Chapter 4: Interprocess Communication

Middleware layers

Applications, services

RMI and RPC

request-reply protocol
marshalling and external data representation

UDP and TCP

This chapter

Middleware layers
API for Internet Protocols (1): IPC characteristics

- synchronous and asynchronous communication
  - blocking send: waits until the corresponding receive is issued
  - non-blocking send: sends and moves on
  - blocking receive: waits until the msg is received
  - non-blocking receive: if the msg is not here, moves on
- blocking send and receive
- non-blocking send and blocking or non-blocking receive

Message Destination
- IP address + port: one receiver, many senders
- Location transparency
  - name server or binder: translate service to location
  - OS (e.g. Mach): provides location-independent identifier mapping to lower-lever addresses
- send directly to processes (e.g. V System)
- multicast to a group of processes (e.g. Chorous)

Reliability
Ordering

API for the Internet Protocols (2): Sockets and ports

- programming abstraction for UDP/TCP
- originated from BSD UNIX

![Diagram](attachment://socket_diagram.png)
API for Internet Protocols (3): UDP Datagram

- **message size**: up to $2^{16}$, usually restrict to 8K
- **blocking**: non-blocking send, blocking receive
- **timeouts**: timeout on blocking receive
- **receive from any**: doesn't specify sender origin (possible to specify a particular host for send and receive)
- **failure model**:
  - omission failures: can be dropped
  - ordering: can be out of order
- **use of UDP**
  - **DNS**
  - **less overhead**: no state information, extra messages, latency due to start up

API for Internet Protocols (4): C and UDP datagrams

**Sending a message**

\[
s = \text{socket(AF_INET, SOCK_DGRAM, 0)}
\]

\[
\text{bind}(s, \text{ClientAddress})
\]

\[
\text{sendto}(s, \text{"message"}, \text{ServerAddress})
\]

**Receiving a message**

\[
s = \text{socket(AF_INET, SOCK_DGRAM, 0)}
\]

\[
\text{bind}(s, \text{ServerAddress})
\]

\[
\text{amount} = \text{recvfrom}(s, \text{buffer, from})
\]

**ServerAddress and ClientAddress** are socket addresses
API for Internet Protocols (5): Java and UDP

Client

```java
Socket = new DatagramSocket();
...
InetAddress aHost = InetAddress.getByName(...);
...
DatagramPacket request = new DatagramPacket(msg, length, aHost, serverPort);
...
Socket.send(request);
...
DatagramPacket reply = new DatagramPacket(buffer, length);
...
Socket.receive(reply);

Server

API for Internet Protocols (6): TCP stream

- message size: unlimited
- lost messages: sequence #, ack, retransmit after timeout of no ack
- flow control: sender can be slowed down or blocked by the receiver
- message duplication and ordering: sequence #
- message destination: establish a connection, one sender-one receiver, high overhead for short communication
- matching of data items: two processes need to agree on format and order (protocol)
- blocking: non-blocking send, blocking receive (send might be blocked due to flow control)
- concurrency: one receiver, multiple senders, one thread for each connection
- failure model
  - checksum to detect and reject corrupt packets
  - sequence # to deal with lost and out-of-order packets
  - connection broken if ack not received when timeout
  - could be traffic, could be lost ack, could be failed process...
  - can't tell if previous messages were received
- use of TCP: http, ftp, telnet, smtp
### API for Internet Protocols (7): C and TCP streams

**Requesting a connection**

- \( s = \text{socket}(AF\_INET, SOCK\_STREAM,0) \)
- connect\((s, ServerAddress)\)
- write\((s, \text{"message"}, \text{length})\)

**Listening and accepting a connection**

- \( s = \text{socket}(AF\_INET, SOCK\_STREAM,0) \)
- bind\((s, ServerAddress)\)
- listen\((s,5)\)
- sNew = accept\((s, ClientAddress)\)
- \( n = \text{read}(sNew, \text{buffer}, \text{amount}) \)

*ServerAddress* and *ClientAddress* are socket addresses

---

### API for Internet Protocols (8): Java and TCP

**Client**

- `Socket s = new Socket(host, serverPort);`
- `...`
- `DataInputStream in = new DataInputStream(s.getInputStream());`
- `DataOutputStream out = new DataOutputStream(s.getOutputStream());`
- `...`
- `out.write(...);`
- `...`
- `in.read(...);`

**Server**

- `ServerSocket listenSocket = new ServerSocket(serverPort);`
- `...`
- `Socket s = listenSocket.accept();`
- `...`
- `DataInputStream in = new DataInputStream(s.getInputStream());`
- `DataOutputStream out = new DataOutputStream(s.getOutputStream());`
- `...`
- `in.read(...);`
- `out.write(...);`
External Data Representation (1):

- different ways to represent int, float, char... (internally)

- byte ordering for integers
  - big-endian: most significant byte first
  - small-endian: least significant byte first

- standard external data representation
  - marshal before sending, unmarshal before receiving
  - send in sender’s format and indicates what format, receivers translate if necessary

- External data representation
  - SUN’s External data representation (XDR)
  - CORBA’s Common Data Representation (CDR)
  - Java’s object serialization
  - ASCII (XML, HTTP)

External Data Representation (2): CDR

- Primitive types (15): short, long ...
  - support both big-endian and little-endian
  - transmitted in sender’s ordering and the ordering is specified
  - receiver translates if needed

- Constructed types

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>
External Data Representation (3):

- CORBA IDL compiler generates marshalling and unmarshalling routines
- Struct with string, string, unsigned long

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>4 bytes</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smith&quot;</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h &quot;</td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;London&quot;</td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on &quot;</td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
</tr>
</tbody>
</table>

The flattened form represents a `Person` struct with value: `{Smith, London, 1934}`

External Data Representation (4): Java serialization

- Serialization and deserialization are automatic in arguments and return values of Remote Method Interface (RMI)
- Flattened to be transmitted or stored on the disk
  - Write class information, types and names of instance variables
  - New classes, recursively write class information, types, names...
  - Each class has a handle, for subsequent references
  - Values are in Universal Transfer Format (UTF)
External Data Representation (5): Java serialization

```java
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;

    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
}
```

### Serialized values

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>h0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String name: java.lang.String place:</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith 6 London</td>
<td>h1</td>
</tr>
</tbody>
</table>

The true serialized form contains additional type markers; h0 and h1 are handles to other objects.

Explanation

- class name, version number
- number, type and name of instance variables
- values of instance variables

---

External Data Representation (6): XML

- Extensible markup language (XML)
  - User-defined tags (vs. HTML has a fixed set of tags)
  - different applications agree on a different set of tags
  - E.g. SOAP for web services, tags are published
  - Tags are in plain text (not binary format)—not space efficient
External Data Representation (7)

Person struct in XML
- Tag names: person, name, place, year
- Element: `<name>Smith</name>`
- Attribute: `id="123456789"` of person
- Binary data need to be converted to characters (base64)

```xml
<person id="123456789">
  <name>Smith</name>
  <place>London</place>
  <year>1934</year>
  <!-- a comment -->
</person>
```

External Data Representation (8): XML schema

- Defines elements and attributes
- Similar to type definition
- `xsd:` namespace for xml schema definition

```xml
<xsd:schema xmlns:xsd = "URL of XML schema definitions">
  <xsd:element name="person" type="personType"/>
  <xsd:complexType name="personType">
    <xsd:sequence>
      <xsd:element name="name" type="xs:string"/>
      <xsd:element name="place" type="xs:string"/>
      <xsd:element name="year" type="xs:positiveInteger"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xs:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
```
External Data Representation (9): Remote object reference

**call methods on a remote object**

- unique reference in the distributed system
- Reference = IP address + port + process creation time + local object # in a process + interface (other ways?)
- Port + process creation time -> unique process
- Address can be derived from the reference
- Objects usually don't move; is there a problem if the remote object moves?

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>interface of remote object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
<td></td>
</tr>
</tbody>
</table>

Client-server communication (1)

- Synchronous: client waits for a reply
- Asynchronous: client doesn’t wait for a reply
### Client-server communication (2): Request-reply message structure

| messageType | int (0=Request, 1=Reply) |
| requestID | int |
| objectReference | RemoteObjectRef |
| methodId | int or Method |
| arguments | array of bytes |

**Why requestID?**

### Client-server communication (3)

- **Failure model**
  - UDP: could be out of order, lost...
  - process can fail...
- not getting a reply
  - timeout and retry
- duplicate request messages on the server
  - How does the server find out?
- **idempotent** operation: can be performed repeatedly with the same effect as performing once.
  - idempotent examples?
  - non-idempotent examples?
- **history of replies**
  - retransmission without re-execution
  - how far back if we assume the client only makes one request at a time?
Client-server communication (4): RPC exchange protocols

<table>
<thead>
<tr>
<th>Name</th>
<th>Messages sent by</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request Reply</td>
</tr>
<tr>
<td>RRA</td>
<td>Request Reply Acknowledge reply</td>
</tr>
</tbody>
</table>

Client-server communication (5)

- using TCP increase reliability and also cost
- HTTP uses TCP
  - one connection per request-reply
  - HTTP 1.1 uses "persistent connection"
    - multiple request-reply
    - closed by the server or client at any time
    - closed by the server after timeout on idle time
  - Marshal messages into ASCII text strings
  - resources are tagged with MIME (Multipurpose Internet Mail Extensions) types: test/plain, image/gif...
  - content-encoding specifies compression alg
Client-server communication (6): HTTP methods

- **GET**: return the file, results of a cgi program, ...
- **HEAD**: same as GET, but no data returned
- **POST**: transmit data from client to the program at url
- **PUT**: store data at url
- **DELETE**: delete resource at url
- **OPTIONS**: server provides a list of valid methods
- **TRACE**: server sends back the request

Client-server communication (6): HTTP request/reply format

<table>
<thead>
<tr>
<th>method</th>
<th>URL or pathname</th>
<th>HTTP version</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>//www.dcs.qmw.ac.uk/index.html</td>
<td>HTTP/1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Headers**: latest modification time, acceptable content type, authorization credentials

<table>
<thead>
<tr>
<th>HTTP version</th>
<th>status code</th>
<th>reason</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
<td>OK</td>
<td></td>
<td>resource data</td>
</tr>
</tbody>
</table>

- **Headers**: authentication challenge for the client
Group communication (1)

- **multicast**
- **useful for:**
  - fault tolerance based on replicated services
    - requests multicast to servers, some may fail, the client will be served
  - discovering services
    - multicast to find out who has the services
  - better performance through replicated data
    - multicast updates
  - event notification
    - new items arrived, advertising services

Group communication (2): IP multicast

- class D addresses, first four bits are 1110 in IPv4
- UDP
- Join a group via socket binding to the multicast address
- messages arriving on a host deliver them to all local sockets in the group
- multicast routers: route messages to out-going links that have members
- multicast address allocation
  - permanent
  - temporary:
    - no central registry, use (time to live) TTL to limit the # of hops, hence distance
    - tools like sd (session directory) can help manage multicast addresses and find new ones
Group communication (3): Reliability and ordering

- UDP-level reliability: missing, out-of-order...
- Effects on
  - fault tolerance based on replicated services
    - ordering of the requests might be important, servers can be inconsistent with one another
  - discovering services
    - not too problematic
  - better performance through replicated data
    - loss and out-of-order updates could yield inconsistent data, sometimes this may be tolerable
  - event notification
    - not too problematic