Chapter 04: Communication

Version: March 28, 2019
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
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.
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**.
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.
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.

A remote reference mechanism enhances access transparency

- Remote reference offers **unified access** to remote data
- Remote references can be **passed as parameter** in RPCs
- Note: stubs can sometimes be used as such references
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.

[Diagram showing the essence of sending multiple RPCs]
RPC in practice

Writing a Client and a Server
Client-to-server binding (DCE)

Issues

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

![Diagram of communication process](attachment:image.png)
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
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

## Representative operations

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

The Message-Passing Interface (MPI)
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>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 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 ⇒ queue managers need to **route** messages.

**Routing**

General architecture of a message-queuing system
General model

Routing

Source queue manager

Local OS

Look up contact address of destination queue manager

Logical queue-level address (name)

Address lookup database

Destination queue manager

Local OS

Network

Contact address

Issues: #1

Queues address lookup: it is preferable that queues have logical, location-independent names.
General model

Routing

Source queue manager

Look up contact address of destination queue manager

Logical queue-level address (name)

Address lookup database

Destination queue manager

Local OS

Network

Contact address

Issues: #2

Name-to-address mapping needs to be available to each queue manager. Common approach: lookup table ⇒ maintenance problem.
General model

Routing

Source queue manager

Look up contact address of destination queue manager

Logical queue-level address (name)

Address lookup database

Local OS

Destination queue manager

Local OS

Network

Contact address

Issues: #3

Scalability - dealing with very large message-queueing systems (lookup tables).
Commom approach: special queue managers operate as routers.
Message broker

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

Broker handles 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 (i.e., publish-subscribe capabilities)
Message broker: general architecture

Diagram showing the general architecture of a message broker. The diagram includes:
- Source: Application connected to Interface, Local OS
- Message broker: Broker plugins, Rules, Queuing layer, Local OS
- Destination: Application connected to Interface, Local OS
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

Message transfer
- Messages are transferred between queues
- Message transfer between queues at different processes, requires a **channel**
- At each end point 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

Schematic overview

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)
## Message channel agents

### Some attributes associated with message channel agents

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport type</td>
<td>Determines the transport protocol to be used</td>
</tr>
<tr>
<td>FIFO delivery</td>
<td>Indicates that messages are to be delivered in the order they are sent</td>
</tr>
<tr>
<td>Message length</td>
<td>Maximum length of a single message</td>
</tr>
<tr>
<td>Setup retry count</td>
<td>Specifies maximum number of retries to start up the remote MCA</td>
</tr>
<tr>
<td>Delivery retries</td>
<td>Maximum times MCA will try to put received message into queue</td>
</tr>
</tbody>
</table>
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.
Application-level multicasting

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 \( \Rightarrow \text{stretch} = 73/47 \).
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.
**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.

**Variation**

Let *Q* forward a message with a certain probability $p_{flood}$, 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).