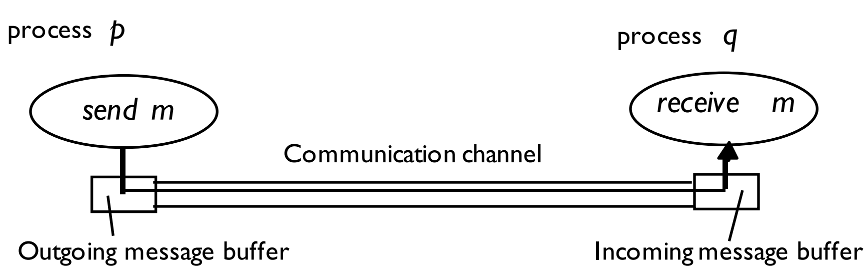
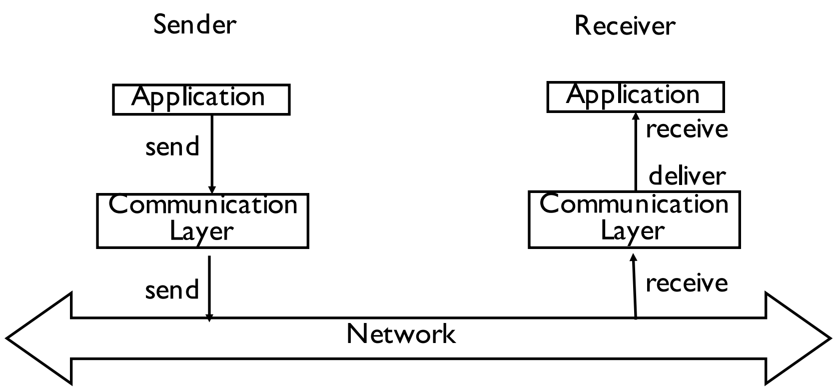
Communication Basics:
1) Inter-process Communication
2) Internet Stack

Inter-Process Communication



- One communication "unit" consists of two primitives
  - ☆ The **`send`** primitive is called by the sending process (caller, sender)
  - ☆ A corresponding **`receive`** primitive must be called by the receiving process (callee, receiver)

Basic Architecture



Blocking vs. Non-blocking send

- ❑ Non-blocking send
  - ☆ send operation returns and sending process is allowed to proceed as soon as the underlying communication layer has received the message and committed to process it
- ❑ Blocking send
  - ☆ send operation only returns once it is confirmed that at the receiver site the message
    - was delivered to the receiving process or
    - was received by the receiving process or
    - will be delivered to the receiving process or
    - (depends on definition)

Blocking vs. Non-blocking receive

- Blocking receive
  - ☆ The receive primitive blocks until a message arrives
  - ☆ Multi-threaded environment:
    - one receiving listener thread that has loop with blocking receive
    - other threads do other work…
- Message Handler
  - ☆ the application process provides the communication layer with a handler routine
  - ☆ the communication layer calls this routing upon message delivery
- Note: Book in chapter 4 calls this synchronous/asynchronous communication; here in class we do NOT

Source and Destination Adresses

- Abstract: process P
- Internet
  - ☆ address (=host) + port (location dependent)
  - ☆ Port is a message destination within a computer,
- Object (location independent)
  - ☆ location of object is determined at runtime
- Service (location independent);
  - ☆ Service name is translated at runtime to server location

Performance

- Network Latency
  - ☆ Delay between the start of transmission of a message over the network and the beginning of its receipt by receiving process
    - Time for the first bit to be transmitted through the network
- Process Latency
  - ☆ Network Latency +
  - ☆ Processing time at sender/receiver (communication layers)
    - Includes marshalling/unmarshalling
  - ☆ Other delays due to load at sender/receiver, network

Performance

- Data transfer rate / bandwidth
  - ☆ Speed at which data can be transferred between 2 computers
  - ☆ Bits/sec
- Jitter:
  - ☆ Variation in time taken to deliver a series of messages (often bursty delivery)

Synchronous vs. asynchronous

- Synchronous distributed system
  - each message is transmitted within a known bounded time
  - The time to execute each step of a process has known lower and upper bounds
  - Each process has a local clock whose drift rate from real time has a known bound
- Asynchronous distributed system
  - Message may need an arbitrary time to be transmitted
  - Each step of a process can take an arbitrary time
  - Clocks drift rates are arbitrary
- In the following: asynchronous model if not stated otherwise

Asynchronous Agreement

- Model:
  - Two armies A1 and A2 on top of two mountains. Enemy B in the valley
  - A1 and A2 can exchange messengers
    - synchronous: Each messenger needs at least *min* at most *max* time units
    - asynchronous but reliable: messengers will *eventually* arrive but there is no time limit on how long the messenger needs

Asynchronous Agreement

- Problem 1: A1 and A2 have to agree who starts attack

  - ☆ Solution:
    - both send message with number of soldiers,
    - wait until receive the message of the other;
    - the army with more soldiers starts attack

Asynchronous Agreement

- Problem2: A1 and A2 have to agree **when** to attack; they have to attack nearly at the same time (difference at most X time-units)

Assume A1 has to start attack

- Asynchronous:
  - ☆ A1 makes suggestion "at 2pm" or "in 20 minutes".
  - ☆ But no guarantee that message arrives before 2 pm or within 20 minutes.
- Synchronous model:
  - ☆ A1 makes suggestion "attack now", waits for *min* time units and attacks
  - ☆ After receipt of message, A2 attacks
  - ☆ A2 attacks at most *(max - min)* time units after A1
  - ☆ If *(max-min)* $\leq$ X, this works!

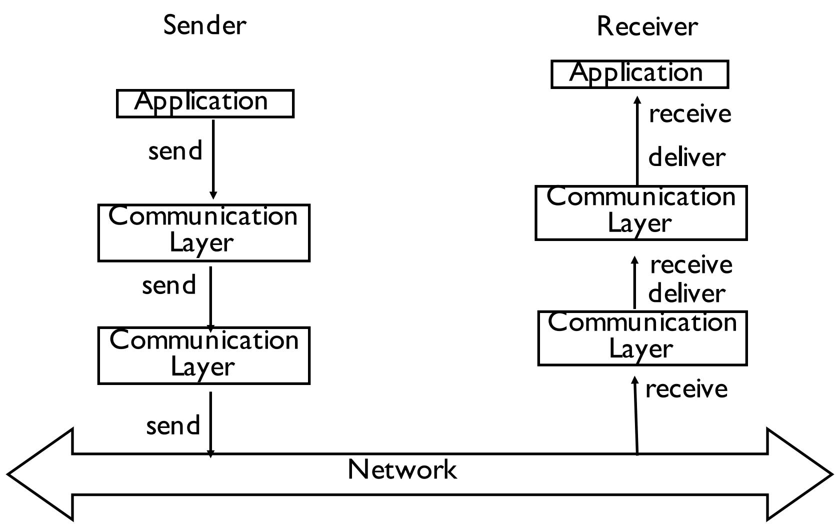
Possible failures

- *Omission failures:*
  - ☆ *process omission failure:*
    - *fail-stop:*
      - Process halts and remains halted;
      - Others can detect this state
    - crash:
      - Process halts and remains halted;
      - other processes many not be able to detect this
  - ☆ communication omission failures
    - receiving process does not receive message
    - typically because of
      - buffer overflow "somewhere": individual *message loss*
      - network partition
- *Arbitrary failures (byzantine failures)*
  - ☆ processes might not follow agreed protocol
  - ☆ messages might be corrupted
    - can be solved by checksums
- Process failures vs. network partitions: In general, it is impossible to distinguish between a process failure and a communication failure.

Reliability

- Reliable delivery:
  - ☆ Validity:
    - any message sent is eventually delivered to the application (unless the application fails)
  - ☆ Integrity:
    - the message received is identical to the one sent, and no messages are delivered twice
- Non-reliable delivery:
  - ☆ The communication primitive only provides a best effort. That is, when no failures occur, the message will arrive. But in case of failures, it might not arrive, be duplicated, etc
  - ☆ message identity typically provided
- Careful !!!
  - ☆ When you design a communication protocol with reliable delivery you must specify the failures that the protocol can handle
  - ☆ The protocol must work correctly if the specified failures occur
  - ☆ The protocol may not work if unspecified failures occur

Layered Architecture



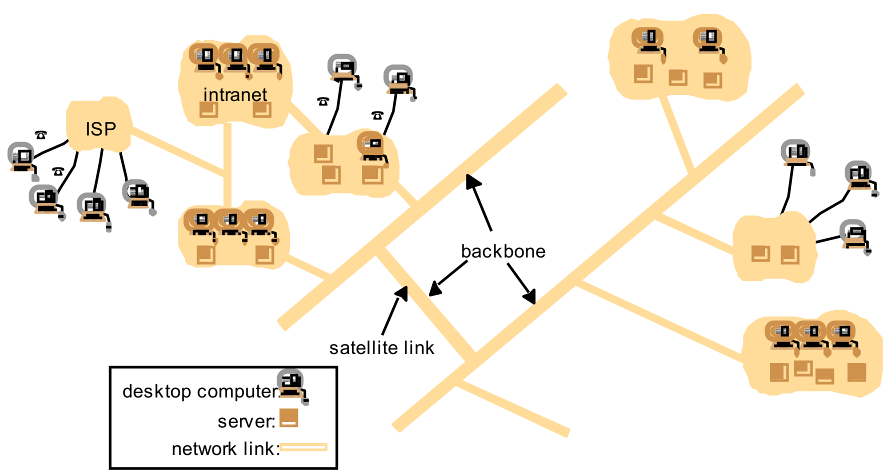
Simple Example

- Assume that
  - Processes never fail
  - No network partitions
  - Network provides unreliable n_send(q, message) / n_receive(message)
    - Messages can get lost
- Reliable communication module
  - Implement reliable send/receive using above unreliable n_send/n_receive pair
  - Basic ideas
    - Acknowledgements
    - Resubmissions
    - Message Identifiers

Extended

- What if processes can crash and messages can get lost?

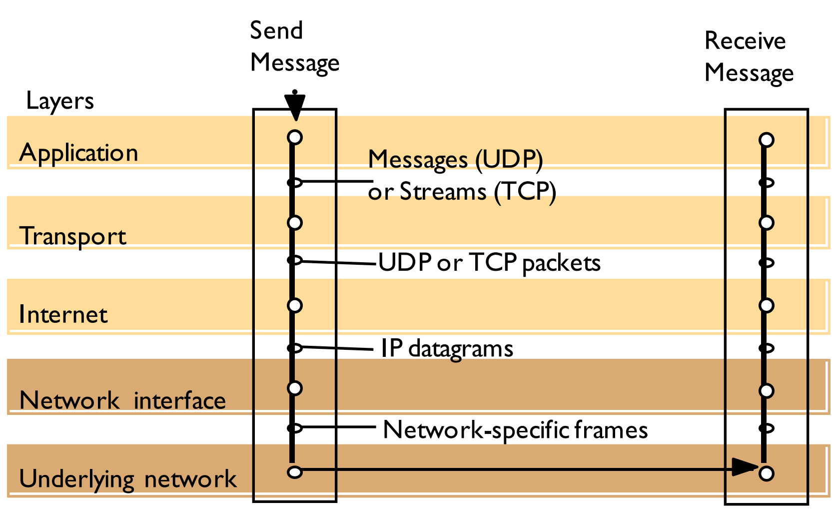
Networks



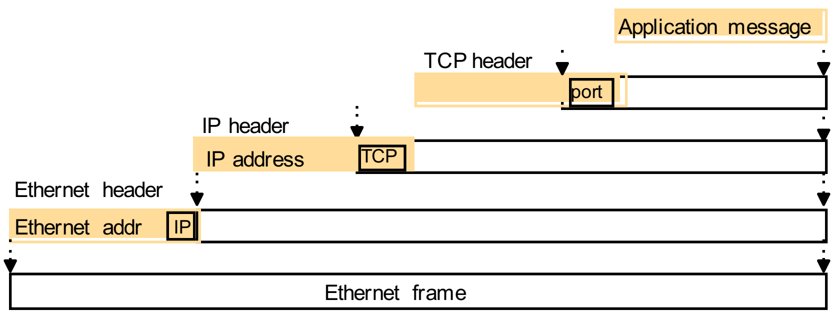
Network Types

	Range	*Bandwidth/ Throughput (Mbps)*	*Latency (ms)*
LAN	1-2 kms	10-10000	0.1-10
WAN	worldwide	0.010-600	100-500
MAN	2-50 kms	1-600	10
Wireless LAN	0.15-1.5 km	11-300	5-20
Wireless WAN	worldwide	348-14.4	100-500
Internet	worldwide	0.5-600	100-500

Communication Layers: Example TCP/IP



Message Format



Addresses

- IP Addresses: 32 bits
  - ☆ Network part: identifies subnet
  - ☆ Host part: identifies host on subnet
  - ☆ Hierarchical assignment
    - Organization is given a subnet identifier
    - Organization assigns host identifiers within subnet
- Routers/Bridges connect different subnets
  - ☆ they have an IP address for each subnet
- Addressing for portable computers
  - ☆ NAT: network address translation
  - ☆ unregistered addresses (not unique throughout the network)
  - ☆ special router does mapping
- Network addresses (physical addresses)

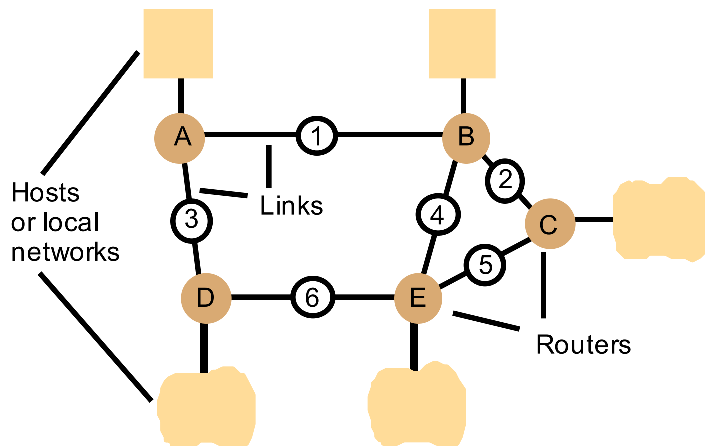
Network layer: Ethernet

- send message packet from node A to node B in same LAN
  - ☆ broadcast message on medium
  - ☆ controller hardware at each node listens on medium
  - ☆ pick up messages that are addressed to them and gives them
  - ☆ header: sender/destination (network addresses), …
- Packet Collision possible
  - ☆ resubmit after waiting certain time

IP protocol

- host-to-host transmission
- uses IP address
- unreliable, best-effort
  - ☆ Message can get lost somewhere
- Tasks:
  - ☆ transform IP address to network address of underlying network protocol
  - ☆ ROUTING

Routing in a wide area network



Routing Table

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Routings from B

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

Routings from C

To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D

To	*Link*	*Cost*
A	3	1
B	3	2
C	6	2
D	local	0
E	6	1

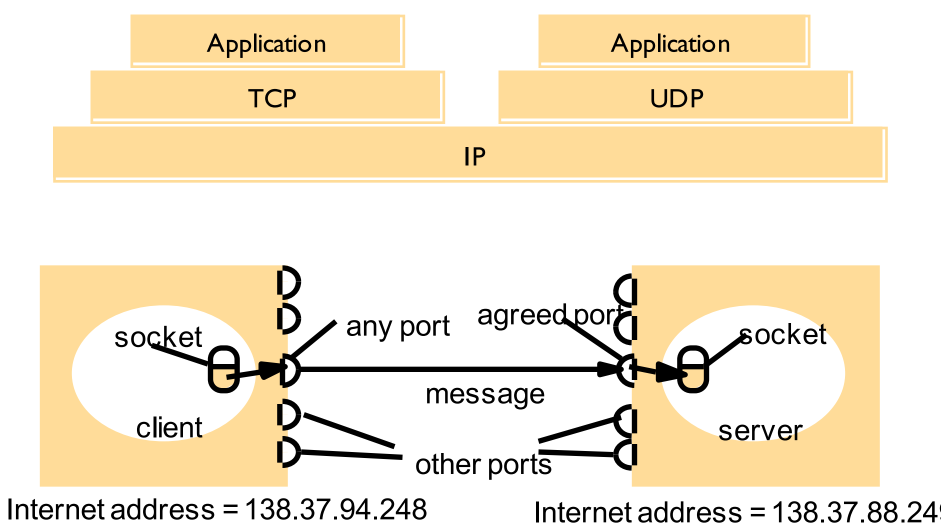
Routings from E

To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

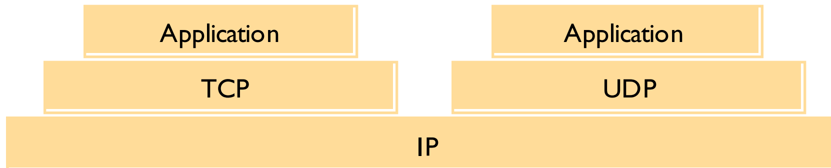
Issues

- ❑ Fast lookup of next hub
- ❑ maintenance
  - ☆ if link broken, set cost to infinity
  - ☆ send periodically routing table to others
    - others update their tables
- ❑ reasonable size
  - ☆ geographically oriented IP addresses
  - ☆ defaults

The programmers view



The programmers view



- UDP
  - ☆ thin software layer on top of IP which builds interface to application
  - ☆ same functionality as IP: unreliable, best-effort, message-oriented
  - ☆ checksums against message corruption: integrity
  - ☆ domain name resolution: application can use logical name (mimi.cs.mcgill.ca) instead of IP address
    - more on how domain names are translated into IP addresses later
      - ▲ (it' s a distributed application by itself!)
  - ☆ connection-less

UDP

- send(message byte string, message-length, IP receiver, port receiver)
  - ☆ UDP then adds IP sender, port sender
- receive(byte string into buffer, length, IP sender, IP port)
- Java
  - ☆ DatagramPacket: message, message length, IP address, port
    - IP address and port have different content for sending / receiving
  - ☆ DatagramSocket:
    - constructor with port as input (for server)
    - constructor without port as input (for client)
    - send and receive methods with DatagramPacket as argument

TCP

- connection-oriented:
  - Both processes first run a connection protocol
    - Handshake that both agree on building a communication channel
  - Only after connection is established, application level messages can be sent in both directions
- stream-oriented
  - both parties have input and output buffer
- reliable:
  - stream split into sequence of data segments (TCP messages) with sequence numbers
  - acknowledgement scheme:
    - receiver sends to sender periodically
      - highest sequence number in input stream + window size
    - Flow Control to avoid buffer overflow at receiver
      - sender can only send window size of data before next acknowledgement
    - retransmission of lost messages
      - sender keeps messages in output buffer until acknowledgment receives from receiver
      - if message not acknowledged within specified timeout, then it retransmits it.
      - duplicate detection using sequence numbers
- FIFO delivery
  - sequence numbers allow detection of out-of-order messages and reorder them before delivering them to the application

TCP

- Overview
  - Client creates socket with input and output stream
  - Server has special socket object with associated port
    - Server listens on socket for connection request message from client
    - Clients makes connection request to this socket
    - upon receiving request from Client C
      - create new socket with input and output stream only for communication with client C
- Java
  - ServerSocket: constructor has port as input
    - accept method returns new instance of Socket class which is already connected to client socket
  - Socket:
    - one constructor (used by client) has as input name and port of server
      - creates socket with local port and connects to ServerSocket
    - has associated input and output streams of certain types
      - (e.g., DataInputStream, ObjectInputStream, …)
    - input/output streams have methods: write/read

Data Representation in Messages

- Messages are a sequence of bytes -> data must be flattened
- Marshalling/serialization: transformation of a collection of data items into a form suitable for transmission
- Unmarshalling: transforming a message back into a collection of data items
- Common Mechanisms:
  - Java Serialization
  - XML / JSOC
  - CDR / Corba
  - Google Protocol Buffer

Java Declaration

```
Class Person implements Serializable {
    int id;
    string name;
    string place;
    int year;

    Public Person (int pid, string pname, string pplace, int pyear) {
    id = pid;
    name = pname
    place = pplace;
    year = pyear;
}

Person myPerson = new Person(12345, “Smith”, “London”, 1934);
```

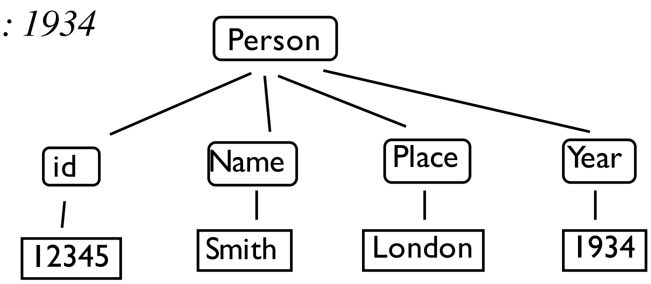
Java usage

- To exchange object of class Person via TCP
  - At sender
    - ObjectOutputStream oos = new ObjectOutputStream(socket);
    - oos.writeObject(myPerson);
  - At receiver
    - ObjectInputStream ois = new ObjectInputStream(socket);
    - readPerson = (Person) ois.readObject();

JSON

- textual format to present structured data

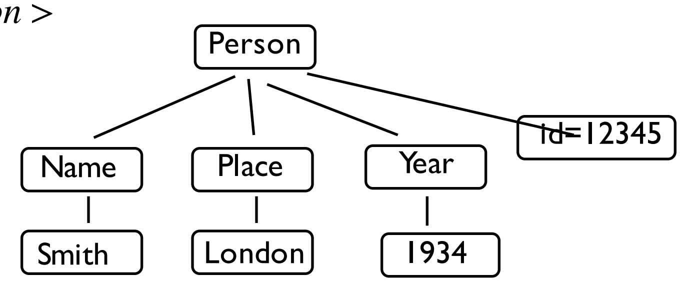
{
"person" : {
"id": 12345
"name" : "Smith"
"place" : "London"
"year" : 1934
}
}



XML

- textual format to present structured data

```
<person id="12345">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```



XML

- elements
  - ☆ basic building blocks
  - ☆ have start and end tag which indicate the name of element
  - ☆ elements can be nested
- attributes
  - ☆ element can have attributes
  - ☆ attribute is name/value pair
  - ☆ embedded in start tag of element
- text
  - ☆ part of an element
  - ☆ represents the content

Json/XML

- There exist many parsers for JSON/XML
  - e.g., Java software for conversion between Java objects and XML/JSON .

Google Protocol Buffer

```
message Person {
    required int32 id = 1;
    required string name = 2;
    optional string location = 3;
    optional int place = 4;
}
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_location(”London");
```

Google Protocol Buffer

- Programming Similar to Java
  - ☆ Output

```
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

  - ☆ Input

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
```


XML/JSON

- Transmitted in Ascii / Text Format
- Contains meta data and data

Internal Representation

- XML/JSON
  - ☆ Test / ascii
  - ☆ Human readable
  - ☆ Contains meta data and data
- Java
  - ☆ Serialized binary format
  - ☆ Contains information about data types

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

Internal Representation

- Google Protocol Buffer

```
message Person {
    required int32 id = 1;
    required string name = 2;
    optional string location = 3;
    optional int32 year = 4;
}
Person person;
person.set_id(1234);
person.set_name("Smith");
person.set_year(1934);
```

- Internal binary presentation
  - ☆ Contains coded info about (1) data type (2) attribute (3) real data

0 1 1234

2 2 "Smith"

0 4 1934

CORBA´s Data Representation

index in sequence of bytes	◄— 4 bytes —►	notes on representation
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

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

CORBA Marshalling

- message contains only the data but not the information about data types
- programmer can specify the structs that are the content of messages in an IDL (interface definition language) file
  - ☆ CORBA generates automatically marshalling and unmarshalling operations for these structs