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?

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.

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
- (Un)marshaling 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...?
Types of Communication (1/3)

Distinguish:

- **Transient** versus **persistent** communication
- **Asynchronous** versus **synchronous** communication

04 – 6  Communication/4.1 Layered Protocols

Types of Communication (2/3)

**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.

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

04 – 7  Communication/4.1 Layered Protocols
Types of Communication (3/3)

Places for synchronization:

- At request submission
- At request delivery
- After request processing

Client/Server

Some observations: Client/Server computing is generally based on a model of transient 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)
Messaging

**Message-oriented middleware:** Aims at high-level persistent 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

Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations
Basic RPC Operation (1/2)

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.

Basic RPC Operation (2/2)

1. Client procedure calls client stub as usual.
2. Client stub builds message and 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 and calls server.
6. Server does work and returns result to the stub.
7. Server stub packs it in message and calls OS.
8. Server's OS sends message to client's OS.
9. Client's OS gives message to client stub.
10. Client stub unpacks result and returns to the client.
RPC: Parameter Passing (1/2)

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

Conclusion: full access transparency cannot be realized.

Observation: If we introduce a 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
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:

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.

---

**Message-Oriented Communication**

- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere
Transient Messaging: Sockets

**Example:** Consider the Berkeley socket interface, which has been adopted by all UNIX systems, as well as Windows 95/NT/2000/XP/Vista:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication endpoint</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 N conn-</td>
</tr>
<tr>
<td></td>
<td>nections</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block until someone remote wants to</td>
</tr>
<tr>
<td></td>
<td>establish a connection</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send data over a connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Receive data over a connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

Server

Client

Synchronization point

Message-Oriented Middleware

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</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 non-</td>
</tr>
<tr>
<td></td>
<td>empty, and remove the first message</td>
</tr>
<tr>
<td>POLL</td>
<td>Check a specified queue for messages,</td>
</tr>
<tr>
<td></td>
<td>and remove the first. Never block</td>
</tr>
<tr>
<td>NOTIFY</td>
<td>Install a handler to be called when a</td>
</tr>
<tr>
<td></td>
<td>message is put into the specified que-</td>
</tr>
<tr>
<td></td>
<td>ue</td>
</tr>
</tbody>
</table>
**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 an MQ system:

- Transforms incoming messages to target format
- Very often acts as an **application gateway**
- May provide **subject-based** routing capabilities

⇒ **Enterprise Application Integration**

---

**IBM’s WebSphere MQ (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
IBM’s WebSphere MQ (2/3)

- Channels are inherently unidirectional
- MQ 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’s WebSphere MQ (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?
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

**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

Streams and QoS

**Essence:** Streams are all about timely delivery of data. How do you specify this **Quality of Service (QoS)**? Basics:

- The required **bit rate** at which data should be transported.
- The **maximum delay** until a session has been set up (i.e., when an application can start sending data).
- The **maximum end-to-end delay** (i.e., how long it will take until a data unit makes it to a recipient).
- The maximum delay variance, or **jitter**.
- The **maximum round-trip delay**.
Enforcing QoS (1/2)

**Observation:** There are various network-level tools, such as **differentiated services** by which certain packets can be prioritized.

**Also:** Use **buffers** to reduce jitter:

![Buffer Diagram]

Enforcing QoS (2/2)

**Problem:** How to reduce the effects of packet loss (when multiple samples are in a single packet)?

**Solution:** Simply spread the samples:

![Sample Spread Diagram]
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).

Multicast Communication

- Application-level multicasting
- Gossip-based data dissemination
**Application-Level Multicasting**

**Essence:** Organize nodes of a distributed system into an **overlay network** and use that network to disseminate data. **Example:** Consider a Chord-based peer-to-peer system:

1. Initiator generates a **multicast identifier** mid.
2. Lookup `succ(mid)`, the node responsible for mid.
3. Request is routed to `succ(mid)`, which will become the **root**.
4. If P wants to join, it sends a join request to the root.
5. When request arrives at Q:
   - Q has not seen a join request before $\Rightarrow$ it becomes forwarder; P becomes child of Q. Join request continues to be forwarded.
   - Q knows about tree $\Rightarrow$ P becomes child of Q. No need to forward join request anymore.

**ALM: Some costs**

- **Link stress:** How often does an ALM message cross the same physical link? **Example:** message from A to D needs to cross $(Ra, Rb)$ twice.
- **Stretch:** Ratio in delay between ALM-level path and network-level path. **Example:** messages B to C follow path of length 71 at ALM, but 47 at network level $\Rightarrow$ stretch = 71/47.
Epidemic Algorithms

- General background
- Update models
- Removing objects

Principles

**Basic idea:** Assume there are no write–write conflicts:

- Update operations are initially performed at one or only a few replicas
- A replica passes its updated state to a limited number of neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

**Anti-entropy:** Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards

**Gossiping:** A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).
Anti-Entropy

- A node $P$ selects another node $Q$ from the system at random.
- **Push**: $P$ only sends its updates to $Q$
- **Pull**: $P$ only retrieves updates from $Q$
- **Push-Pull**: $P$ and $Q$ exchange mutual updates (after which they hold the same information).

**Observation**: for push-pull it takes $O(\log(N))$ rounds to disseminate updates to all $N$ nodes (**round** = when every node as taken the initiative to start an exchange).

Gossiping

**Basic model**: A server $S$ having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, $S$ stops contacting other servers with probability $1/k$.

If $s$ is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers

$$s = e^{-(k+1)(1-s)}$$

**Observation**: If we really have to ensure that all servers are eventually updated, gossiping alone is not enough.
Deleting Values

**Fundamental problem:** We cannot remove an old value from a server and expect the removal to propagate. Instead, mere removal will be undone in due time using epidemic algorithms.

**Solution:** Removal has to be registered as a special update by inserting a *death certificate*.

**Next problem:** When to remove a death certificate (it is not allowed to stay for ever):
- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection).
- Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers).

**Note:** it is necessary that a removal actually reaches all servers.

**Question:** What’s the scalability problem here?

---

Example Applications

**Data dissemination:** Perhaps the most important one. Note that there are many variants of dissemination.

**Aggregation:** Let every node $i$ maintain a variable $x_i$. When two nodes gossip, they each reset their variable to

$$x_i, x_j \leftarrow (x_i + x_j) / 2$$

Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i / N$.

**Question:** What happens if initially $x_i = 1$ and $x_j = 0, j \neq i$?