Interprocess Communication (IPC)

- Middleware
  - low layer: supports basic IPC
  - next layer: high level communication paradigm RMI, RPC

Overview

Java APIs for Internet Protocols

- UDP
  - message passing abstraction
  - processes transmit a single datagram to a receiving process
  - best effort
  - no guarantees
- TCP
  - abstraction of 2-way stream
  - streams have no message boundaries
  - basis of producer/consumer communication
  - transparent recovery
  - higher overhead than UDP
  - reliable
  - if connection fails, exception is produced

Data Representation

- how objects/data are translated into suitable form for sending as messages over network
  - receiver needs to be able to decode what it receives

Higher level Protocols

- request-reply protocols: client-server
- group multicast protocol: group communication

API for IP

- processes use two message communication functions: `send`, `receive`
- queue associated with each message destination
  - receive side: OS is producer, process is consumer
- **synchronous communication:** both `send` and `receive` are **blocking**
  - when a `send` is issued, sending process is blocked until `receive` is issued
  - when a `receive` is issued, process blocks until a message arrives
- **asynchronous communication:** `send` is non-blocking
  - sending process returns as soon as the message is copied to a local buffer
  - transmission of the message proceeds in parallel
  - `receive` usually blocking, but can be non-blocking
- non-blocking `receive`: provides buffer to be filled in the background
  - needs an interrupt/polling to be notified when the buffer is filled
  - may be more efficient, requires more complex code to acquire incoming message
- blocking `receive`: when you can have multiple threads in a single process (e.g. in Java), there are no disadvantages, as one thread can issue the blocking call while other threads remain active

Communication	`send`	`receive`
Synchronous	blocking	blocking
Asynchronous	non-blocking	blocking (usually)

- producer/consumer: linked blocking queue
- **producer:**
  - `offer()`: look at queue - if full, returns False and doesn't add to the queue
    - otherwise adds data to the queue
    - non-blocking
  - `put()`: blocks until it can be put in the queue
    - causes context switch
- **consumer:**
  - `take()`: blocks until there's something to take from the queue

        * causes context switch
    - `peek()`: non-blocking look at the first element without removing it
    - `poll()`: returns **`null`** if empty, or 1st item from queue (removing it)
        * non-blocking

- NB: different to synchronous protocol: send a message and don't do anything else until reply received (doesn't mean thread is blocked)
- Node uses non-blocking calls and is single-threaded

Sockets

- **socket:** provides end point for communication between processes
    - to receive messages, its socket must be **bound** to a local port on one of the Internet addresses of the host
    - same socket can be used for both sending/receiving
    - each socket is associated with single protocol: TCP/UDP

Java Internet Address

- `IntAddress`: class encapsulating Internet address
- call `getByName` to get an instance
- throws `UnknownHostException`

```
1 IntAddress aComputer = IntAddress.getByName("registermachine.com")
```

UDP datagram Communication

- server (receiver) binds its socket to a server port (known to the client)
- client (sender) binds socket to any free port
- receive method returns Internet address/port of the sender with the message
    - this allows replies to be sent
- message size
    - receiving process defines array of bytes to receive message
    - if too big message is truncated
    - practical limit 8kB
    - protocol allows packets up to 2^16 bytes

- barebones: low overhead
- e.g. DNS, VoIP

Blocking

- non-blocking `send`s
- blocking `receive`s
- message delivered to message buffer of socket bound to the destination port
- invocations of receive on the socket collect the messages
- messages discarded if no socket bound to the port

Timeouts

- `receive` waits indefinitely until messages received
- can set timeouts on sockets to exit from infinite waits and check condition of sender
- `receive` allows receiving from any port
  - can be restricted to given IP addr/port

Possible failures

- **data corruption:** detected with checksum
- **omission failures:** buffers full, corruption, dropping
- **order:** messages may be delivered out of order

Java API

- `DataGramPacket`
  - 2 constructors for sending or for receiving
  - `getData()`
  - `getPort()`
  - `getAddress()`
- `DatagramSocket`
  - constructors: port number/no argument
  - `send()`
  - `receive()`

    - `setSoTimeout()`
    - `connect()`

- see textbook for client/server e.g.

TCP Stream Communication

- **message sizes:** no limit on data size
- **lost messages:** acknowledgement scheme retransmits unacknowledged packets
- **flow control:** receive window; match speed between sender/receiver
- **congestion control:** prevent congestion collapse of network
- **duplication/ordering:** sequence numbers ensure duplicates are rejected and reordering occurs as necessary
- **destinations:** connection established before communication
- e.g. HTTP, FTP, Telnet, SMTP

Establishing TCP stream socket

- **client:**
    - create socket with server address + port
    - read/write data using stream associated with socket
- **server:**
    - create listening socket bound to server port
    - wait for clients to request connection: listening socket maintains a queue of incoming connection requests
    - server accepts a connection and creates new stream socket for the server to communicate with the client
- pair of sockets (client/server) now connected by pair of streams, one in each direction. A socket has an input stream and an output stream

Closing a socket

- data in output buffer sent to other end with indication stream is broken
- no further communication possible

Issues

- need pre-agreed format for data sent
- blocking is possible at both ends
- if the process supports threads, best approach is to assign a thread to each connection so that other clients are not blocked

Failure model

- checksum: detect/reject corrupt packets
- sequence number: detect/reject duplicates
- timeout + retransmission: lost packets
- severe congestion: TCP streams declare connection broken
  - breaks reliable communication
- communication broken: processes cannot distinguish between process failure and process crash
- communicating processes cannot definitely say whether messages sent recently were received
- clean exit: very confident all data received correctly

Java API

- `ServerSocket`
  - used to create a listening socket
  - `accept()`: gets connect request from queue, returns `Socket` instance
  - `accept()`: blocks until connection arrives
- `Socket`
  - used by pair of processes with a connection
  - client: uses constructor specifying DNS hostname:port, creating a socket bound to a local port and connects to remote computer
  - `getInputStream()`
  - `getOutputStream()`
- see textbook for TCP client/server

External Data Representation and Marshalling

- data structures need to be flattened to a sequence of bytes for transmission
- approaches to allow computers to interpret data
  - use agreed external format
  - transmit in senders format, with indication of format used
- **external data representation:** agreed standard for representing data structures and primitive data
  - CORBA common data representation
  - Java serialization
  - JSON
  - XML
- **marshalling:** process of converting data to form suitable for transmission
- **unmarshalling:** disassembling data at receiver
  - lots of validation required to ensure it conforms to expected format

CORBA's Common Data Representation

Java Object serialization

XML Extensible Markup Language

JSON JavaScript Object Notation

Group Communication

IP Multicast

Overlay Networks