for a collection of objects
- **Client stub (proxy)** implements interface
- **Server skeleton** handles (un)marshaling and object invocation

![Diagram of Remote Object Invocation]

**Diagram:**
- **Client machine** to **Server machine** connected by **Network**
- **Client** invokes a **method** to **Proxy**
- **Proxy** invokes the same method at **Object**
- **Skeleton** handles marshaling and unmarshaling
- **Marshalled invocation** is passed across network

*02 – 19 Communication/2.3 Remote Object Invocation*
Remote Distributed Objects (2/2)

Compile-time objects: Language-level objects, from which proxy and skeletons are automatically generated.

Runtime objects: Can be implemented in any language, but require use of an object adapter that makes the implementation appear as an object.

Transient objects: live only by virtue of a server: if the server exits, so will the object.

Persistent objects: live independently from a server: if a server exits, the object’s state and code remain (passively) on disk.
Client-to-Object Binding (1/2)

**Object reference:** Having an object reference allows a client to **bind** to an object:

- Reference denotes server, object, and communication protocol
- Client loads associated stub code
- Stub is instantiated and initialized for specific object

**Two ways of binding:**

- **Implicit:** Invoke methods directly on the referenced object
- **Explicit:** Client must first explicitly bind to object before invoking it
Some remarks:

- Reference may contain a URL pointing to an implementation file
- (Server, object) pair is enough to locate target object
- We need only a standard protocol for loading and instantiating code

**Observation:** Remote-object references allow us to pass references as parameters. This was difficult with ordinary RPCs.
Remote Method Invocation

**Basics:** (Assume client stub and server skeleton are in place)

- Client invokes method at stub
- Stub marshals request and sends it to server
- Server ensures referenced object is active:
  - Create separate process to hold object
  - Load the object into server process
  - ...
- Request is unmarshaled by object’s skeleton, and referenced method is invoked
- If request contained an object reference, invocation is applied recursively (i.e., server acts as client)
- Result is marshaled and passed back to client
- Client stub unmarshals reply and passes result to client application
RMI: Parameter Passing (1/2)

Object reference: Much easier than in the case of RPC:

- Server can simply bind to referenced object, and invoke methods
- Unbind when referenced object is no longer needed

Object-by-value: A client may also pass a complete object as parameter value:

- An object has to be marshaled:
  - Marshall its state
  - Marshall its methods, or give a reference to where an implementation can be found
- Server unmarshals object. Note that we have now created a copy of the original object.
- Object-by-value passing tends to introduce nasty problems
Question: What’s an alternative implementation for a remote-object reference?
Message-Oriented Communication

- Synchronous versus asynchronous communications
- Message-Queuing System
- Message Brokers
- Example: IBM MQSeries
Synchronous Communication

Some observations: Client/Server computing is generally based on a model of synchronous communication:

- Client and server have to be active at the time of communication
- Client issues request and blocks until it receives reply
- Server essentially waits only for incoming requests, and subsequently processes them

Drawbacks synchronous communication:

- Client cannot do any other work while waiting for reply
- Failures have to be dealt with immediately (the client is waiting)
- In many cases the model is simply not appropriate (mail, news)
Async onous Comm unication:

Messa ging

Messa g e-oriented mid dle ware: Aims at high-level asynchronous communication:

- Processes send each other messages, which are queued
- Sender need not wait for immediate reply, but can do other things
- Middleware often ensures fault tolerance

![Diagram of messaging interface]
Persistent vs. Transient Communication

**Persistent communication:** A message is stored at a communication server as long as it takes to deliver it at the receiver.

**Transient communication:** A message is discarded by a communication server as soon as it cannot be delivered at the next server, or at the receiver.
Messaging Combinations

A sends message and continues
A stopped running

B is not running
B starts and receives message

(a)

A sends message and waits until accepted
A stopped running

Message is stored at B's location for later delivery

B is not running
B starts and receives message

(b)

A sends message and continues

Message can be sent only if B is running

B receives message

(c)

Send request and wait until received

Request is received

Running, but doing something else
Process request

(d)

Send request and wait until accepted

Request is received

Accepted

Running, but doing something else
Process request

(e)

Send request and wait for reply

Request is received

Accepted

Running, but doing something else
Process request

(f)

02 – 30 Communication/2.4 Message-Oriented Communication
Message-Oriented Middleware

**Essence:** Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

**Canonical example:** IBM MQSeries
IBM MQSeries (1/3)

Basic concepts:

- **Application-specific messages** are put into, and removed from **queues**
- Queues always reside under the regime of a **queue manager**
- Processes can put messages only in local queues, or through an RPC mechanism

Message transfer:

- Messages are transferred between queues
- Message transfer between queues at different processes, requires a **channel**
- At each endpoint of channel is a **message channel agent**
- Message channel agents are responsible for:
  - Setting up channels using lower-level network communication facilities (e.g., TCP/IP)
  - (Un)wrapping messages from/in transport-level packets
  - Sending/receiving packets
- Channels are inherently unidirectional
- MQSeries provides mechanisms to automatically start MCAs when messages arrive, or to have a receiver set up a channel
- Any network of queue managers can be created; routes are set up manually (system administration)
IBM MQSeries (3/3)

Routing: By using logical names, in combination with name resolution to local queues, it is possible to put a message in a remote queue

Question: What’s a major problem here?
Message Broker

Observation: Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)

Message broker: Centralized component that takes care of application heterogeneity in a message-queuing system:

- Transforms incoming messages to target format, possibly using intermediate representation
- May provide subject-based routing capabilities
- Acts very much like an application gateway
Stream-Oriented Communication

- Support for continuous media
- Streams in distributed systems
- Stream management
Continuous Media

**Observation:** All communication facilities discussed so far are essentially based on a *discrete*, that is *time-independent* exchange of information.

**Continuous media:** Characterized by the fact that values are time dependent:

- Audio
- Video
- Animations
- Sensor data (temperature, pressure, etc.)

**Transmission modes:** Different timing guarantees with respect to data transfer:

- **Asynchronous:** no restrictions with respect to *when* data is to be delivered
- **Synchronous:** define a maximum end-to-end delay for individual data packets
- **Isochronous:** define a maximum and minimum end-to-end delay (*jitter* is bounded)
Stream (1/2)

Definition: A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission.

Some common stream characteristics:

- Streams are unidirectional
- There is generally a single source, and one or more sinks
- Often, either the sink and/or source is a wrapper around hardware (e.g., camera, CD device, TV monitor, dedicated storage)

Stream types:

- Simple: consists of a single flow of data, e.g., audio or video
- Complex: multiple data flows, e.g., stereo audio or combination audio/video

02 – 38 Communication/2.5 Stream-Oriented Communication
**Stream (2/2)**

**Issue:** Streams can be set up between two processes at different machines, or directly between two different devices. Combinations are possible as well.

![Diagram](attachment:image.png)

- **(a)**: Sending process to receiving process via network.
- **(b)**: Camera sending stream to display via network.

*Communication/2.5 Stream-Oriented Communication*
Streams and QoS

**Essence:** Streams are all about timely delivery of data. How do you specify this **Quality of Service (QoS)**? Make distinction between **specification** and **implementation** of QoS.

**Flow specification:** Use a token-bucket model and express QoS in that model

<table>
<thead>
<tr>
<th>Input characteristics</th>
<th>Required Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum data unit size (bytes)</td>
<td>Loss sensitivity (bytes)</td>
</tr>
<tr>
<td>Token bucket rate (bytes/sec)</td>
<td>Loss interval (µsec)</td>
</tr>
<tr>
<td>Token bucket size (bytes)</td>
<td>Burst loss sensitivity (data units)</td>
</tr>
<tr>
<td>Max. transmission rate (bytes/sec)</td>
<td>Min. delay noticed (µsec)</td>
</tr>
<tr>
<td></td>
<td>Max. delay variation (µsec)</td>
</tr>
<tr>
<td></td>
<td>Quality of guarantee</td>
</tr>
</tbody>
</table>

02 – 40  Communication/2.5 Stream-Oriented Communication
Implementing QoS

Problem: QoS specifications translate to resource reservations in underlying communication system. There is no standard way of (1) QoS specs, (2) describing resources, (3) mapping specs to reservations.

Approach: Use Resource reSerVation Protocol (RSVP) as first attempt. RSVP is a transport-level protocol.
Stream Synchronization

**Problem:** Given a complex stream, how do you keep the different substreams in synch?

**Example:** Think of playing out two channels, that together form stereo sound. Difference should be less than 20–30 µsec!

**Alternative:** multiplex all substreams into a single stream, and demultiplex at the receiver. Synchronization is handled at multiplexing/demultiplexing point (MPEG).