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
- Violates access transparency
Low-level layers

Recap

- **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 often considered a standard available service (which may be dangerous to assume).
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**
- *(Un)marshaling* of data, necessary for integrated systems
- **Naming protocols**, to allow easy sharing of resources
- **Security protocols** for secure communication
- **Scaling mechanisms**, such as for replication and caching

**Note**

What remains are truly **application-specific** protocols... such as?
Types of communication

Distinguish

- Transient versus persistent communication
- Asynchronous versus synchronous communication
Types of communication

- **Transient communication**: Comm. server discards message when 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.
Types of communication

- **Client**
  - Synchronize after processing by server

- **Server**
  - Synchronize at request delivery
  - Synchronize at request submission

**Places for synchronization**

- At request submission
- At request delivery
- After request processing
Some observations

Client/Server computing is generally based on a model of *transient synchronous communication*:

- Client and server have to be active at 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 handled immediately: the client is waiting.
- The model may simply not be appropriate (mail, news).
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 time of commun.
- 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 handled immediately: the client is waiting
- The model may simply not be 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
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

**Client machine**

- **Client OS**
- **Client process**
  - Procedure call: `k = add(i,j)`
  - Parameters:
    - `proc: "add"`
    - `int: val(i)`
    - `int: val(j)`

**Server machine**

- **Server OS**
- **Server process**
  - Implementation of `add`
  - Procedure call: `k = add(i,j)`
  - Parameters:
    - `proc: "add"`
    - `int: val(i)`
    - `int: val(j)`

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"**
7. **Server makes local call and returns result to stub.**
8. **Stub builds message; calls OS.**
9. **OS sends message to client’s OS.**
10. **Client’s OS gives message to stub.**
    - **Client stub unpacks result and returns to the client.**
RPC: Parameter passing

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

**RPC parameter passing: some assumptions**

- **Copy in/copy out** semantics: while procedure is executed, nothing can be assumed about parameter values.
- **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**

A remote reference mechanism enhances access transparency:

- Remote reference offers unified access to remote data
- Remote references can be passed as parameter in RPCs
RPC: Parameter passing

**RPC parameter passing: some assumptions**

- Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
- **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**

A remote reference mechanism enhances access transparency:

- Remote reference offers **unified access** to remote data
- Remote references can be **passed as parameter** in RPCs
RPC: Parameter passing

**RPC parameter passing: some assumptions**

- **Copy in/copy out** semantics: while procedure is executed, nothing can be assumed about parameter values.
- **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**

A **remote reference** mechanism enhances access transparency:

- 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.
Deferred synchronous RPCs

Variation

Client can also do a (non)blocking poll at the server to see whether results are available.
RPC in practice

- Uuidgen
- Interface definition file
- IDL compiler

Client code
- Client stub
- Header
- Server stub
- Server code

- C compiler
- C compiler
- C compiler
- C compiler

- Client object file
- Client stub object file
- Server stub object file
- Server object file

- Include
- Include

- Linker
- Runtime library
- Runtime library
- Linker

- Client binary
- Server binary
Client-to-server binding (DCE)

**Issues**

1. Client must locate server machine, and 2. locate the server.
Message-Oriented Communication

- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere
## Transient messaging: sockets

### Berkeley socket interface

<table>
<thead>
<tr>
<th>Function</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 $N$ connections</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block until request to 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>
Transient messaging: sockets

Server

- socket
- bind
- listen
- accept
- read
- write
- close

Client

- socket
- connect
- write
- read
- close

Synchronization point

Communication
Sockets: Python code

**Server**

```python
import socket
HOST = '
PORT = SERVERPORT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(N) # listen to max N queued connection
(conn, addr) = s.accept() # returns new socket + addr client
while 1: # forever
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

**Client**

```python
import socket
HOST = 'distsys.cs.vu.nl'
PORT = SERVERPORT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
```
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 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>
**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
Message broker

Source client

Message broker

Broker program

Repository with conversion rules and programs

Queuing layer

Destination client

OS

Network
IBM’s WebSphere MQ

**Basic concepts**

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

**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
Automatically start MCAs when messages arrive
Any network of queue managers can be created
Routes are set up manually (system administration)
IBM’s WebSphere MQ

**Routing**

By using **logical names**, in combination with name resolution to local queues, it is possible to put a message in a **remote queue**.
Stream-oriented communication

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

**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)
- **Simple stream**: a single flow of data, e.g., audio or video
- **Complex stream**: 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

Observation

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

Also

Use buffers to reduce jitter:

- Packet departs source: 1 2 3 4 5 6 7 8
- Packet arrives at buffer: 1 2 3 4 5 6 7 8
- Packet removed from buffer: 1 2 3 4 5 6 7 8

Time in buffer: 1 2 3 4 5 6 7 8

Gap in playback:

Time (sec)
Enforcing QoS

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

(a) Lost packet
(b) Lost frames
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).
Stream synchronization

**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.

**Chord-based tree building**
1. Initiator generates a multicast identifier \( \text{mid} \).
2. Lookup \( \text{succ}(\text{mid}) \), the node responsible for \( \text{mid} \).
3. Request is routed to \( \text{succ}(\text{mid}) \), 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 \( \langle Ra, Rb \rangle \) 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 performed at a single server
- A replica passes updated state to only a few neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Two forms of epidemics

- **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

Principle operations
- 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 $\mathcal{O}(\log(N))$ rounds to disseminate updates to all $N$ nodes (round = when every node as taken the initiative to start an exchange).
Consider a single source, propagating its update. Let $p_i$ be the probability that a node has not received the update after the $i$-th cycle.

**Basics**

**Analysis: staying ignorant**

- With **pull**, $p_{i+1} = (p_i)^2$: the node was not updated during the $i$-th cycle and should contact another ignorant node during the next cycle.
- With **push**, $p_{i+1} = p_i \left(1 - \frac{1}{N}\right)^N(1-p_i) \approx p_i e^{-1}$ (for small $p_i$ and large $N$): the node was ignorant during the $i$-th cycle and no updated node chooses to contact it during the next cycle.
- With **push-pull**: $(p_i)^2 \cdot (p_i e^{-1})$
Anti-entropy performance
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 $1/k$.

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

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

Consider 10,000 nodes

<table>
<thead>
<tr>
<th>k</th>
<th>s</th>
<th>Ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.203188</td>
<td>2032</td>
</tr>
<tr>
<td>2</td>
<td>0.059520</td>
<td>595</td>
</tr>
<tr>
<td>3</td>
<td>0.019827</td>
<td>198</td>
</tr>
<tr>
<td>4</td>
<td>0.006977</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>0.002516</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>0.000918</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>0.000336</td>
<td>3</td>
</tr>
</tbody>
</table>

Note

If we really have to ensure that all servers are eventually updated, gossiping alone is not enough.
Gossiping

Consider 10,000 nodes

<table>
<thead>
<tr>
<th>k</th>
<th>s</th>
<th>N_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.203188</td>
<td>2032</td>
</tr>
<tr>
<td>2</td>
<td>0.059520</td>
<td>595</td>
</tr>
<tr>
<td>3</td>
<td>0.019827</td>
<td>198</td>
</tr>
<tr>
<td>4</td>
<td>0.006977</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>0.002516</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>0.000918</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>0.000336</td>
<td>3</td>
</tr>
</tbody>
</table>

Note

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.
Deleting values

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

Typical apps

- **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

$$\bar{x} = \sum_i x_i / N.$$
Example application: aggregation

Aggregation (continued)

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

\[ \bar{x} = \frac{\sum_i x_i}{N}. \]

Question
What happens if initially \( x_i = 1 \) and \( x_j = 0, j \neq i \)?

Question
How can we start this computation without pre-assigning a node \( i \) to start as only one with \( x_i \leftarrow 1 \)?
## Example application: aggregation

### Aggregation (continued)

When two nodes gossip, they each reset their variable to

\[ x_i, x_j \leftarrow \frac{x_i + x_j}{2} \]

Result: in the end each node will have computed the average

\[ \bar{x} = \frac{\sum_i x_i}{N}. \]

### Question

What happens if initially \( x_i = 1 \) and \( x_j = 0, j \neq i \)?

### Question

How can we start this computation without pre-assigning a node \( i \) to start as only one with \( x_i \leftarrow 1 \)?
Example application: aggregation

Aggregation (continued)

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 \( x_j = 0, j \neq i \)?

Question

How can we start this computation without pre-assigning a node \( i \) to start as only one with \( x_i \leftarrow 1 \)?