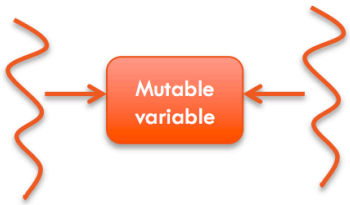
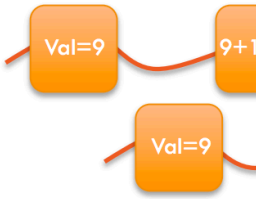
Lecture2

Monday, 14 August 2017 2:06 PM

Multi threading:

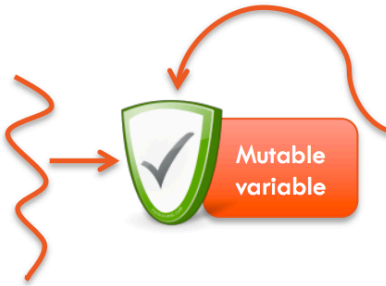
- Thread safety:
  - Thread safety is about protecting data from uncontrolled concurrent access
    - An object state is its data, stored in state variables such as instances or sta
    - A shared variable is a variable that could be accessed by multiple threads
    - A mutable variable is a variable whose value may change
    - Whenever more than one thread access a given state variable and one of they all must coordinate their access to it using synchronization
    - If multiple threads access the same mutable state variable without approp program is broken
    - 


```
public class UnsafeSequence {
  private int value;

  // Returns a unique value
  public int getNext() {
    return value++;
  }
}
```



      - Multiple threads accessing a mutable variable may produce erroneo
  - Potential solutions:
    - Do not share the state variable among threads, make it thread specific
    - Make the state variable mutable (locked)
    - Use synchronization to access the state variable
    -

```
public class UnsafeSequence {
  private int value;

  // Returns a unique value
  public synchronized int getNext() {
    return value++;
  }
}
```



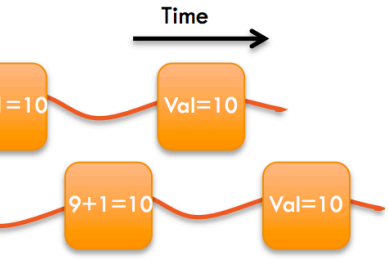
Using synchronization

  - Thread-specific data:

atic fields

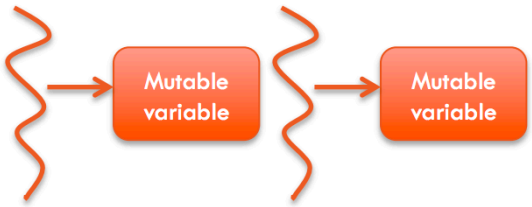
them might write to it,

priate synchronization, the



ous results

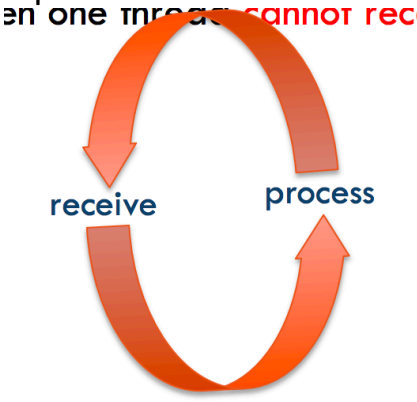
- Threads belonging to the same process share data
- How to declare data to be "thread-specific"?
  - By inheriting from thread into Mythread in which data is declared
  - By using the ThreadLocal class initialized by initialValue()/set() and a get()/set()



Thread-specific variables

```
public class Service {
 private static ThreadLocal errorC

 private static void transaction() {
  try {
    // some operation where an er
  } catch (Exception e) {
    errorCode.set(e);
  }

  // get the error code for this tran
  public static Object getErrorCod
    return errorCode.get();
  }
}
```

- Example (producer/consumer threads):
  - Problem:
    - Assume that we may receive requests almost simultaneously
    - It takes us same non-null amount of time to process a request
    - Yet, we would like to eventually process all requests
    - How to proceed when one thread cannot receive while processing a



By the time it can receive again, some request may be lost

    - Solution:
      - Use multithreading in a consumer producer scenario
        - Ideal for asynchronous communication (mailbox != pho threads
        - Decouples the consuming and production that run at di
        - One thread typically passes the work that arrives to som
        - The producer loop can receive a request while the prod

accessed through

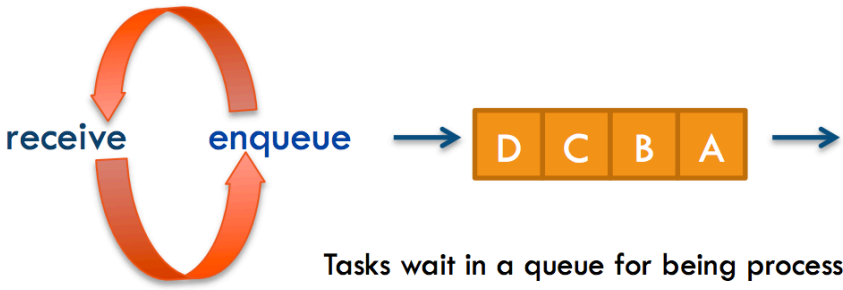
```
ode = new ThreadLocal();


rror may occur


nsaction
e() {
```

request?

ne analogy) between

fferent rates
ne other thread
ucer loop is processing

another



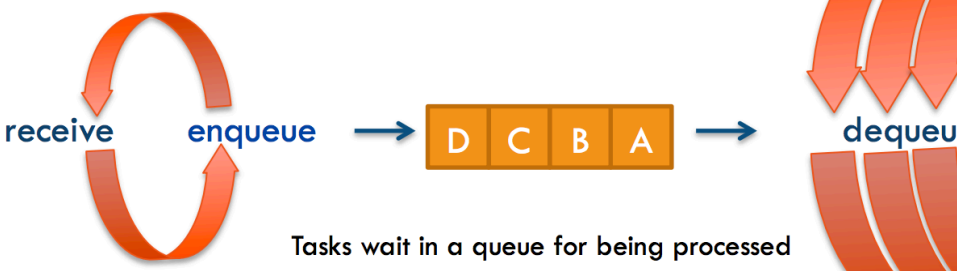
Tasks wait in a queue for being process

Producer thread:
Rapidly takes
inputs to not miss any

- One thread acts as a consumer
- Another thread acts as a producer

```
import java.util.Date;

public class Factory {
  public static void main(String args[]) {
    // create the message queue
    Channel<Date> queue = new MessageQueue<Date>();

    // create the producer and consumer threads and pass
    // each thread a reference to the MessageQueue object.
    Thread producer = new Thread(new Producer(queue));
    Thread consumer = new Thread(new Consumer(queue));

    producer.start();
    consumer.start();
}
```

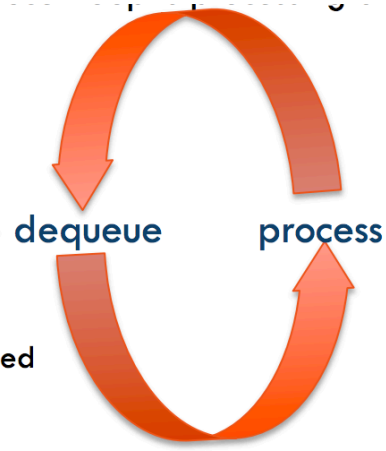
- Speeding up the request processing using multiple consumer thread



Tasks wait in a queue for being processed

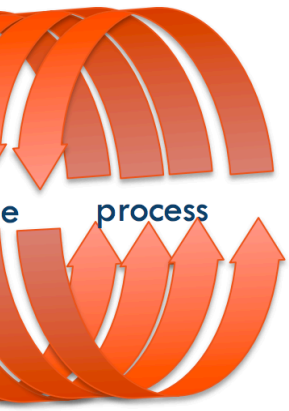
Producer thread:
Rapidly takes
inputs to not miss any

Multipl
Slov
but



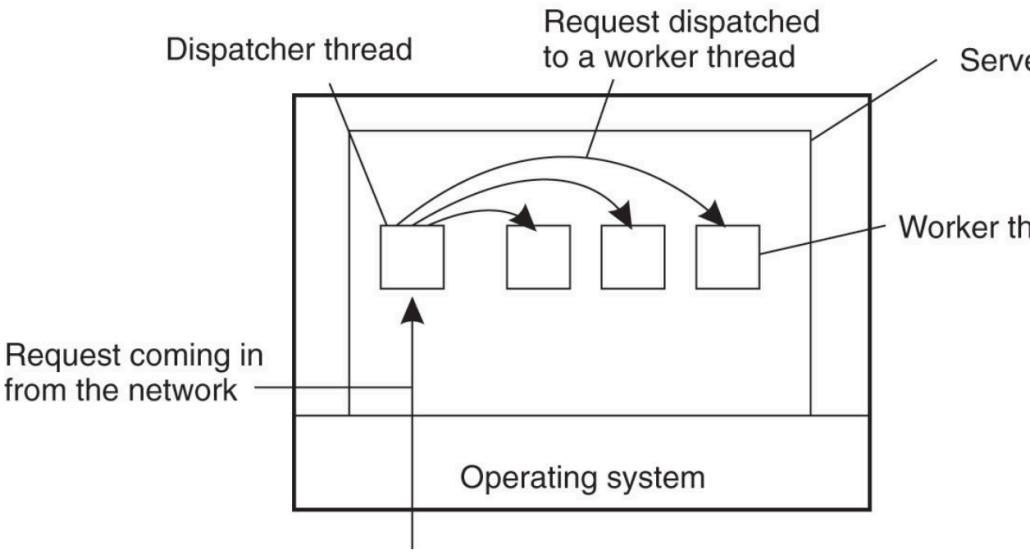
Consumer thread:
Slowly execute tasks

ds



le consumer threads:
wly execute tasks,
multiple at a time

- Multithreaded server
  - A multithreaded server organized in a dispatcher/ worker model
    - One thread, the dispatcher, reads incoming clients requests
    - The server chooses an idle worker thread and hands it the request



  - Other alternatives:
    - Single-threaded server:
      - One loop iteration must treat a request before starting to treat anot request losses
    - Finite state machine:
      - The server treats the request if the requested info is in the cache, ot for I/O to disk to complete but stores the state of the request in a ta system calls

Model	Characteristi
Threads	Parallelism, blocking sys
Single-threaded process	No parallelism, blocking
Finite-state machine	Parallelism, nonblocking

  - Creating threads upon request:
    - For each request that arrives at the server
    - Creates the request in a separate thread

```
...
  // keep treating incoming request
  while (true) {
   // block until accepting a new request
   try { client.socket = this.serverSocket.accept();
   } catch (IOException ioe) {System.out.println(ioe); }
```

er

read

ther one => potential

therwise it does not wait
able. It uses non-blocking

cs
tem calls
system calls
system calls

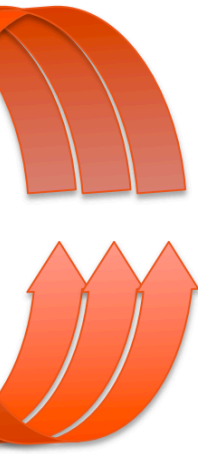
```
        // start a new task in a new thread
        new Thread(new MyThread(new Task()).start();
      }
    ...
```

Main process thread: spawns one thread upon request

3 threads f

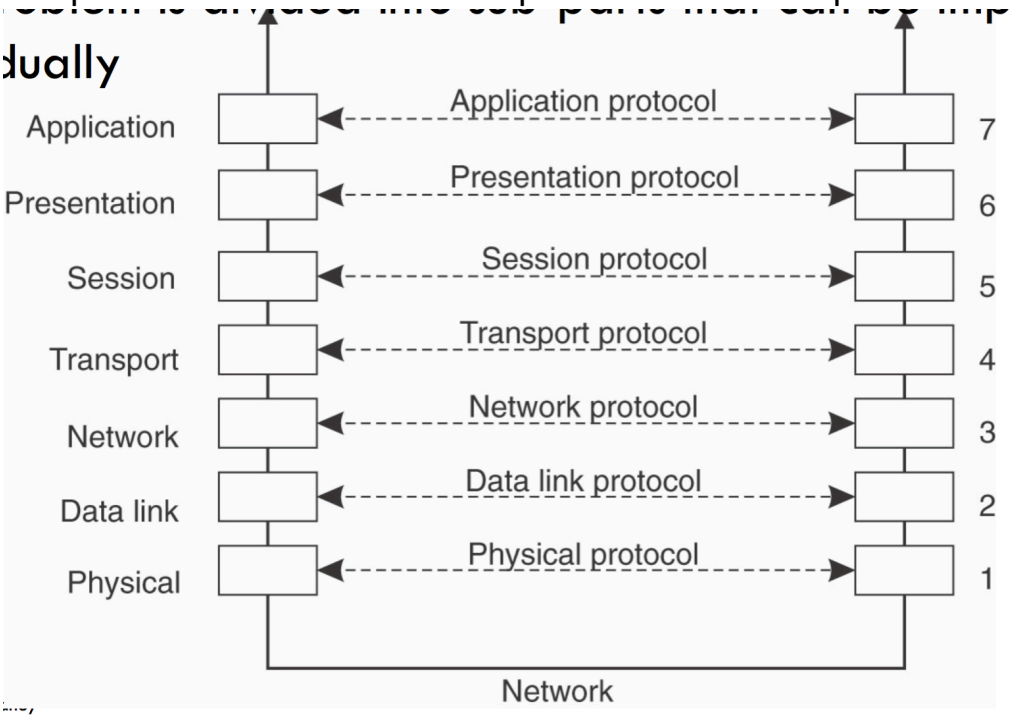
- Thread pool:
  - Creates a pool of threads
  - That the server can (re)use
  - No need to initialized many threads
  - Not to destroy many threads
  -

```
import java.util.concurrent.*;

public class TPExample {

   public static void main(String[] args) {
     int numTasks = Integer.parseInt(args[0].trim());

     // create the thread pool
     ExecutorService pool = Executors.newCachedThreadPool();

     // Run each task using a thread in the pool
     for (int i=0; i<numTasks; i++)
        pool.execute(new Task());

     // Shut down the pool. This shuts down the pool only
     // after all threads have completed.
     pool.shutdown();
   }
}
```



or 3 tasks

Communication:

- Layered protocol:
  - Communication problem:
    - Communicating is about transmitting encoded information:
      - Problems:
        - Two computers must agree on the code (language) they use
      - Open standards give rules for everyone to communicate with each
        - International standard organization (!SO)
        - Internet engineering task force (IEFT)
      - Private "closed" protocols exist
        - Example: Skype
  - Two modes of communication:
    - Connection oriented:
      - Establish explicitly a connection with a partner before exchanging
      - Protocol example: transmission control protocol (TCP)
      - Application usage: file transfer, web browsing, email
    - Connectionless:
      - No setup in advance is needed
      - Protocol example: user datagram packet (UDP, IP)
      - Application usage: VoIP (skype), IPTV
  - Layers:
    - OSI layer:
      - Each layer deals with one specific aspect of the communication
      - The problem is divided into sub-parts that can be implemented inc
      - 

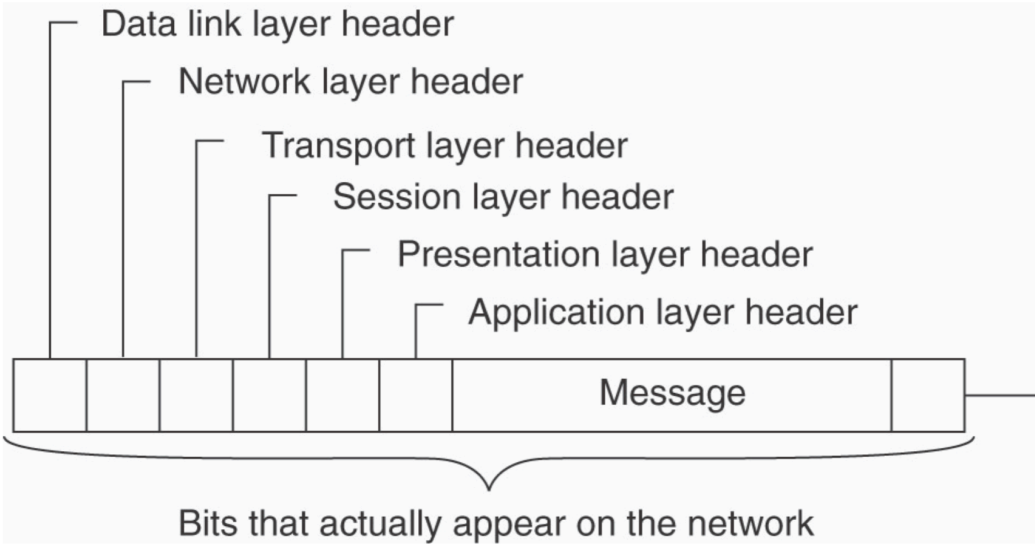
    - Message structure:

n other

data

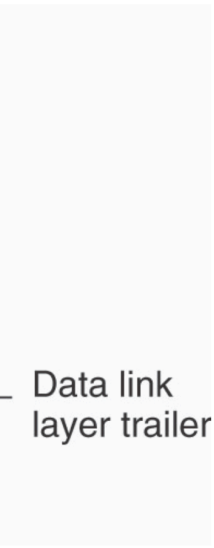
dividually

Message structure:

- As the message go through lower layers before being sent, each la
- Upon reception, the message is unmarshalled by the successive lay
- 

- Network communication:
  - Low-level layers
    - Physical layer: send bits (0 and 1)
    - Data link layer:
      - Groups bits into frames
      - Assigns sequence numbers to frames and adds special bits a
      - Adds a checksum (the result of some operation on the frame
      - If receiver disagree about the checksum, then it asks the sen
      - Example:
        - Ethernet for local area network (LAN) or PPTP in virtua
  - Transport and higher level layers:
    - Transport layer:
      - Split the message from the application layer
      - Number them
      - Add the amount of sent and remaining packets
    - Reliable transport layer can be built on top of:
      - Connection-oriented protocol: packet would be ordered
      - Connectionless protocol: packet could be reordered
    - Example of transport layer protocol: transmission control protocol
    - Example of transport layer and network layer combination: TCP/IP
  - High level layers:
    - OSI specifies three higher level layers:
      - Session layer: dialog control, transfer check-pointing
      - Presentation layer: says how information is represented
      - Application

yer adds its header to the message
yers from bottom to top



t the beginning and end
e content)
nder to resend

al private network(VPN)

(TCP)

- Session:
  - Example:
    - Domain name service (DNS)
    - Lightweight directory protocol (LDAP)
  - Sometimes the session layer is omitted: internet protocol su
- Application:
  - hypertext transfer protocol (HTTP)
  - File transfer protocol (FTP)
  - TCP/IP terminal emulation protocol (telnet)
  - X -window

- Analogy: mail service VS. web service:
  - Application (no clue of intermediary steps):
    - Post a letter with some address, the receiver reads it
    - HTTP: a user types a URL in a browser, a server sends back t
  - Transport (error control):
    - Upon writing a wrong address on a letter, the letter will be s
    - TCP initialized a connection, checks for potential error and m
  - Internet (recipient is unknown):
    - An airplane moves letters between cities without knowing th
    - IP brings packets over the WAN potentially from one LAN to
  - Data link:
    - Trucks move letters within a city
    - Ethernet handles transmission within the LAN
  - Physical layer:
    - Use pen to write and glasses to read letter

IPv6

Internet Protocol version 6 (IPv6): new version of IP based on IPv4

- IPv6 packets are carried over Ethernet with the content type 86DD (hexadecimal) i IPv4's 0800.

End 2011, there ~2.3x10^9 u

- Scalability in the number of addresses
  IPv6 increases the IP address size from 32 bits (4,294,967,296 in total) to 128 bits
  - more levels of addressing hierarchy, a much greater number of addressable nodes
  - simpler auto-configuration of addresses

- Scalability of multicast addresses is introduced
  A new type of address called an *anycast address* is also defined, to send a packet of a group of nodes

- Efficient forwarding

ite

he web page

end back to you
nay retransmit

ne recipients
another

nstead of

e were
users

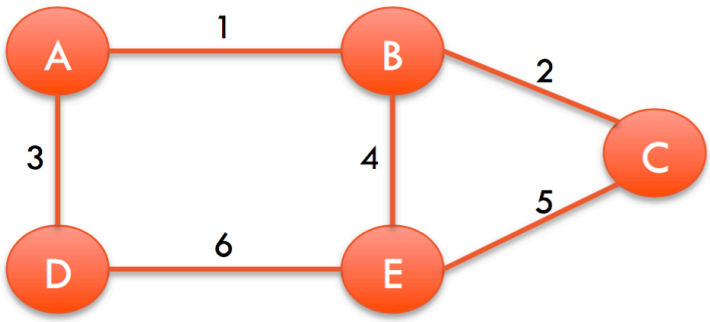
, to support
and

to any one

- Efficient forwarding
  Changes in the way IP header options are encoded: Hop-by-Hop Option, Routing (T
  Fragment, Destination Option, Authentication, Encapsulation Payload

---

- Routing:
  - Routing problem:
    - The use of routing protocols:
      - Related to path finding problem in graphs
      - Graph nodes represent internet routers, edges are communicatior
      - Dijkstra
    - Two main classes of protocols/algorithm:
      - Distance-vector: neighbors exchange their routing table
      - Link-state: neighbors only exchange connectivity information
    - Routing is necessary in all networks except local area networks (LANs) w
      attached hosts
    - The routing protocol is implemented in the network layer of each router
      destination
    - In large networks, adaptive routing is employed: the best route for comr
      periodically, taking into account the current traffic in the network
  - Distance-vector routing algorithm:
    - Router information protocol (RIP):
      - Mainly used in internet up to 1979
      - Relies on Bellman-Ford algorithm: Bellman algorithm [1957] dist

      - General idea: maintain a table representing the direction for ea
        1. Every period t (=30sec) and whenever the local routing table T_L chang
        2. Upon reception of a table T_R with a new destination or a lower-cost rou

        If a node detects a link failure, it sets ∞ as the associated cost of such a link

        Eventually (when failures stop) each router gets for each destinat

Type 0),

n links

here Ethernet provides direct communication between all pairs of

to determine the route for the transmission of packets to their

munication between two points in the network is re-evaluated

ributed by Ford and Fulkerson [1972]

Routing table of router A		
Dest.	**Dir**	**Cost**
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

ach destination

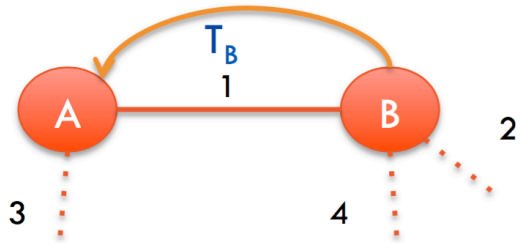
es, send T_L to all accessible neighbours

ute to an existing destination, update the local table T_L

and sends its local table to neighbours

ion the direction leading to the minimal cost

Example: A receives table T_B from B



- Routing table T_L of router A

Dest.	Dir	Cost
A	local	0
D	3	1

- Received routing table T_R

Dest.	Dir	Cost
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

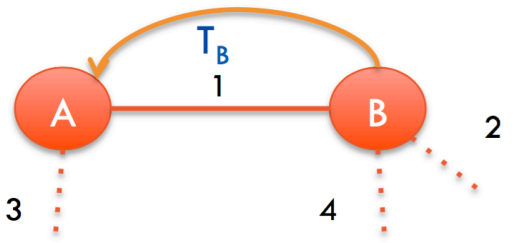
Each router execute
1. Send: Each t se non-faulty out
2. Receive: Wher

```
for all rows R_R in
  if (R_R.dir ≠ n) {
    R_R.cost = R_R
    R_R.dir = n
    // add new
    if (R_R.dest i
    else for all
      if (R_R.des
    (R_R.cost < R_L
    R_L = R_R
      // R_R.co
      // R_L.di
    }
  }
}
```

- Advantages:
  - Simple
  - Efficient in small networks
- Limitations:
  - Costs based on the number of hops is unrealistic (bandwidth
  - No big deal: this is just a matter of defining link weights (sup
  - Inefficient in large networks as loops mat occur before the c
- To address this inefficiency more scalable routing protocols, called

- Link-state routing algorithm:

Example: A receives table T_B from B



- Routing table T_L of router A

Dest.	Dir	Cost

- Received routing table T_R

Dest.	Dir	Cost

Each router executes
1. Send: Each t sec non-faulty outg
2. Receive: Whene

```
for all rows R_R in T
  if (R_R.dir ≠ n) {
    R_R.cost = R_R.c
    R_R.dir = n
    // add new d
    if (R_R.dest is
    else for all ro
      if (R_R.dest
    (R_R.cost < R_L.c
```

es the same code:
econds or when T_L changes, send T_L on each
going link

never a routing table T_R is received on link n:
T_R {

.cost + 1

destination to T_L
s not in T_L) add R_R to T_L
rows R_L in T_L {
t = R_L.dest **and**
.cost **or** R_L.dir = n))

ost < R_L.cost: remote node's better route
r = n: remote node is more authoritative

n of links counts)
pport negative values)
onvergence state is reached
linked state, were proposed

s the same code:
onds or when T_L changes, send T_L on each
oing link

ver a routing table T_R is received on link n:
$_R$ {

ost + 1

estination to T_L
not in T_L) add R_R to T_L
ws R_L in T_L {
= R_L.dest **and**
ost **or** R_L.dir = n))

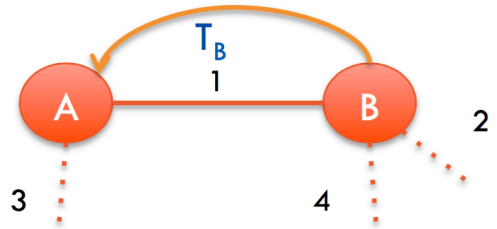
A	local	0
D	3	1

A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

```
            R_L = R_R
                // R_R.cost
                // R_L.dir =
            }
        }
    }
```

- Example:

Example: A receives table T_B from B



- ◆

Routing table T_L of router A		
Dest.	**Dir**	**Cost**
A	local	0
D	3	1

Received routing table T_R		
Dest.	**Dir**	**Cost**
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

```
Each router exec
1. Send: Each
   non-faulty c

2. Receive: Wh
   for all rows R_R
     if (R_R.dir ≠ n
        R_R.cost =
        R_R.dir = r
        // add ne
        if (R_R.des
        else for a
           if (R_R.d
        (R_R.cost <
           R_L = R_R
              // R_R
              // R_L
        }
     }
   }
```

- Socket:

  Definition

  - *Socket*: a communication end-point to which an application can wr that is to be sent out over the underlying network, and from which data can be read
  - A pair of processes communicating over a network employ a pair sockets, one for each process
    - ○
  - A socket is identified by an IP address concatenated with a port n
  - In general, sockets uses the client-server model:
    - The server waits for incoming client requests by listening to a specified p
    - Once a request is received, the server accepts a connection from the clie to complete the connection

$< R_L$.cost: remote node's better route
= n: remote node is more authoritative

cutes the same code:
t seconds or when T_L changes, send T_L on each
outgoing link

henever a routing table T_R is received on link n:
in T_R {
) {
R_R.cost + 1
n
ew destination to T_L
st is not in T_L) add R_R to T_L
all rows R_L in T_L {
lest = R_L.dest **and**
R_L.cost **or** R_L.dir = n))

.cost < R_L.cost: remote node's better route
.dir = n: remote node is more authoritative

ite data
incoming

of

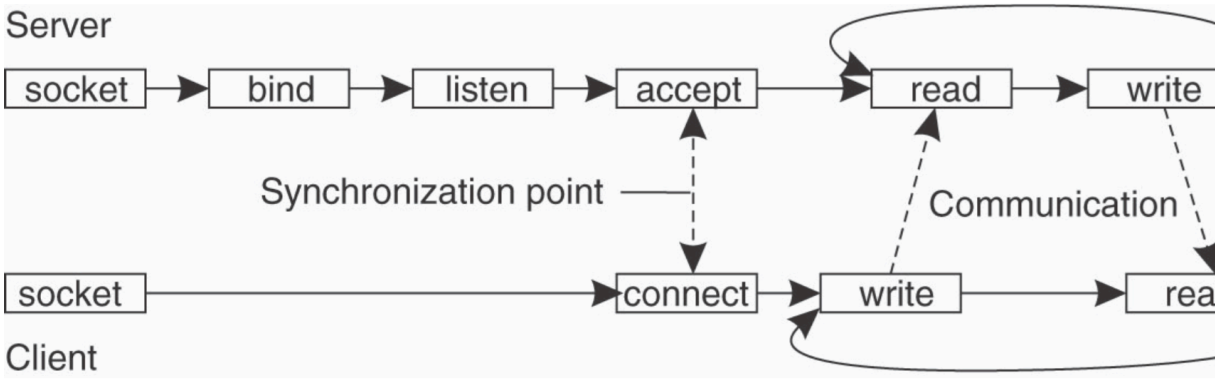
umber

ort
ent socket

Berkley Sockets: Socket interface as proposed in Berkley UNIX in the 70

1. Servers generally execute the first 4 primitives and block on accept
2. Bind associates the newly created socket to a local address and a port; the client binds im available port
3. The client connects to a specified address which blocks until response
4. Once the connection is accepted by the server, the system creates a new socket with the so the original one; the client and server can communicate with send and receive.

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection



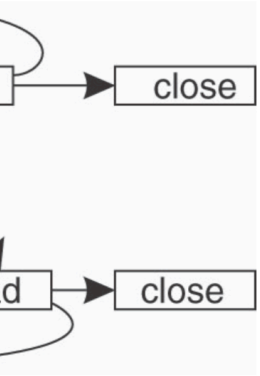
Port

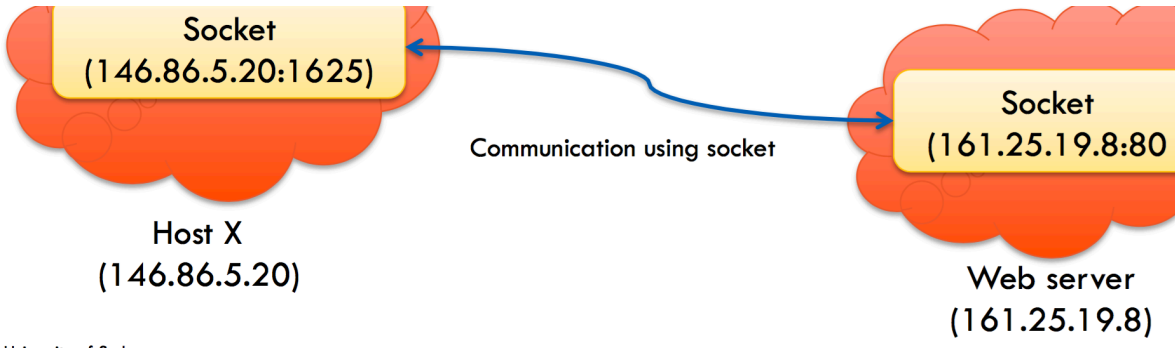
- Port numbers <1024 are for specific service protocols e.g., 80:HTTP, SSH:22, FTP:21, SMTP:25
- A client initiating a connection is assigned a free port number (>1024) by its host

's

plicitly to any

ame properties as





Java Sockets

- Three socket classes:
  - DatagramSocket: Connectionless (UDP) socket, to open a lightwei communication favoring throughput rather than reliability
  - MulticastSocket: Multicast socket (subclass of DatagramSocket) t to multiple recipients
  - Socket: Connection-oriented (TCP) socket, to open a connection fo communication

```
public class Server {

  public static void main(String[] args) {
    try {
     // Create a socket on port 6013
     ServerSocket sock = new ServerSocket(6013);
     while (true) {
       Socket client = sock.accept();
       ... // communicate with the client
       client.close(); // close socket
     }
    } catch (Exception e) {
       System.err.println("I/O problem");
    }
  }
}
```

```
public class Client {

  public static void main(String[]
    try {
      // Specify server address a
      Socket sock = new Socket(


      ... // communicate with th
      sock.close(); // close socke

    } catch (Exception e) {
      System.err.println("server
    }
  }
}
```

- Message-passing interface (MPI):
  - MPI vs Socket:
    - Socket:
      - Limited set of primitives, providing only send and receive for comm
      - Exploits general purpose communication layered protocol
      - Example:

ight

o send data

r reliable

```
 args) {

nd port for socket
"127.0.0.1", 6013);

e server
t

? or I/O problem");
```

munication

- TCP/IP aims at being tolerant to message losses

- MPI:
  - Hardware and platform independent
  - Direct use of underlying network (no multiple layers)
  - Assume no failures, communication involves a group of processors

Some of the MPI communication primitives:

- Blocking synchronous send (MPI_ssend) and asynchronous send (MPI_bsend)
- RPC-like mechanism (MPI_sendrecv)
- Pointers passed as a parameter to MPI_isend and MPI_issend, MPI takes care o
- Blocking recv (MPI_recv) and non-blocking one (MPI_irecv)
- Variety of communication primitives allows for optimizations but makes it more c

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or rem
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until r
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not bl

MPI is used in high-performance parallel applications

- Message-oriented middleware (MOM):
  - **Characteristics:**
    - Do not require sender and receiver to be active during communication (= cf. next week)
    - Offering intermediate storage capacity for messages
    - Typically target long (> minutes) message transfers
  - **Principles:**
    - Each application maintains its local queue
    - And sends messages to other application queues
    - Unaware of the reading of its messages (recipient can be down at send time)

s

f communication

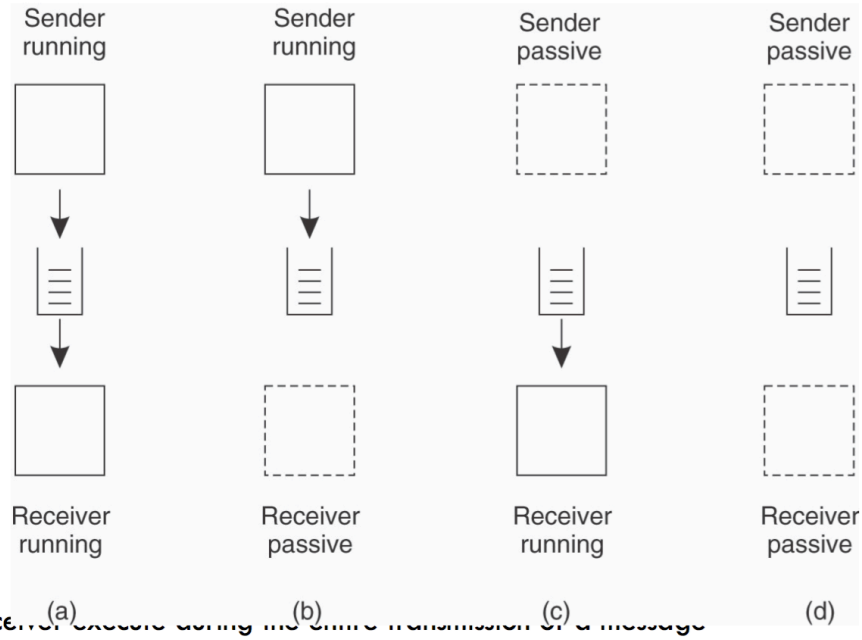
omplex

hote buffer
receipt starts
lock

≠RMI

ing

Scenarios



- (a) Both sender and receiver execute during the entire transmission of a message
- (b) Only the sender is executing while the receiver is passive
- (c) The sender is passive while the receiver is active
- (d) The system is storing message (and potentially sending to intermediate servers) while send

Interface

- The sender uses put to enqueue a message in a non-blocking
- The get primitive returns the longest pending message of the or may be tuned to return the highest priority message (in a queue)
- poll is the non-blocking variant of the get
- Callbacks can be used to automatically fetch new messages receiver side via notify

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first mess
Poll	Check a specified queue for messages, and remove the first. Never bl
Notify	Install a handler to be called when a message is put into the specified

er and receiver are passive

way

queue

priority

from the

sage
ock
queue