Basic networking model

The Open Systems Interconnection (OSI) 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
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
An adapted layering scheme
Types of communication

Distinguish...

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

**Transient versus persistent**

- **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

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 time of communication
- Client issues request and blocks until it receives reply
- Server essentially waits only for incoming requests, and subsequently processes them
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 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)
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
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.
Basic RPC operation

1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters; calls server.
6. Server does local call; returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client’s OS.
9. Client’s OS gives message to stub.
10. Client stub unpacks result; returns to client.
RPC: Parameter passing

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)

Conclusion

Client and server need to properly interpret messages, transforming them into machine-dependent representations.
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.
Sending out multiple RPCs

**Essence**

Sending an RPC request to a group of servers.

---

**Be careful...**

Order of multiple RPCs (are they embarrassingly parallel?)
**Transient messaging: sockets**

### Berkeley socket interface

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</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>Tell operating system what the maximum number of pending connection requests should be</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>

---

**Server**

- `socket` → `bind` → `listen` → `accept` → `receive` → `send` → `close`  
- Synchronization point: `accept`  
- Communication: `receive` → `send` → `receive`

**Client**

- `socket` → `connect` → `send` → `receive` → `close`
Sockets: Python code

Server

```python
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.bind((HOST, PORT))
4 s.listen(1)
5 (conn, addr) = s.accept()  # returns new socket and addr. client
6 while True:  # forever
7    data = conn.recv(1024)  # receive data from client
8    if not data: break  # stop if client stopped
9    conn.send(str(data)+"*")  # return sent data plus an "*"
10   conn.close()  # close the connection
```

Client

```python
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT))  # connect to server (block until accepted)
4 s.send('Hello, world')  # send same data
5 data = s.recv(1024)  # receive the response
6 print data  # print the result
7 s.close()  # close the connection
```
Making sockets easier to work with

Observation

Sockets are rather low level and programming mistakes are easily made. However, the way that they are used is often the same (such as in a client-server setting).

Alternative: ZeroMQ (more read: Kafka, RabbitMQ)

Provides a higher level of expression by pairing sockets: one for sending messages at process $P$ and a corresponding one at process $Q$ for receiving messages. All communication is asynchronous.

Three patterns

- Request-reply
- Publish-subscribe
- Pipeline
MPI: When lots of flexibility is needed. The de facto Protocol in High-Performance Computing (HPC)

### Representative operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_bsend</td>
<td>Append outgoing message to a local send buffer</td>
</tr>
<tr>
<td>MPI_send</td>
<td>Send a message and wait until copied to local or remote buffer</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send a message and wait until transmission starts</td>
</tr>
<tr>
<td>MPI_sendrecv</td>
<td>Send a message and wait for reply</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and continue</td>
</tr>
<tr>
<td>MPI_issend</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Receive a message; block if there is none</td>
</tr>
<tr>
<td>MPI_irecv</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>
# Message-oriented middleware

## Essence

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

## Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>put</strong></td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td><strong>get</strong></td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td><strong>poll</strong></td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td><strong>notify</strong></td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>
General model

Queue managers

Queues are managed by **queue managers**. An application can put messages only into a **local** queue. Getting a message is possible by extracting it from a **local** queue only $\implies$ queue managers need to **route** messages.

Routing

![General architecture of a message-queuing system](image-url)
Application-level multicasting (ALM)

Essence
Organize nodes of a distributed system into an overlay network and use that network to disseminate data:

- Oftentimes a tree, leading to unique paths
- Alternatively, also mesh networks, requiring a form of routing
ALM: Some costs

Different metrics

- **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 73 at ALM, but 47 at network level \( \Rightarrow \) stretch = 73/47.
**Flooding**

**Essence**

P simply sends a message m to each of its neighbors. Each neighbor will forward that message, except to P, and only if it had not seen m before.

**Performance**

The more edges, the more expensive!

The size of a random overlay as function of the number of nodes

![Graph showing the size of a random overlay as function of the number of nodes with different probabilities for edge forwarding.](image)

Key:
- $p_{edge} = 0.6$
- $p_{edge} = 0.4$
- $p_{edge} = 0.2$
**Flooding**

**Essence**

*P* simply sends a message *m* to each of its neighbors. Each neighbor will forward that message, except to *P*, and only if it had not seen *m* before.

**Performance**

The more edges, the more expensive!

**Variation**

Let *Q* forward a message with a certain probability $p_{\text{flood}}$, possibly even dependent on its own number of neighbors (i.e., node degree) or the degree of its neighbors.
Epidemic protocols

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
- **Rumor spreading**: 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.
- **Pull**: $P$ only pulls in new updates from $Q$
- **Push**: $P$ only pushes its own updates to $Q$
- **Push-pull**: $P$ and $Q$ send updates to each other

Observation
For push-pull it takes $\mathcal{O}(\log(N))$ rounds to disseminate updates to all $N$ nodes
(round = when every node has taken the initiative to start an exchange).
Anti-entropy performance

Information dissemination models

$N = 10,000$

Probability not yet updated

Round
Rumor spreading

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 $p_{stop}$. 
Rumor spreading

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 $p_{stop}$.

Note

If we really have to ensure that all servers are eventually updated, rumor spreading alone is not enough.
Deleting values

Fundamental problem
We cannot remove an old value from a server and expect the removal to propagate.

Solution
Removal has to be registered as a special update by inserting a death certificate.
Deleting values

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.