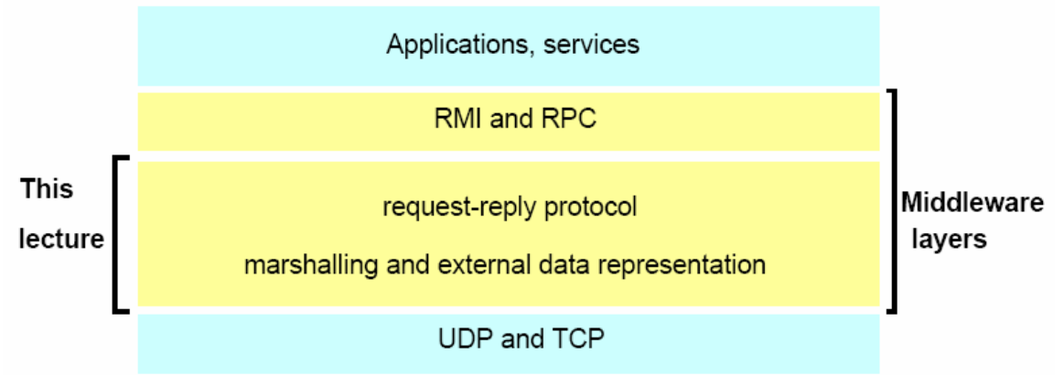
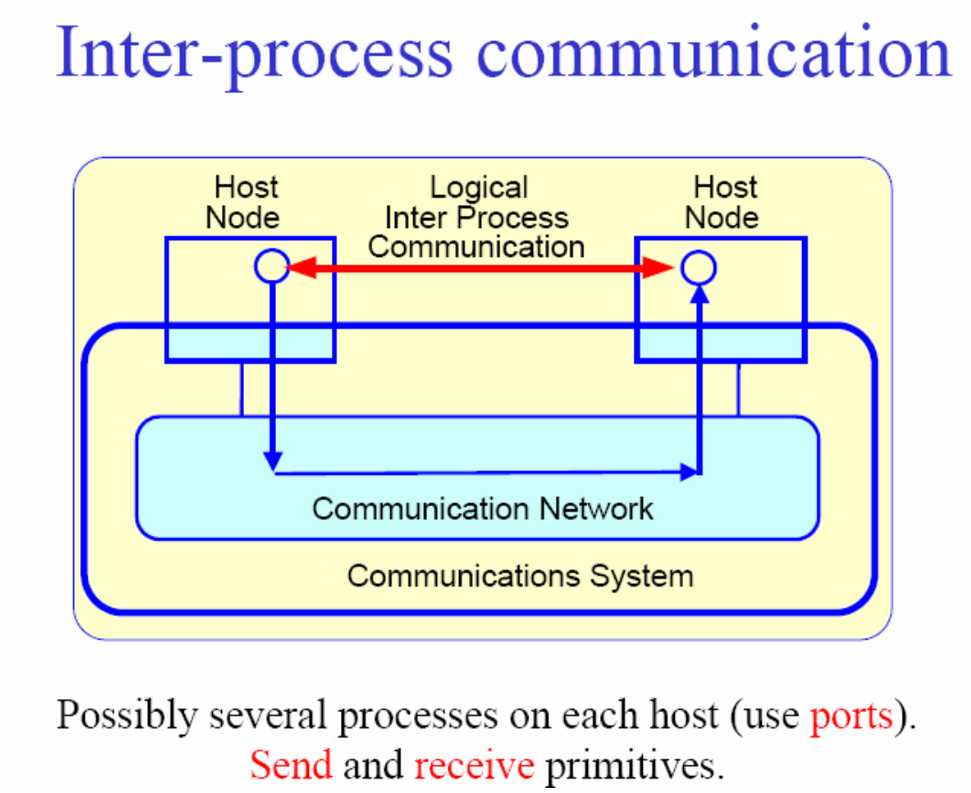
Komunikasi Antar Proses (Inter-Process Communication)

API untuk Pemrograman Internet



Komunikasi Antar-Proses (Inter-process communication)
• Sistem Terdistribusi
– terdiri dari komponen (processes, objects) yang berkomunikasi untuk kooperasi dan sinkronisasis
– melakukan pengeiriman pesan (message passing) karena tidak terdapat shared memory
• Middleware menyediakan dukungan bahasa pemrograman, yang memiliki sifat
– tidak mendukung low-level untyped data primitives (yg merupakan fungsi sistem operasi)
– mengimplementasikan higher-level language primitives + typed data



Communication service types
• Connectionless: UDP
– ‘send and pray’ pengiriman yang unreliable
– efisien dan mudah diimplementasikan
• Connection-oriented: TCP
– menjamin reliability
– kurang efficient, butuh memory dan time overhead untuk error correction

Connectionless service UDP (User Datagram Protocol)
– messages dimungkinkan hilang, duplicated, delivered out of order, tanpa pemberitahuan ke user
– tidak memelihara state information, shg tidak dapat mendeteksi lost, duplicate atau out-of-order messages
– setiap message mengandung alamat sumber dan tujuan

– dapat mengabaikan pesan discard corrupted untuk no error correction (simple checksum) atau congestion
Digunakan untuk DNS (Domain Name System) atau RIP.

Connection-oriented service TCP (Transmission Control Protocol)
– menyediakan data stream connection to meyakinkan reliable, pada urutan pengiriman
– error checking dan reporting pada kedua sisi (Client/Server)
– menyesuaikan kecepatan (timeouts, buffering)
– termasuk sliding window: state information
- unacknowledged messages
- message sequence numbers
- flow control information (matching the speeds)

Digunakan untuk HTTP, FTP, SMTP di Internet.

Timing pad Sistem Terdistribusi
• No global time
• Computer clocks
– memiliki beragam drift rate
– mengirim mll GPS radio signals (not always reliable), atau synchronise melalui clock synchronisation algorithms
• Mengurutkan Event (message sending, arrival)
– carry timestamps
– dimungkinkan tiba dgn urutan yang salah shg terjadi transmission delays (email)

Kesalahan pada Sistem Terdistribusi

Omission and arbitrary failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Tipe Interaksi
• Model interkasi Synchronous:
– mengetahui batas atas/bawah kecepatan eksekusi, message transmission delays dan clock drift rates
– lebih sulit dikembangkan, tetapi secara konsep merupakan model yg lebih sederhana,
• Asynchronous interaction model (more common, cf Internet, more general):
– arbitrary process execution speeds, message transmission delays dan clock drift rates
– beberapa masalah tidak mungkin dipecahkan (spt. agreement)
– jika solution valid utk asynchronous maka akan valid utk synchronous.

Send dan receive
• Send
– send suatu message ke socket bound utk suatu process
– dapat melakukan blocking atau non-blocking

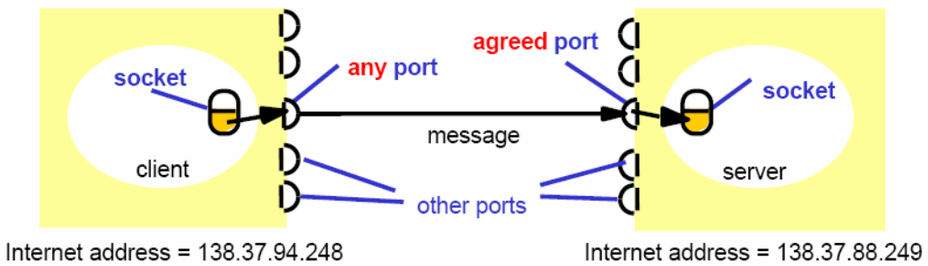
• Receive
  – receive suatu message pd a socket
  – dapat melakukan blocking atau non-blocking
• Broadcast/multicast
– send ke semua processes/all processes pada suatu group

Receive
• Blocked receive
– destination process di-blok hingga message arrives
– most commonly used
• Variations
– conditional receive (continue until receiving indication that message arrived or polling)
– timeout
– selective receive (wait for message from one of a
number of ports)

Asynchronous Send
• Characteristics:
– unblocked (process continues after the message sent out)
– buffering needed (at receive end)
– mostly used with blocking receive
– usable for multicast
– efficient implementation
• Problems
– buffer overflow
– error reporting (difficult to match error with message)
Maps closely onto connectionless service.

Synchronous Send
• Characteristics:
– blocked (sender suspended until message received)
– synchronisation point for both sender & receiver
– easier to reason about
• Problems
– failure and indefinite delay causes indefinite blocking (use timeout)
– multicasting/broadcasting not supported
– implementation more complex
Maps closely onto connection-oriented service.

Sockets and ports



Socket = Internet address + port number.
Only one receiver but multiple senders per port.
Disadvantages: location dependence (but see Mach study, chap 18)
Advantages: several points of entry to process.

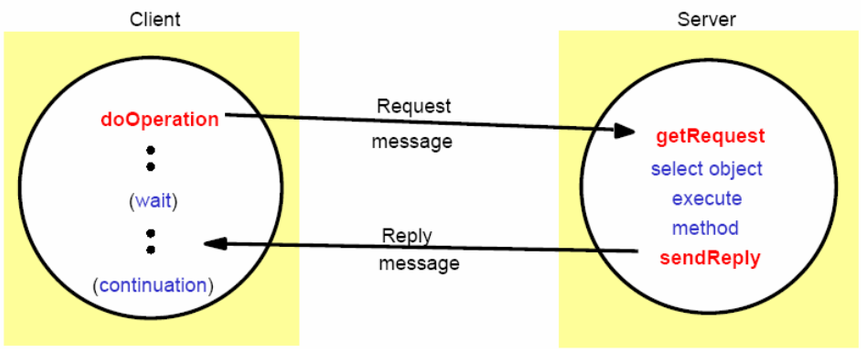
Sockets
• Characteristics:
– endpoint for inter-process communication

– message transmission between sockets
– socket associated with either UDP or TCP
– processes bound to sockets, can use multiple ports
– no port sharing unless IP multicast
• Implementations
– originally BSD Unix, but available in Linux, Windows,…
– here Java API for Internet programming

Client-Server Interaction
• Request-reply:
– port harus diketahui oleh client processes (biasanya dipublish oleh server)
– client memiliki private port untuk menerima reply
• Skema Lainnya:

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Request-Reply Communication



Operations of Request-Reply

- *public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)*
  - sends a request message to the remote object and returns the reply.
  - the arguments specify the remote object, the method to be invoked and the arguments of that method.
- *public byte[] getRequest ();*
  - acquires a client request via the server port.
- *public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*
  - sends the reply message reply to the client at its Internet address and port.

Java API for Internet addresses

- Class *InetAddress*
  - uses DNS (Domain Name System)

InetAddress aC =
InetAddress.getByName("gromit.cs.bham.ac.uk");

  - throws *UnknownHostException*
  - encapsulates detail of IP address (4 bytes for IPv4 and 16 bytes for IPv6)

Remote Object Reference

An identifier for an object that is valid throughout the distributed system

- must be unique
- may be passed as argument, hence need external representation

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Java API for Datagram Comms

- Simple send/receive, with messages possibly lost/out of order
- Class *DatagramPacket*

message (=array of bytes)	message length	Internet addr	port no

  - packets may be transmitted between sockets
  - packets truncated if too long
  - provides *getData*, *getPort*, *getAddress*

Java API for Datagram Comms

- Class *DatagramSocket*
  - *socket constructor* (returns free port if no arg)
  - *send* a *DatagramPacket*, non-blocking
  - *receive DatagramPacket*, blocking
  - *setSoTimeout* (receive blocks for time T and throws *InterruptedIOException*)
  - *close* DatagramSocket
  - throws *SocketException* if port unknown or in use

UDP client example

```
public class UDPClient{
public static void main(String args[]){
// args give message contents and server hostname
 DatagramSocket aSocket = null;
  try {     aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new
                        DatagramPacket(m,args[0].length(),aHost,serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
  }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
  }catch (IOException e){System.out.println("IO: " + e.getMessage());}
} finally {if(aSocket != null)  aSocket.close(); }
}}
```

UDP server example

```
public class UDPServer{
  public static void main(String args[]){
   DatagramSocket aSocket = null;
     try{
       aSocket = new DatagramSocket(6789);
       byte[] buffer = new byte[1000];
       while(true) {
            DatagramPacket request = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(request);
            DatagramPacket reply = new DatagramPacket(request.getData(),
                        request.getLength(), request.getAddress(), request.getPort());
            aSocket.send(reply);
                  }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
}
```

Java API for Data Stream Comms

- Data stream abstraction
  - attempts to match the data between sender/receiver
  - marshaling/unmarshaling
- Class *Socket*
  - used by processes with a connection
  - *connect,* request sent from client
  - *accept,* issued from server; waits for a connect request, blocked if none available

Java API for Data Stream Comms

- Class *ServerSocket*
  - socket constructor (for listening at a server port)
  - *getInputStream, getOutputStream*
  - *DataInputStream, DataOutputStream* (automatic marshaling/unmarshaling)
  - *close* to close a socket
  - raises *UnknownHost*, *IOException*, etc

Data Marshaling/Unmarshaling

- Marshaling (=conversion of data into machine-independent format)
  - necessary due to heterogeneity & varying formats of internal data representation
- Approaches
  - CORBA CDR (Common Data Representation)
  - Java object serialisation, cf DataInputStream, DataOutputStream on previous and next slides

TCP client example

```
public class TCPClient {
        public static void main (String args[]) {
          // arguments supply message and hostname of destination
          Socket s = null;
           try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding, see Sec 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
                s.close();
           }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
           }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
           }catch (IOException e){System.out.println("IO:"+e.getMessage());}
         }finally {if(s!=null} try {s.close();}catch (IOException e)….}
```

TCP server example

```
public class TCPServer {
   public static void main (String args[]) {
        try{
                int serverPort = 7896;
                ServerSocket listenSocket = new ServerSocket(serverPort);
                while(true) {
                        Socket clientSocket = listenSocket.accept();
                        Connection c = new Connection(clientSocket);
                }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream( clientSocket.getInputStream());
                out =new DataOutputStream( clientSocket.getOutputStream());
                this.start();
           } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
          try {                                     // an echo server
                String data = in.readUTF();
                out.writeUTF(data);
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally {try {clientSocket.close();}catch (IOException e)…..}
}
```