

COMP2121: Principles of Distributed Systems and Networks

Communication 1 / 2

Unit coordinator Dr. Vincent Gramoli
School of Information Technologies



THE UNIVERSITY OF
SYDNEY



Introduction

- Previous lecture:
 - Shared memory allows multiple programs to communicate
- Today's lecture:
 - What if we don't have shared memory?
 - How to implement an alternative communication medium, like message channel?

Outline

- Layered protocols
- Routing
- Socket
- Message passing interface (MPI)
- Message-oriented middleware (MOM)

Layered Protocols



THE UNIVERSITY OF
SYDNEY

Communication problem

Communicating is about transmitting encoded information

- **Problem:** two computers must agree on the code (language) they use
- **Open** standards give rules for everyone to communicate with each other
 - International Standards Organization (ISO)

The Open Systems Interconnection (**OSI**) reference model names communication levels, and assigns roles to each level

- Internet Engineering Task Force (IETF)

The **Request For Comments** (RFC) are a public description of internet communication protocols (e.g., TCP->RFC793, UDP->RFC768, SMTP->RFC2821, ICMP->RFC792)

- **Private** “closed” protocols exist
 - Skype: people did reverse engineering to discover the protocol, find security issues, or to implement IM clients



Two modes of communication

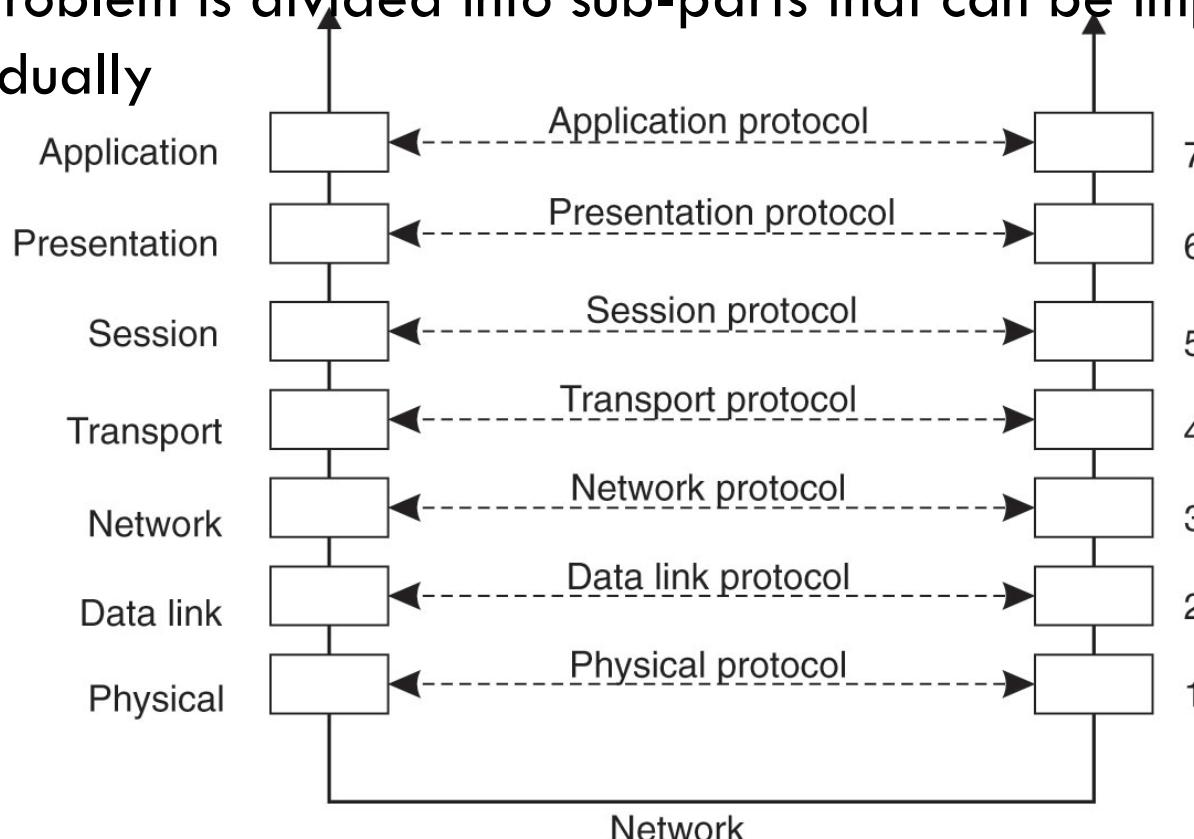
Connection oriented vs. Connectionless

- **Connection oriented**
 - Establish **explicitly a connection** with a partner before exchanging data
 - 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 layers

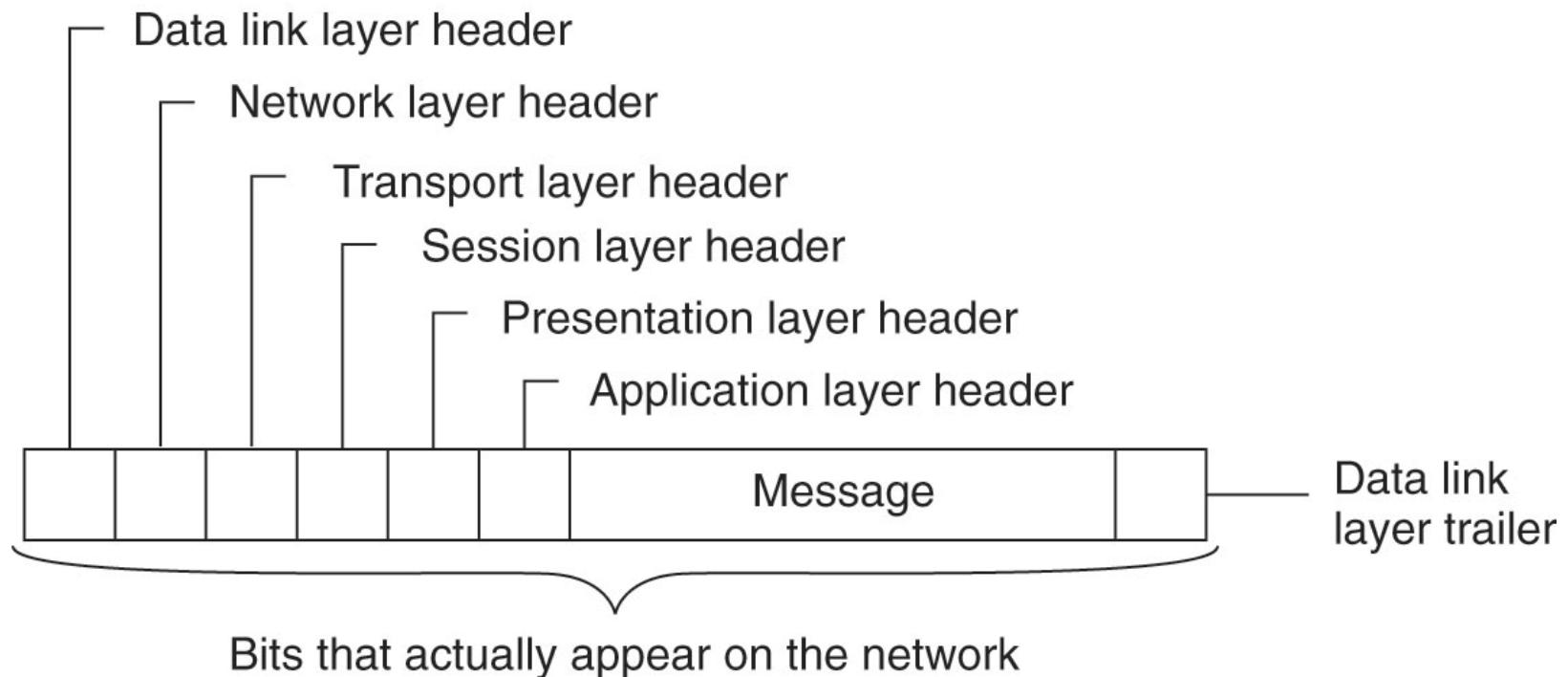
- Each layer deals with one specific aspect of the communication
- The problem is divided into sub-parts that can be implemented individually



Layers

Message structure

- As the message goes through lower layers before being sent, each layer adds its header to the message.
- Upon reception, the message is unmarshalled by the successive layers from bottom to top



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 at the beginning and end
 - adds a **checksum** (the result of some operation on the frame content)
 - If **receiver disagree about the checksum**, then it asks the sender to resend
 - Example: Ethernet for *Local Area Network (LAN)* or PPTP in *Virtual private network (VPN)*
- Network layer
 - Example: *Internet Protocol (IP)* for *Wide Area Network (WAN)*

Network communication

Transport and higher level layers

- Transport layer:
 - split the message from the application layer
 - number them
 - add the amounts 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 (TCP)
- Example of transport layer and network layer combination: TCP/IP

Network communication

High level layers

- OSI specifies three higher level layers:
 - Session layer: dialog control, transfer check-pointing
 - Presentation layer: says how information is represented
 - Application
- Session
 - Example:
 - Domain Name Service (DNS)
 - Lightweight Directory Protocol (LDAP)
 - Sometimes the session layer is omitted: Internet protocol suite.
- Application
 - Hypertext Transfer Protocol (HTTP)
 - File Transfer Protocol (FTP)
 - TCP/IP Terminal Emulation Protocol (Telnet)
 - X-Window

Network communication

Analogy: Mail service vs. Web service

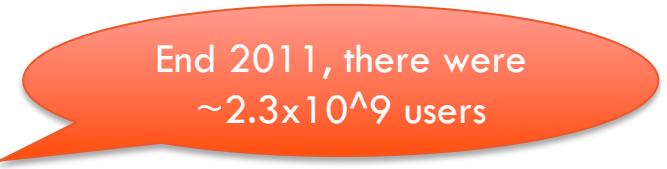
- Application (no clue of intermediary steps):
 - You post a letter with some address, the receiver reads it
 - **HTTP**: A user types a URL in a browser, a server sends back the web page
- Transport (error control):
 - Upon writing a wrong address on a letter, the letter will be sent back to you
 - **TCP** initialized a connection, checks for potential errors and may retransmit
- Internet (recipient is unknown):
 - An **airplane** moves letters between cities without knowing the recipients
 - **IP** brings packets over the WAN potentially from one LAN to another
- Data link:
 - **Trucks** move letters within a city
 - **Ethernet** handles transmission within the LAN
- Physical layer:
 - Use **pen** to write and **glasses** to read letters
 - Specifying fiber, wire, radio to transmit the one and zero encoding the message



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) instead of IPv4's 0800.
- Scalability in the number of addresses
IPv6 increases the IP address size from 32 bits (**4,294,967,296** in total) to 128 bits, to support more levels of addressing hierarchy, a much greater number of addressable nodes and 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 to any one of a group of nodes
- Efficient forwarding
Changes in the way IP header options are encoded: Hop-by-Hop Option, Routing (Type 0), Fragment, Destination Option, Authentication, Encapsulation Payload
- Greater flexibility for additional options



End 2011, there were
 $\sim 2.3 \times 10^9$ users

Routing



THE UNIVERSITY OF
SYDNEY

Routing Problem

What are routing protocols used for?

- Related to **path finding problem** in graphs
- Graph nodes represent internet routers, edges are communication links
- Dijkstra thought about **the shortest path algorithm** in Amsterdam when trying to find his way [Frana&Misa, CACM'10]
- Two main classes of protocols/algorithms:
 1. Distance-vector: neighbours exchange their routing table
 2. Link-state: neighbours only exchange connectivity information
- **Routing is necessary** in all networks except *Local Area Networks (LANs)* where Ethernet provides direct communication between all pairs of attached hosts
- The routing protocol is implemented in the **network layer** of each router to determine the route for the transmission of packets to their destination
- In large networks, **adaptive routing** is employed: the best route for communication between two points in the network is re-evaluated periodically, taking into account the current traffic in the network

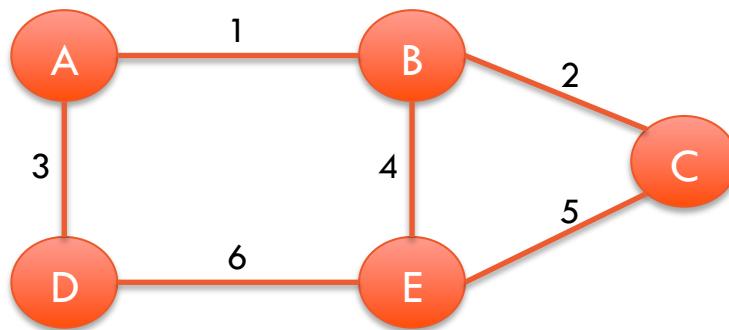


GPS clients use shortest path algorithms

Distance-vector routing algorithm

Router Information Protocol (RIP)

- Mainly used in internet up to 1979
- Relies on Bellman-Ford algorithm: Bellman algorithm [1957] distributed 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

- General idea: maintain a table representing the direction for each destination
 1. Every period t ($=30\text{sec}$) and whenever the local routing table T_L changes, send T_L to all accessible neighbours
 2. Upon reception of a table T_R with a new destination or a lower-cost route to an existing destination, update the local table T_L

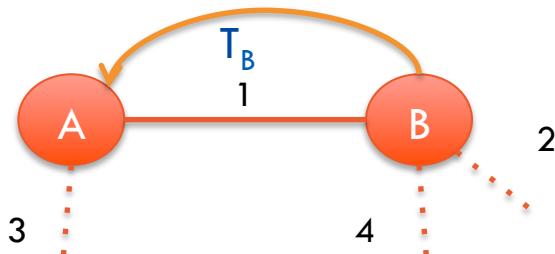
If a node detects a link failure, it sets ∞ as the associated cost of such a link and sends its local table to neighbours

Eventually (when failures stop) each router gets for each destination the direction leading to the minimal cost

Distance-vector routing algorithm

RIP (con't)

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

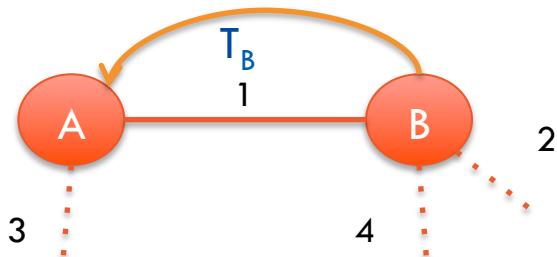
Each router executes the same code:

1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n :
for all rows R_R in T_R {
 if ($R_R.dir \neq n$) {
 $R_R.cost = R_R.cost + 1$
 $R_R.dir = n$
 // add new destination to T_L
 if ($R_R.dest$ is not in T_L) add R_R to T_L
 else for all rows R_L in T_L {
 if ($R_R.dest = R_L.dest$ and
 ($R_R.cost < R_L.cost$ or $R_L.dir = n$))
 $R_L = R_R$
 // $R_R.cost < R_L.cost$: remote node's better route
 // $R_L.dir = n$: remote node is more authoritative
 }
 }
 }

Distance-vector routing algorithm

RIP (con't)

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

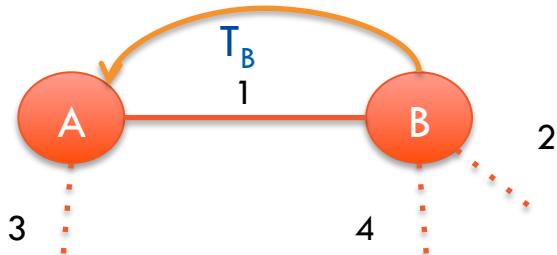
Each router executes the same code:

1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n:
for all rows R_R in T_R {
 if ($R_R.dir \neq n$) {
 $R_R.cost = R_R.cost + 1$
 $R_R.dir = n$
 // add new destination to T_L
 if ($R_R.dest$ is not in T_L) add R_R to T_L
 else for all rows R_L in T_L {
 if ($R_R.dest = R_L.dest$ and
 ($R_R.cost < R_L.cost$ or $R_L.dir = n$))
 $R_L = R_R$
 // $R_R.cost < R_L.cost$: remote node's better route
 // $R_L.dir = n$: remote node is more authoritative
 }
 }
 }

Distance-vector routing algorithm

RIP (con't)

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-1	0-1
C	-2-1	+2
D	+1	2-3
E	-4-1	+2

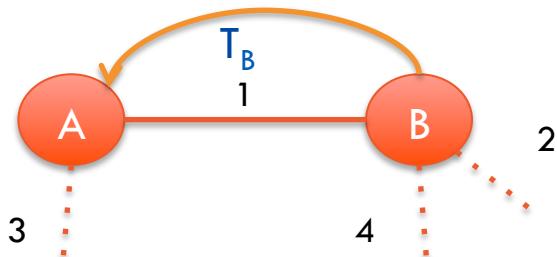
Each router executes the same code:

1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n:
for all rows R_R in T_R {
if ($R_R.dir \neq n$) {
 $R_R.cost = R_R.cost + 1$
 $R_R.dir = n$
 // add new destination to T_L
 if ($R_R.dest$ is not in T_L) add R_R to T_L
 else for all rows R_L in T_L {
 if ($R_R.dest = R_L.dest$ and
 ($R_R.cost < R_L.cost$ or $R_L.dir = n$))
 $R_L = R_R$
 // $R_R.cost < R_L.cost$: remote node's better route
 // $R_L.dir = n$: remote node is more authoritative
 }
 }
}

Distance-vector routing algorithm

RIP (con't)

Example: A receives table T_B from B



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

Received routing table T_R		
Dest.	Dir	Cost
A	1	1
B	local-1	0-1
C	-2-1	+2
D	+1	2-3
E	-4-1	+2

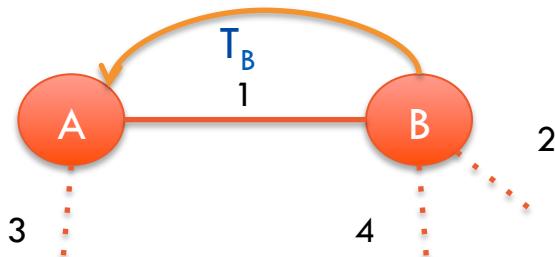
Each router executes the same code:

1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n:
for all rows R_R in T_R {
 if ($R_R.dir \neq n$) {
 $R_R.cost = R_R.cost + 1$
 $R_R.dir = n$
 // add new destination to T_L
 if ($R_R.dest$ is not in T_L) add R_R to T_L
 else for all rows R_L in T_L {
 if ($R_R.dest = R_L.dest$ and
 ($R_R.cost < R_L.cost$ or $R_L.dir = n$))
 $R_L = R_R$
 // $R_R.cost < R_L.cost$: remote node's better route
 // $R_L.dir = n$: remote node is more authoritative
 }
 }
 }
}

Distance-vector routing algorithm

RIP (con't)

Example: A receives table T_B from B



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

Received routing table T_R		
Dest.	Dir	Cost
A	1	1
B	local-1	0-1
C	-2 1	+ 2
D	+ 1	2 3
E	-4 1	+ 2

Each router executes the same code:

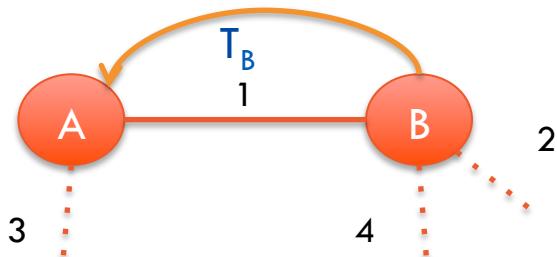
1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n:

```
for all rows  $R_R$  in  $T_R$  {
    if ( $R_R$ .dir ≠ n) {
         $R_R$ .cost =  $R_R$ .cost + 1
         $R_R$ .dir = n
        // add new destination to  $T_L$ 
        if ( $R_R$ .dest is not in  $T_L$ ) add  $R_R$  to  $T_L$ 
    } else for all rows  $R_L$  in  $T_L$  {
        if ( $R_R$ .dest =  $R_L$ .dest and
            ( $R_R$ .cost <  $R_L$ .cost or  $R_L$ .dir = n))
             $R_L$  =  $R_R$ 
        //  $R_R$ .cost <  $R_L$ .cost: remote node's better route
        //  $R_L$ .dir = n: remote node is more authoritative
    }
}
```

Distance-vector routing algorithm

RIP (con't)

Example: A receives table T_B from B



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

Received routing table T_R		
Dest.	Dir	Cost
A	1	1
B	local-1	0-1
C	-2 1	+ 2
D	+ 1	2 3
E	-4 1	+ 2

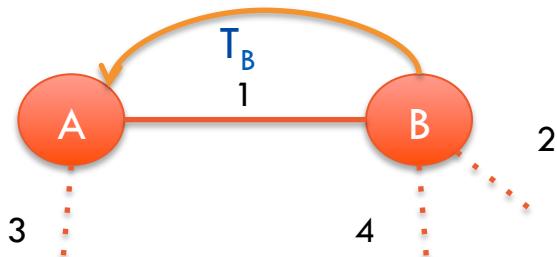
Each router executes the same code:

1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n:
for all rows R_R in T_R {
 if ($R_R.dir \neq n$) {
 $R_R.cost = R_R.cost + 1$
 $R_R.dir = n$
 // add new destination to T_L
 if ($R_R.dest$ is not in T_L) add R_R to T_L
 else for all rows R_L in T_L {
 if ($R_R.dest = R_L.dest$ and
 ($R_R.cost < R_L.cost$ or $R_L.dir = n$))
 $R_L = R_R$
 // $R_R.cost < R_L.cost$: remote node's better route
 // $R_L.dir = n$: remote node is more authoritative
 }
 }
 }

Distance-vector routing algorithm

RIP (con't)

Example: A receives table T_B from B



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

Each router executes the same code:

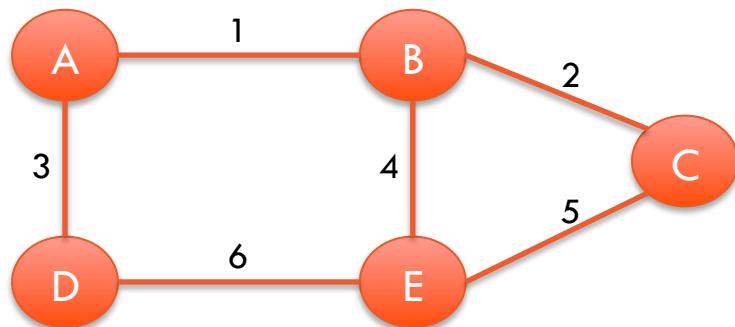
1. Send: Each t seconds or when T_L changes, send T_L on each non-faulty outgoing link
2. Receive: Whenever a routing table T_R is received on link n :

```
for all rows RR in TR {
    if (RR.dir ≠ n) {
        RR.cost = RR.cost + 1
        RR.dir = n
        // add new destination to TL
        if (RR.dest is not in TL) add RR to TL
        else for all rows RL in TL {
            if (RR.dest = RL.dest and
                (RR.cost < RL.cost or RL.dir = n))
                RL = RR
            // RR.cost < RL.cost: remote node's better route
            // RL.dir = n: remote node is more authoritative
        }
    }
}
```

Distance-vector routing algorithm

RIP (con't)

Let's re-start the algorithm from scratch: all routing tables are empty



Routing table of router A		
Dest.	Dir	Cost
A	local	0

Routing table of router B		
Dest.	Dir	Cost
B	local	0

Routing table of router C		
Dest.	Dir	Cost
C	local	0

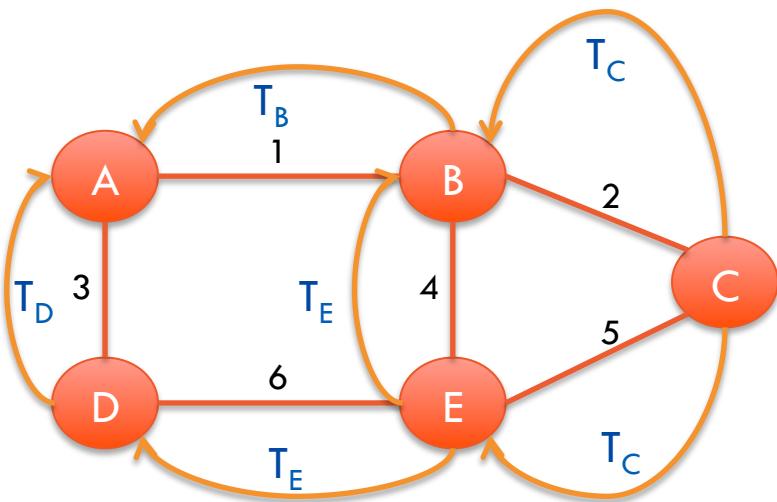
Routing table of router D		
Dest.	Dir	Cost
D	local	0

Routing table of router E		
Dest.	Dir	Cost
E	local	0

Distance-vector routing algorithm

RIP (con't)

- Example: How is routing table A built from scratch?

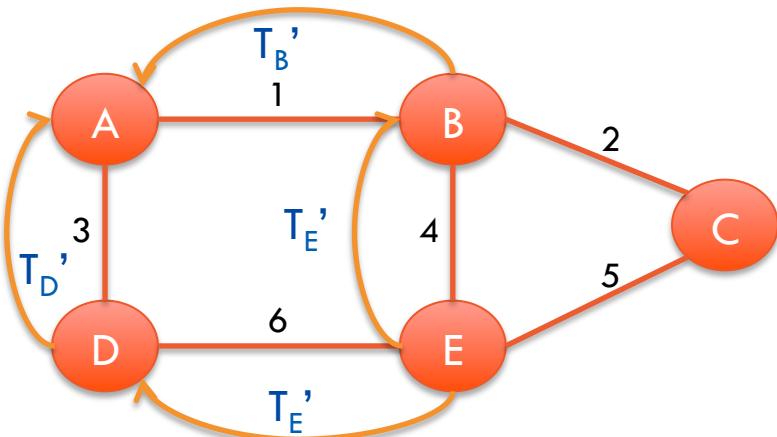


Routing table of router A		
Dest.	Dir	Cost
A	local	0
B	1	1
D	3	1

Distance-vector routing algorithm

RIP (con't)

- Example: How is routing table A built from scratch?

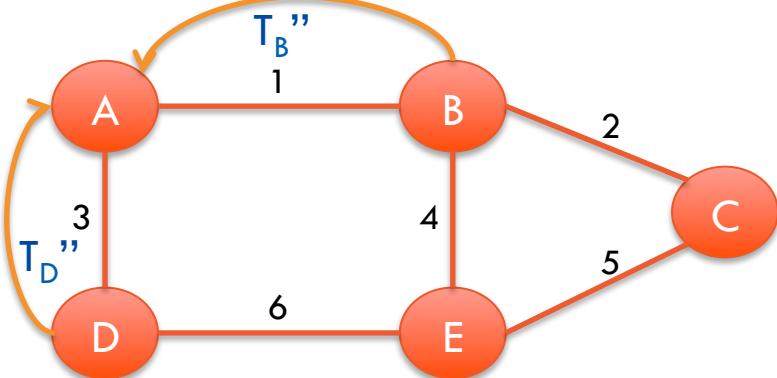


Routing table of router A		
Dest.	Dir	Cost
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Distance-vector routing algorithm

RIP (con't)

- Example: How is routing table A built from scratch?



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

Distance-vector routing algorithm

RIP (con't)

- Advantages:
 - Simple
 - Efficient in small networks
- Limitations:
 - Costs based on the number of hops is unrealistic (bandwidth of links counts)
 - No big deal: this is just a matter of defining link weights (support negative values)
 - Inefficient in large networks as loops may occur before the convergence state is reached

To address this inefficiency more scalable routing protocols, called linked state, were proposed

Link-state routing protocol

Dijkstra's Algorithm: find the shortest path from a node i to other nodes

- Edsger Dijkstra 1956

- Assume that each link (from i to j) has an associated non-negative cost (C_{ij})
- N , set of nodes
- s , source node that starts the algo
- $V = \{s\}$, set of visited nodes
- Find the shortest distance D_j from s to every node j

Init the routing table for node s :

```
for all nodes in N:  
    if  $j$  neighbor of  $s$ :  $D_j = C_{sj}$   
    else  $D_j = \infty$   
 $D_s = 0$ 
```

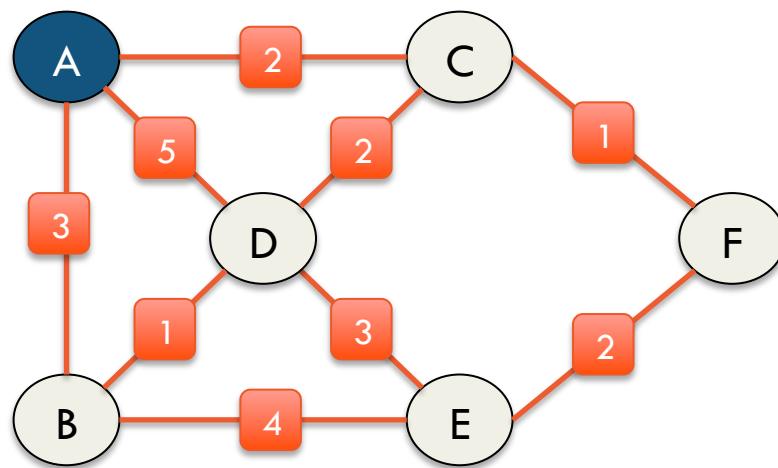
Algo for routing table of node s :

```
while ( $V \neq N$ ) { // while some nodes are unvisited  
    select node  $i$  such that  
         $D_i = \min\{ D_j : j \notin V\}$  //  $i$ , closer unvisited node  
    for all nodes  $j \notin V$ : // for all unvisited node  $j$   
        if ( $D_j > D_i + C_{ij}$ ) // if cheaper through  $i$   
             $D_j = D_i + C_{ij}$  // update the distance to  $j$   
             $L_j = i \rightarrow j$  // take link  $i$  to route to  $j$   
     $V = V \cup \{i\}$   
}
```

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



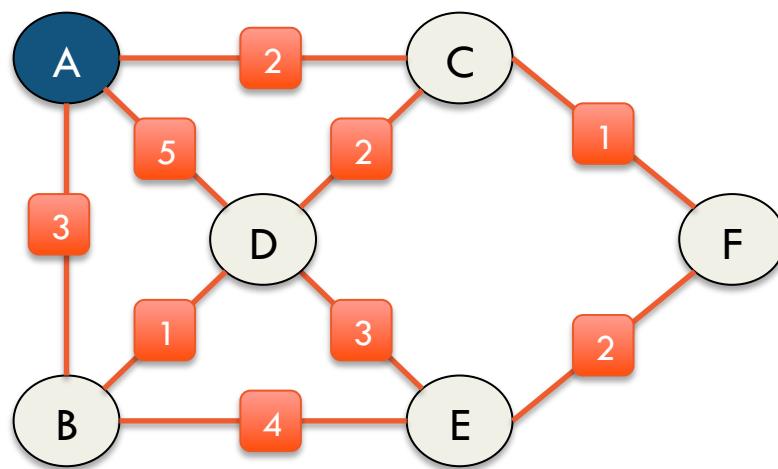
i = C

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	A→D	5
E	?	∞
F	?	∞

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



$i = C$

Routing table of router A		
Dest.	L	D
A	\emptyset	0
B	$A \rightarrow B$	3
C	$A \rightarrow C$	2
D	$A \rightarrow D$	5
E	?	∞
F	?	∞

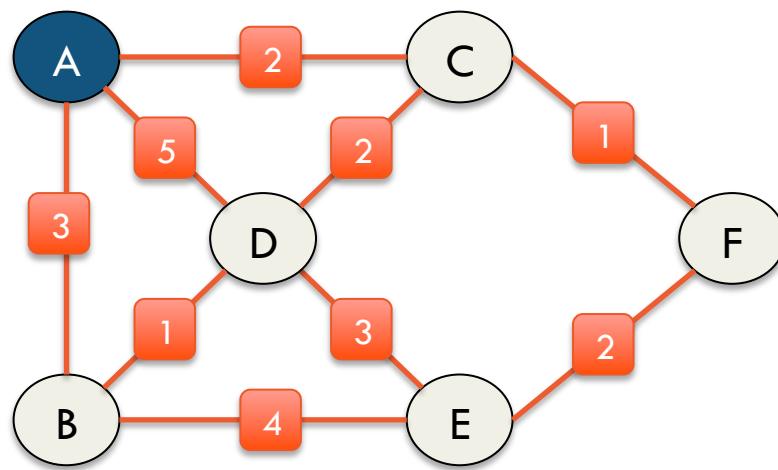
$D_D > D_C + C_{CD}$

$D_F > D_C + C_{CF}$

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



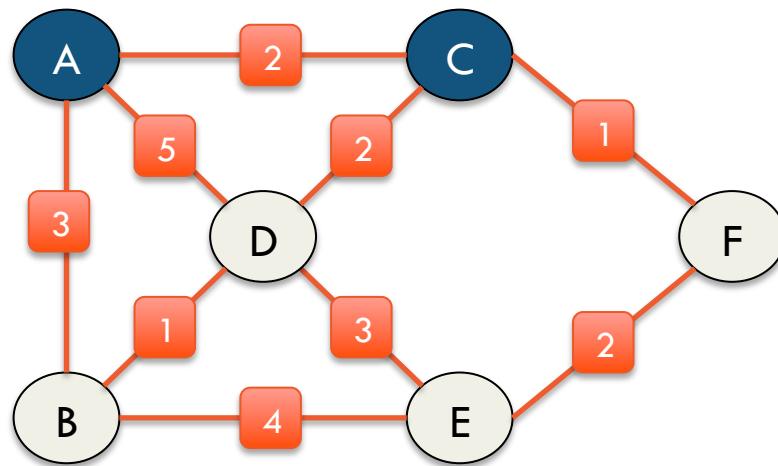
i = C

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	?	∞
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



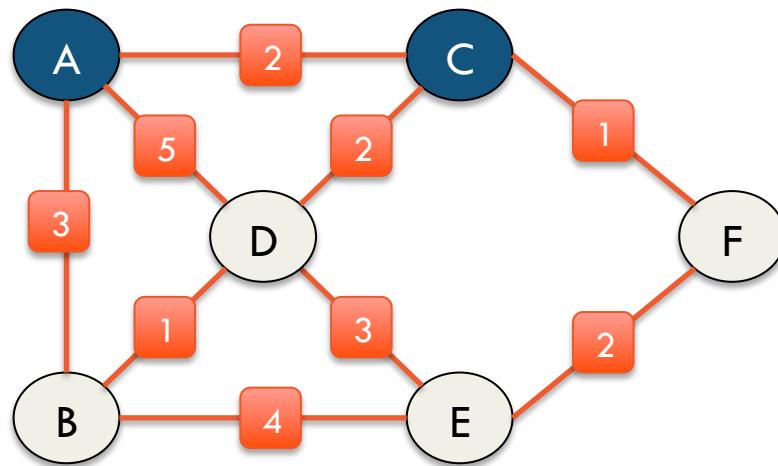
i = B

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	?	∞
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



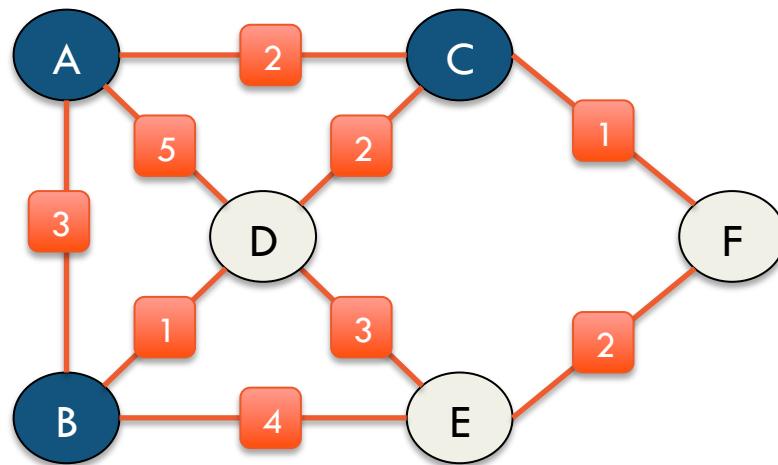
i = B

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	B→E	7
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



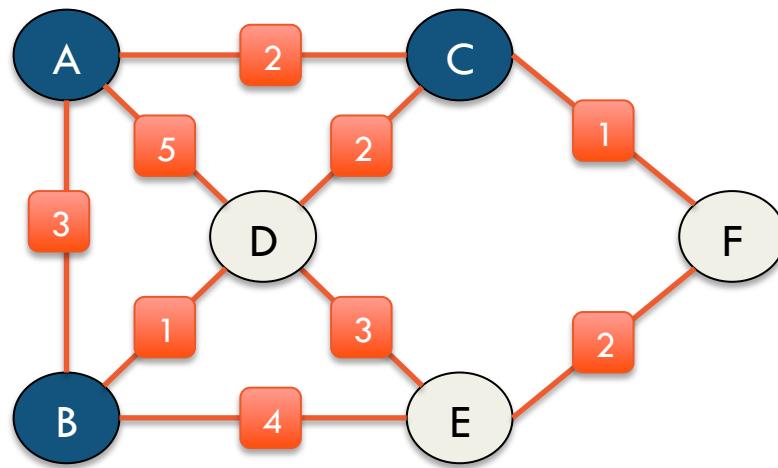
i = F

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	B→E	7
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



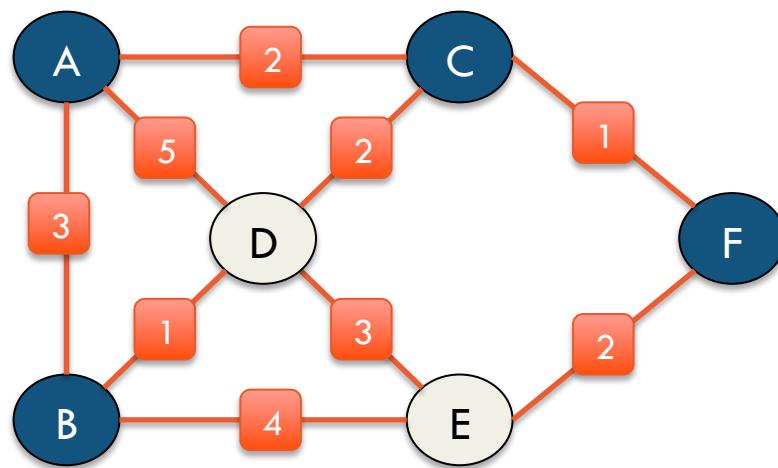
i = F

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	F→E	5
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



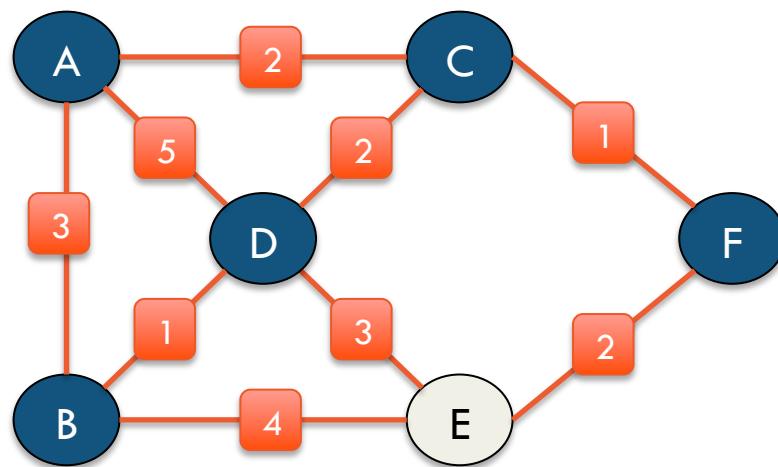
i = D

Routing table of router A		
Dest.	L	D
A	Ø	0
B	A→B	3
C	A→C	2
D	C→D	4
E	F→E	5
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



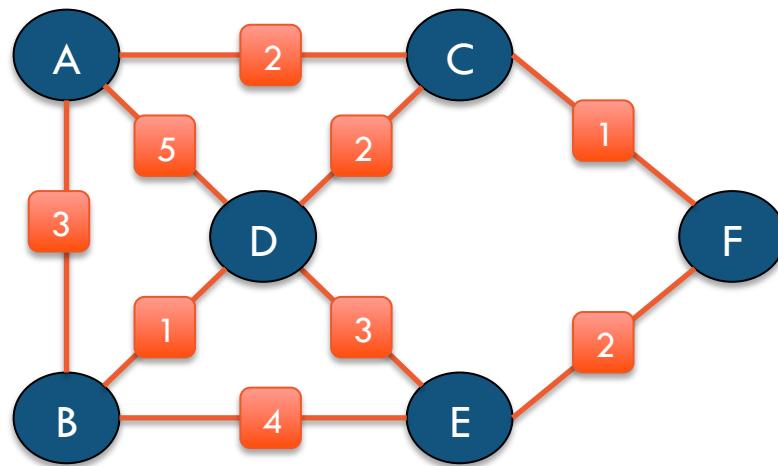
i = E

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	F→E	5
F	C→F	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



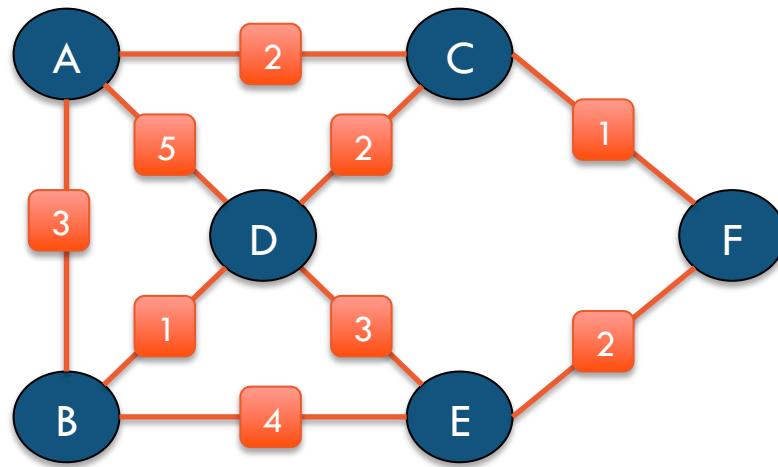
done

Routing table of router A		
Dest.	L	D
A	\emptyset	0
B	$A \rightarrow B$	3
C	$A \rightarrow C$	2
D	$C \rightarrow D$	4
E	$F \rightarrow E$	5
F	$C \rightarrow F$	3

Link-state routing protocol

Dijkstra's Algorithm (con't)

- Example: routing table A



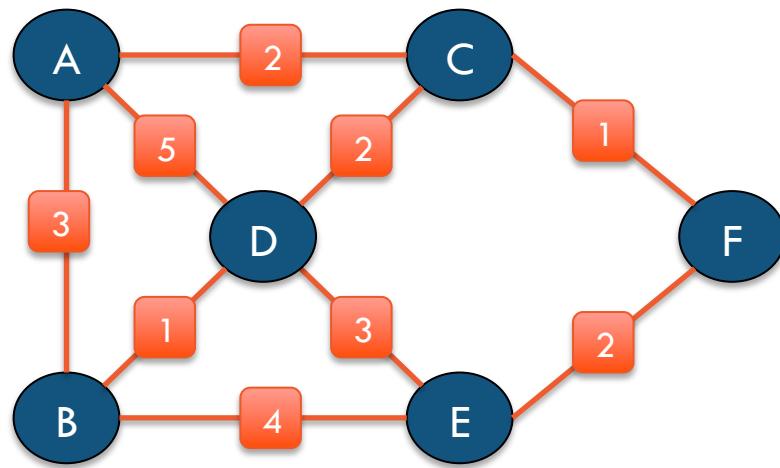
Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	F→E	5
F	C→F	3

- How to exploit such an inverted routing table?

Link-state routing protocol

How to route?

- Example: routing from A to E



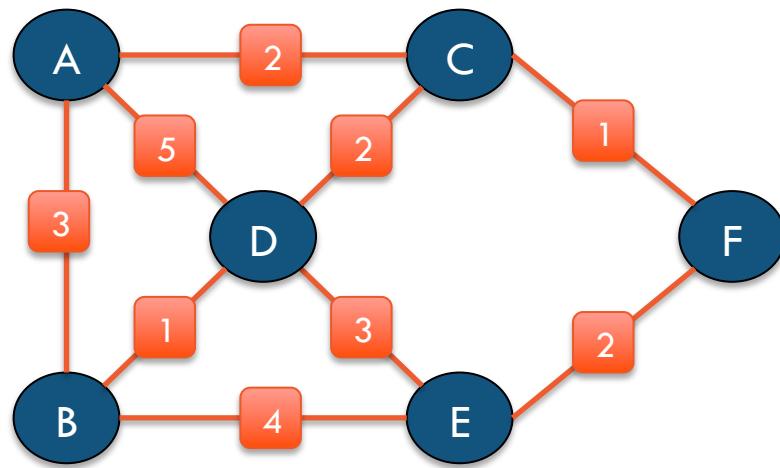
E

Routing table of router A		
Dest.	L	D
A	∅	0
B	A→B	3
C	A→C	2
D	C→D	4
E	F→E	5
F	C→F	3

Link-state routing protocol

How to route?

- Example: routing from A to E



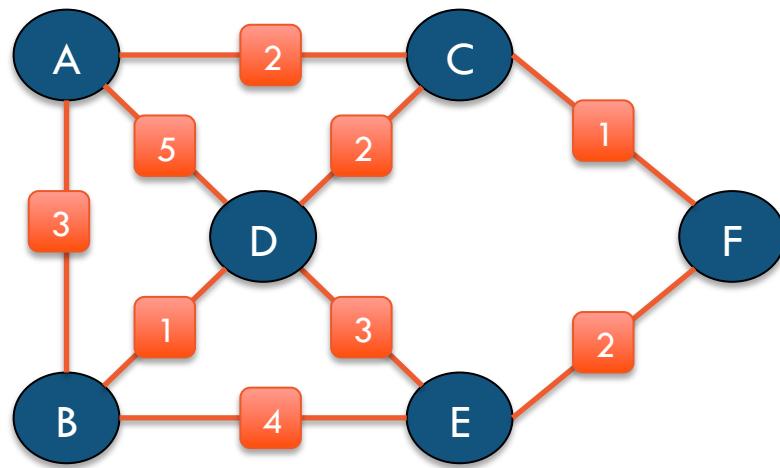
F → E

Routing table of router A		
Dest.	L	D
A	∅	0
B	A → B	3
C	A → C	2
D	C → D	4
E	F → E	5
F	C → F	3

Link-state routing protocol

How to route?

- Example: routing from A to E



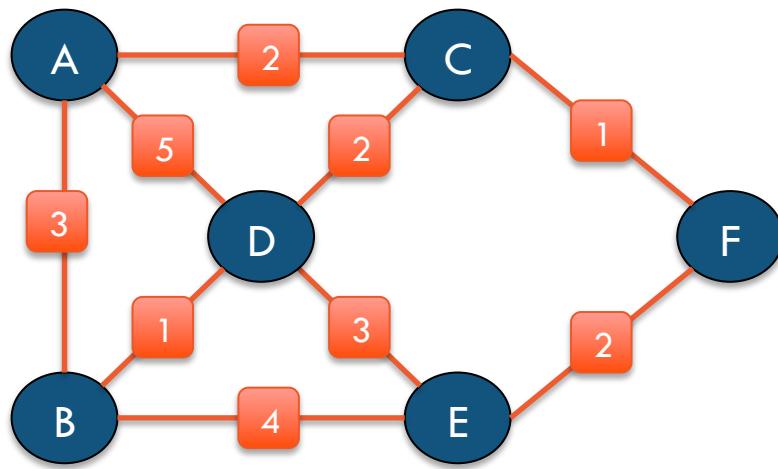
C → F → E

Routing table of router A		
Dest.	L	D
A	∅	0
B	A → B	3
C	A → C	2
D	C → D	4
E	F → E	5
F	C → F	3

Link-state routing protocol

How to route?

- Example: routing from A to E



A → C → F → E

Routing table of router A		
Dest.	L	D
A	∅	0
B	A → B	3
C	A → C	2
D	C → D	4
E	F → E	5
F	C → F	3

- A → C → F → E is the shortest path from A to E

Socket



THE UNIVERSITY OF
SYDNEY

Socket

Definition

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

Socket

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

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 implicitly to any 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 same properties as 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

Socket

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

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 implicitly to any 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 same properties as the original one; the client and server can communicate with send and receive.

Server



Synchronization point

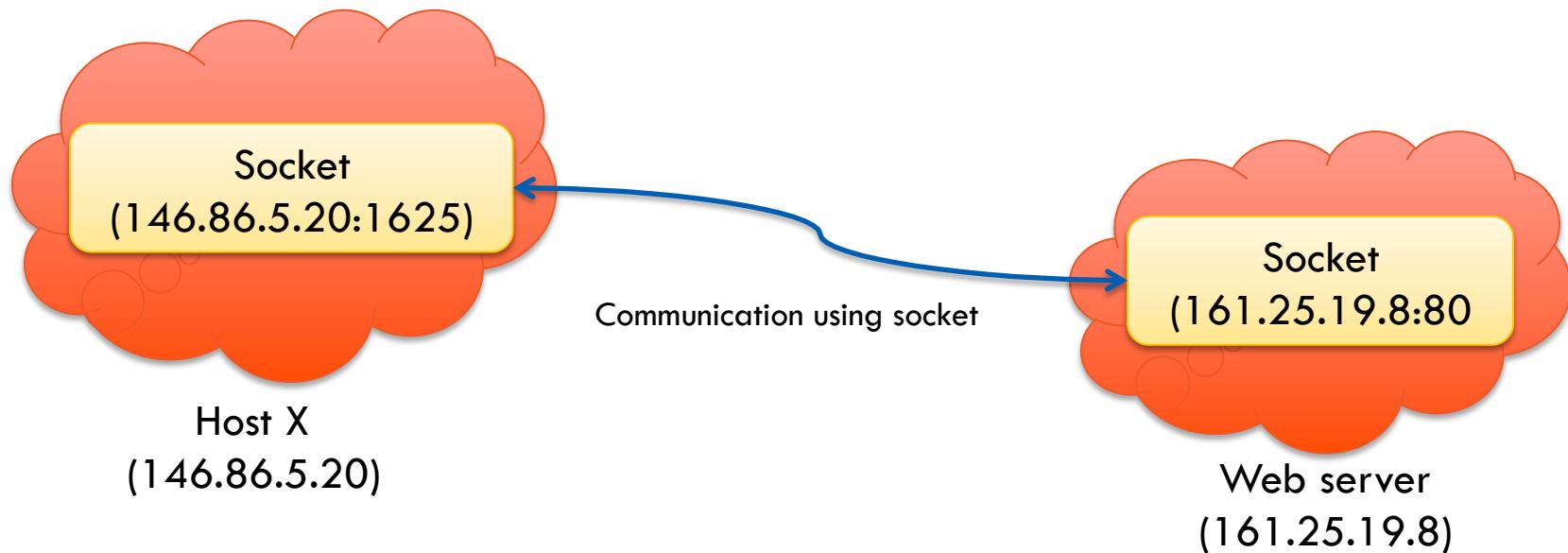


Client

Socket

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



Socket

Java Sockets

- Three socket classes:
 - DatagramSocket: Connectionless (UDP) socket, to open a lightweight communication favoring throughput rather than reliability
 - MulticastSocket: Multicast socket (subclass of DatagramSocket) to send data to multiple recipients
 - Socket: Connection-oriented (TCP) socket, to open a connection for reliable communication

Socket

Java TCP Sockets

```
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[] args) {  
        try {  
            // Specify server address and port for socket  
            Socket sock = new Socket("127.0.0.1", 6013);  
  
            ... // communicate with the server  
            sock.close(); // close socket  
        } catch (Exception e) {  
            System.err.println("server? or I/O problem");  
        }  
    }  
}
```

Message-Passing Interface (MPI)



THE UNIVERSITY OF
SYDNEY

MPI

MPI vs. Sockets

- Sockets
 - Limited set of primitives, providing only send and receive for communication
 - Exploits general purpose communication layered protocols (e.g., TCP/IP, UDP)
 - For example, 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

MPI

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 of communication
- **Blocking** recv (`MPI_recv`) and **non-blocking** one (`MPI_irecv`)
- Variety of communication primitives allows for **optimizations** but makes it **more complex**

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

MPI is used in high-performance parallel applications

Message-Oriented Middleware (MOM)



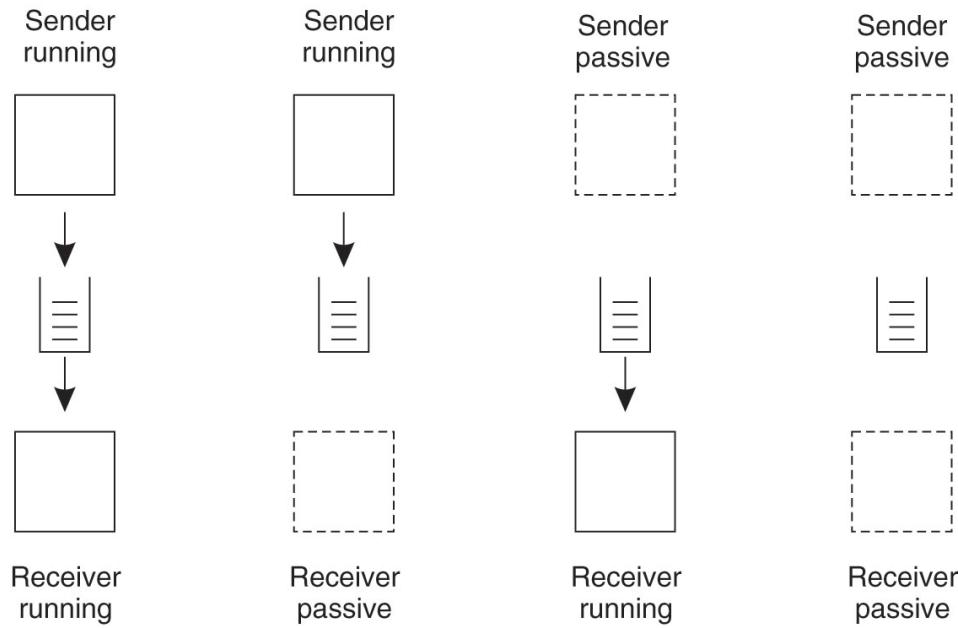
THE UNIVERSITY OF
SYDNEY

MOM

- Characteristics:
 - Do not require sender and receiver to be active during communication (\neq RMI 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 sending time)

MOM

Scenarios



- (a) Both sender and receiver are executing during transmission of 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 sender and receiver are passive

MOM

Interface

- The sender uses `put` to enqueue a message in a **non-blocking** way
- The `get` primitive returns the longest pending message of the queue or may be tuned to return the highest priority message (in a priority queue)
- `poll` is the **non-blocking** variant of the `get`
- Callbacks can be used to **automatically** fetch new messages from the 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 message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Conclusion

- Network protocols are divided into layers
- IPv6 is necessary for the Internet Protocol scalability
- Routing protocols are
 - simple but not scalable (distance-vector) or
 - more complex but scalable (link-state)
- There are various ways of communicating depending on the needs:
 - General-purpose communication (Socket)
 - High performance parallel communication (MPI)
 - Very long message transmission (MOM)