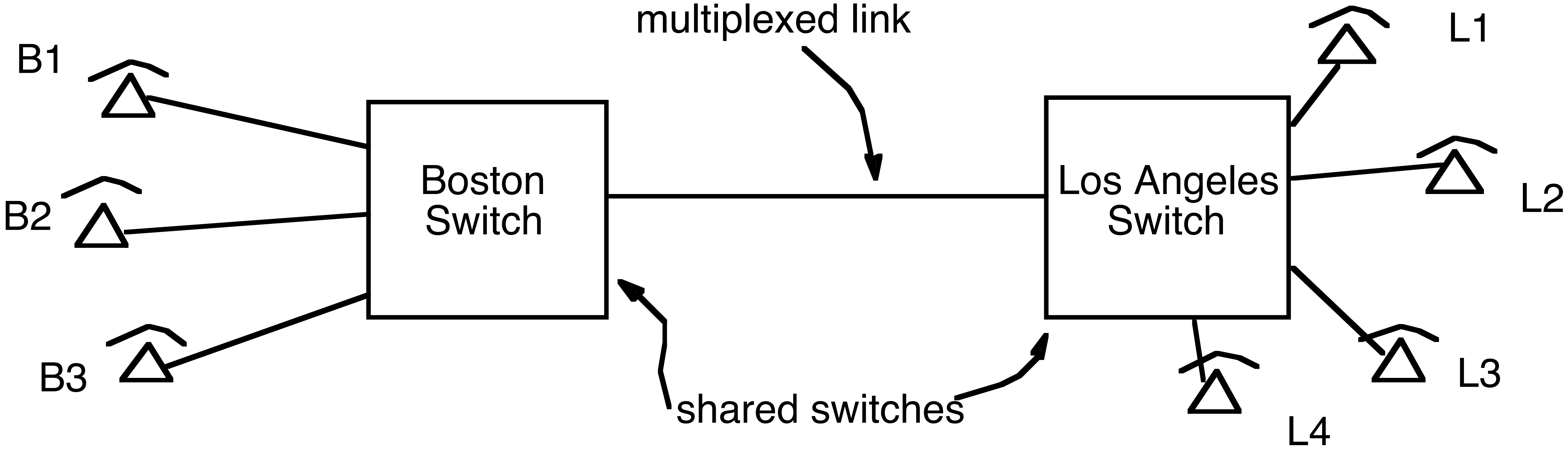
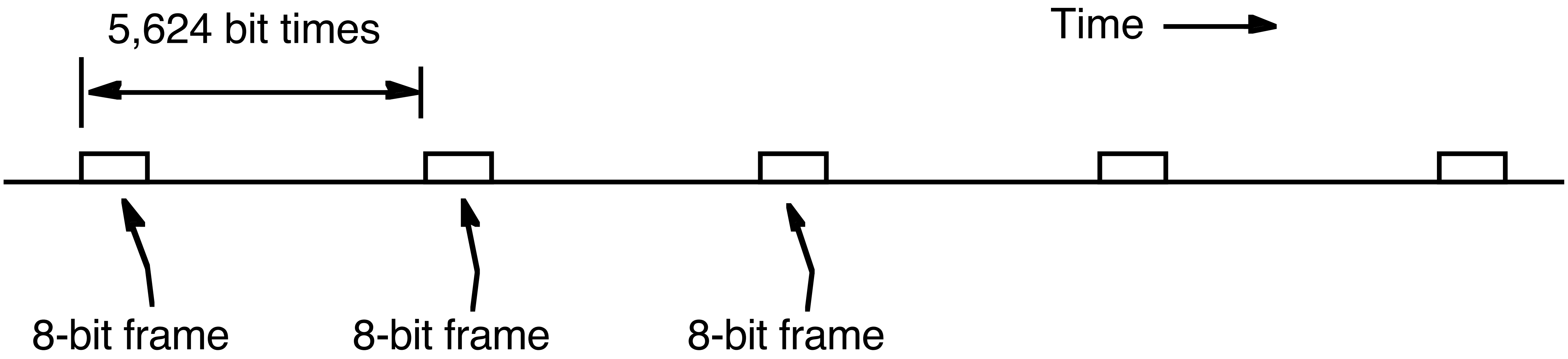
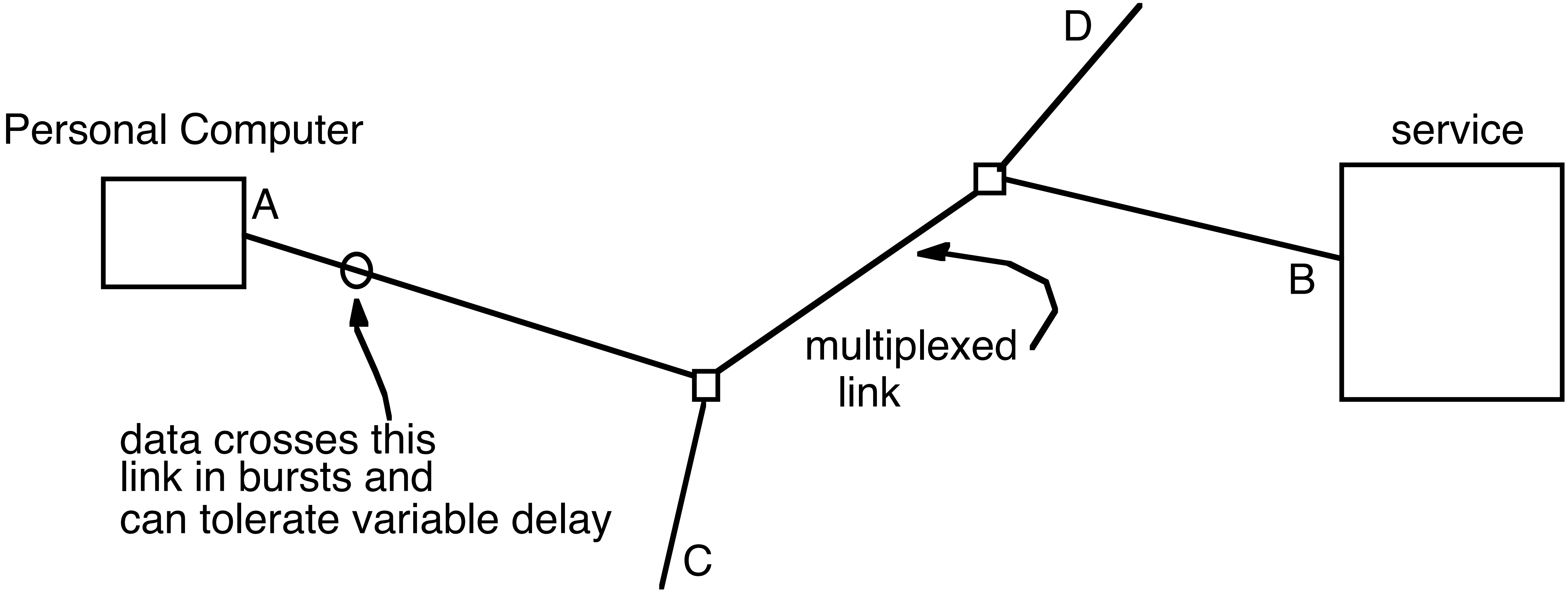


multiplexed link
B1
B2
B3
Boston Switch
Los Angeles Switch
L1
L2
L3
L4
shared switches


5,624 bit times
Time
8-bit frame
8-bit frame
8-bit frame



Personal Computer

A

data crosses this
link in bursts and
can tolerate variable delay

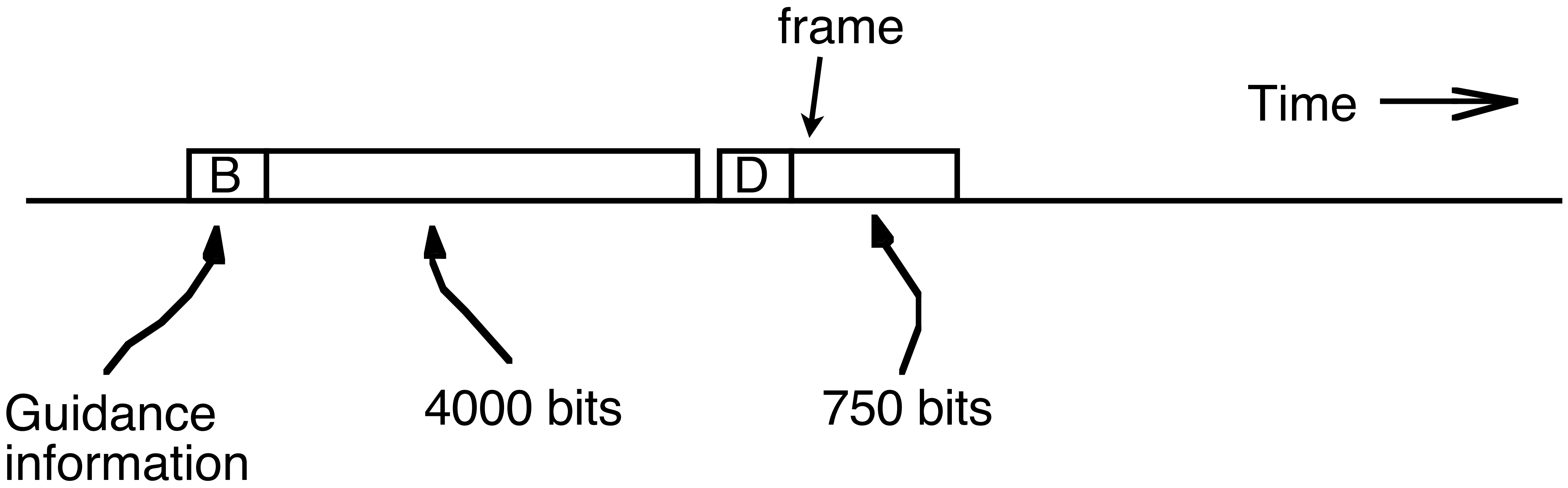
C

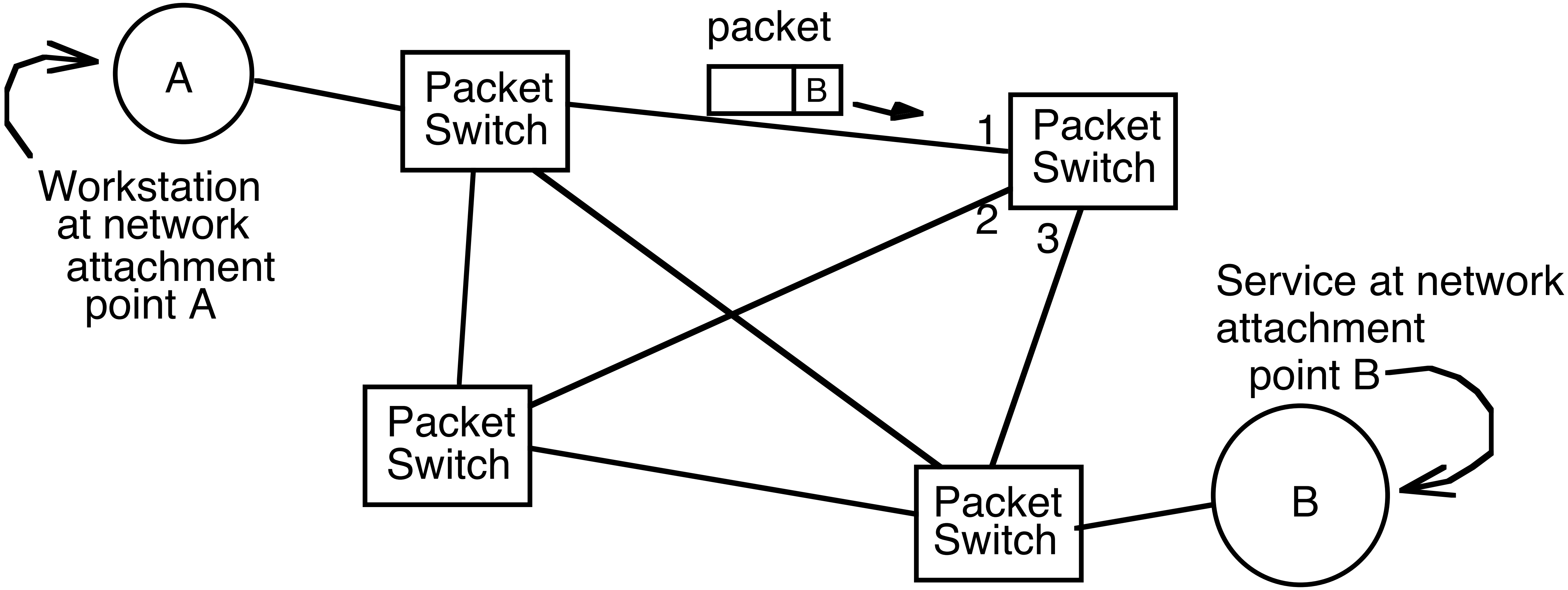
D

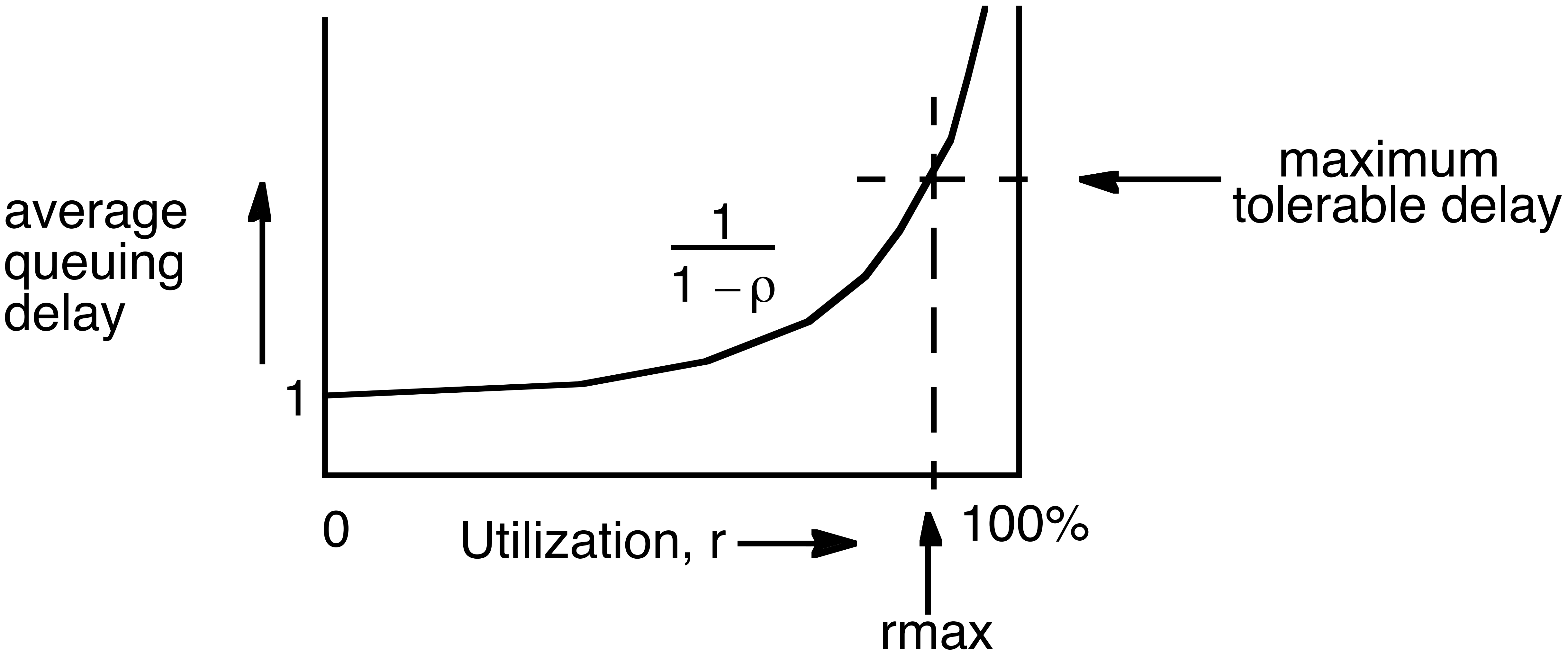
multiplexed
link

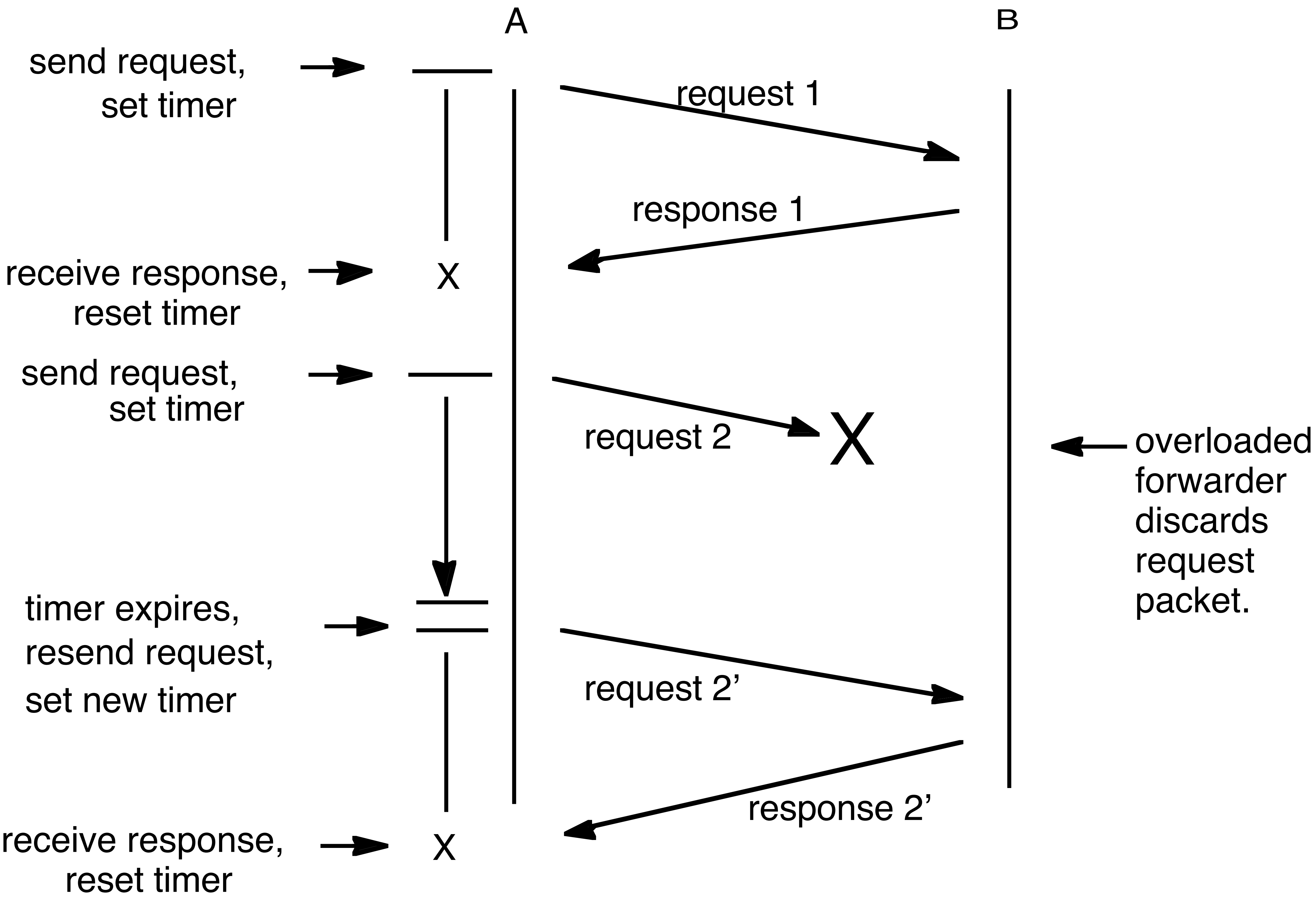
service

B


frame
Time
B
D
Guidance information
4000 bits
750 bits


packet
B
A
Packet Switch
Packet Switch
1
2
3
Packet Switch
Packet Switch
B
Workstation at network attachment point A
Service at network attachment point B


average
queuing
delay
1
$\frac{1}{1-\rho}$
maximum
tolerable delay
0
Utilization, r
100%
rmax



A

B

time

send request,
set timer

request 1

response 1

receive response,
reset timer

send request,
set timer

request 2

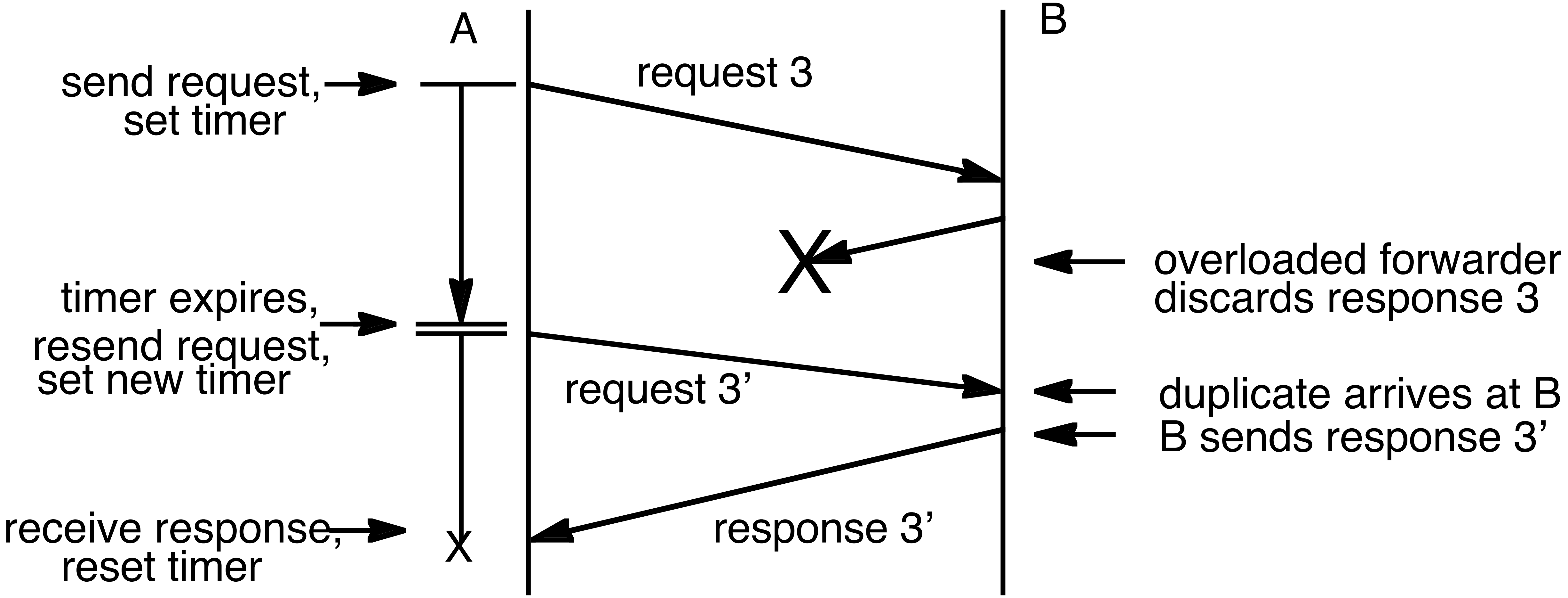
overloaded
forwarder
discards
request
packet.

timer expires,
resend request,
set new timer

request 2’

response 2’

receive response,
reset timer



A

B

send request, set timer

request 3

overloaded forwarder discards response 3

timer expires, resend request, set new timer

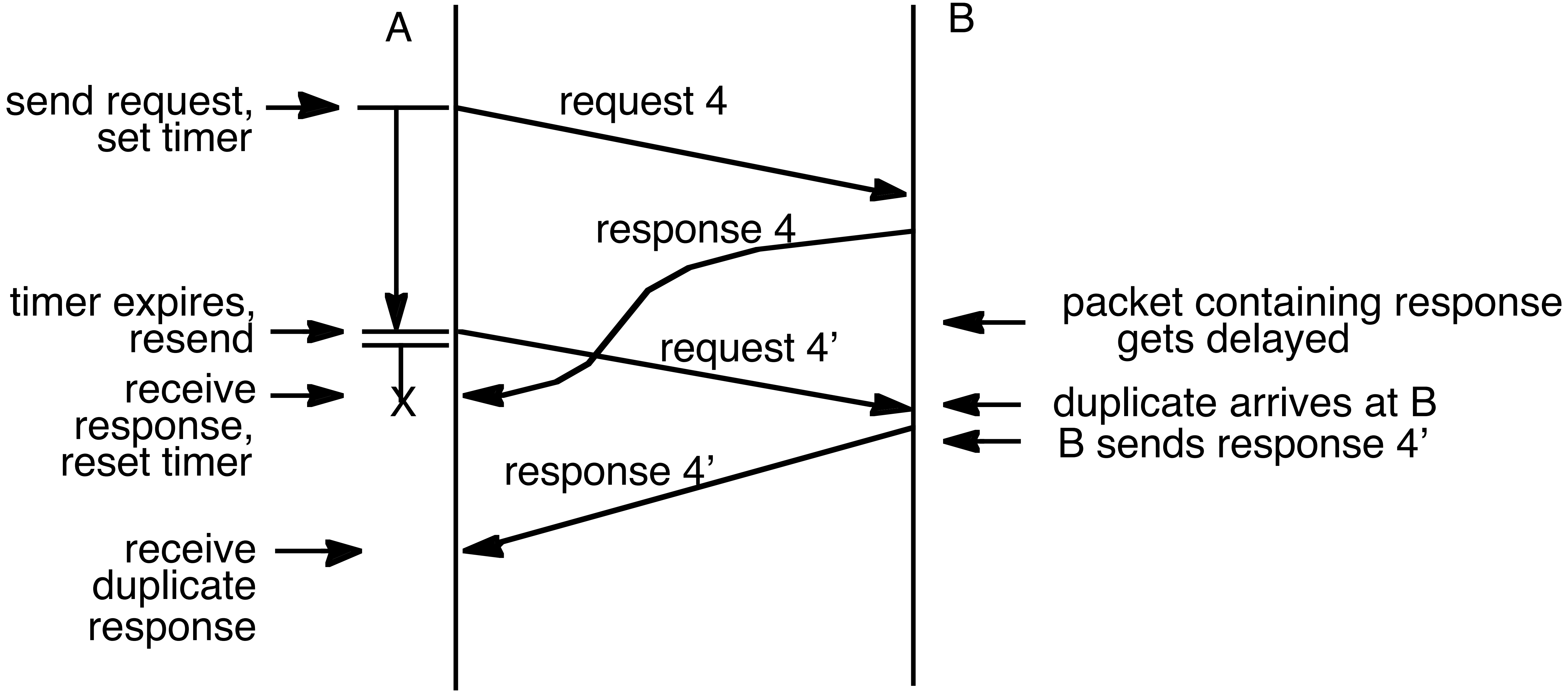
request 3’

duplicate arrives at B

B sends response 3’

receive response, reset timer

response 3’


A
B
send request, set timer
request 4
response 4
timer expires, resend
packet containing response gets delayed
request 4’
receive response, reset timer
X
duplicate arrives at B
B sends response 4’
response 4’
receive duplicate response

		Application characteristics		
		Continuous stream (e.g., interactive voice)	Bursts of data (most computer-to-computer data)	Response to load variations
Network Type	isochronous (e.g., telephone network)	good match	wastes capacity	(hard-edged) either accepts or blocks call
	asynchronous (e.g., Internet)	variable latency upsets application	good match	(gradual) 1 variable delay 2 discards data 3 rate adaptation

Networks encounter a vast range of

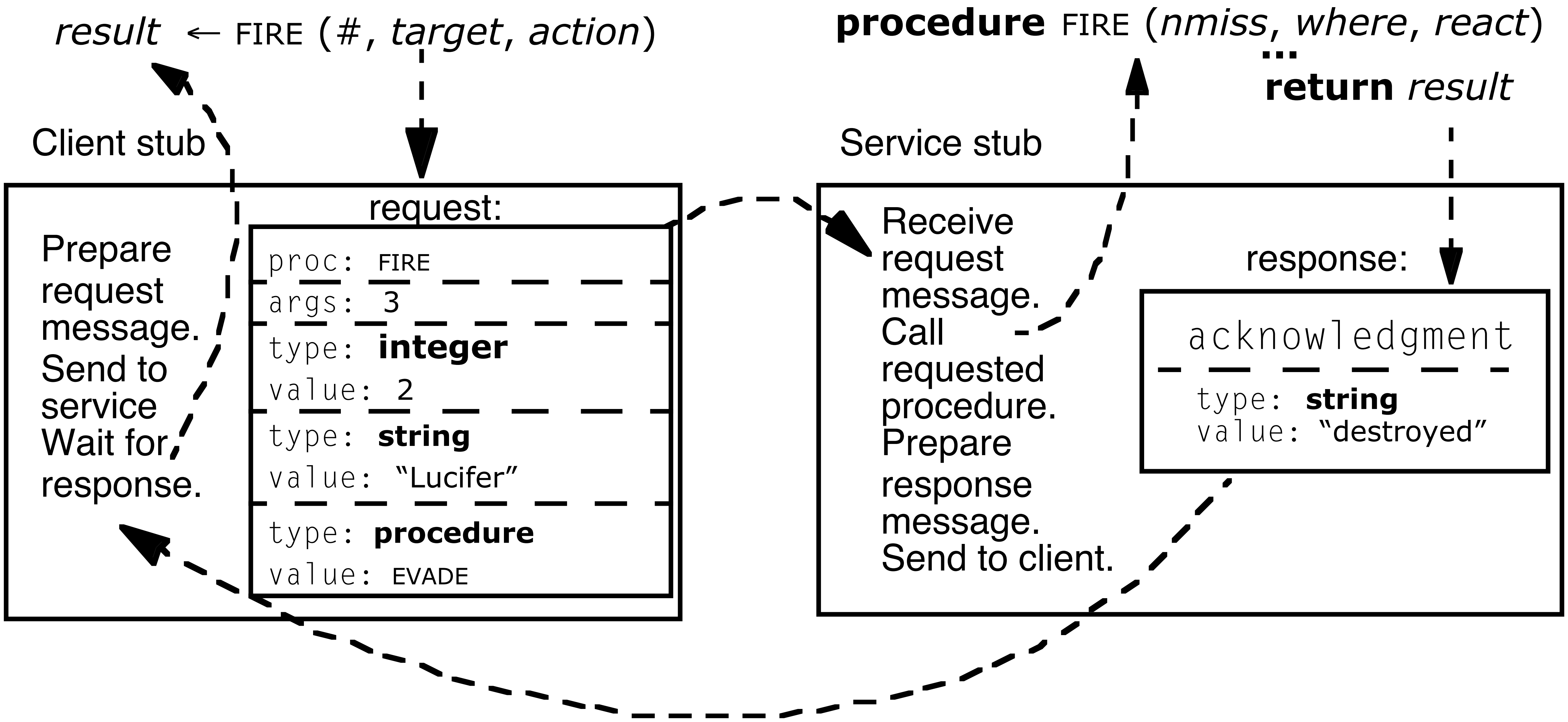
- Data rates
- Propagation, transmission, queuing, and processing delays.
- Loads
- Numbers of users

Networks traverse hostile environments

- Noise damages data
- Links stop working

Best-effort networks have

- Variable delays
- Variable transmission rates
- Discarded packets
- Duplicate packets
- Maximum packet length
- Reordered delivery



result ← FIRE (#, *target*, *action*)

procedure FIRE (*nmiss*, *where*, *react*)

...

return *result*

Client stub

request:

Prepare request message. Send to service Wait for response.

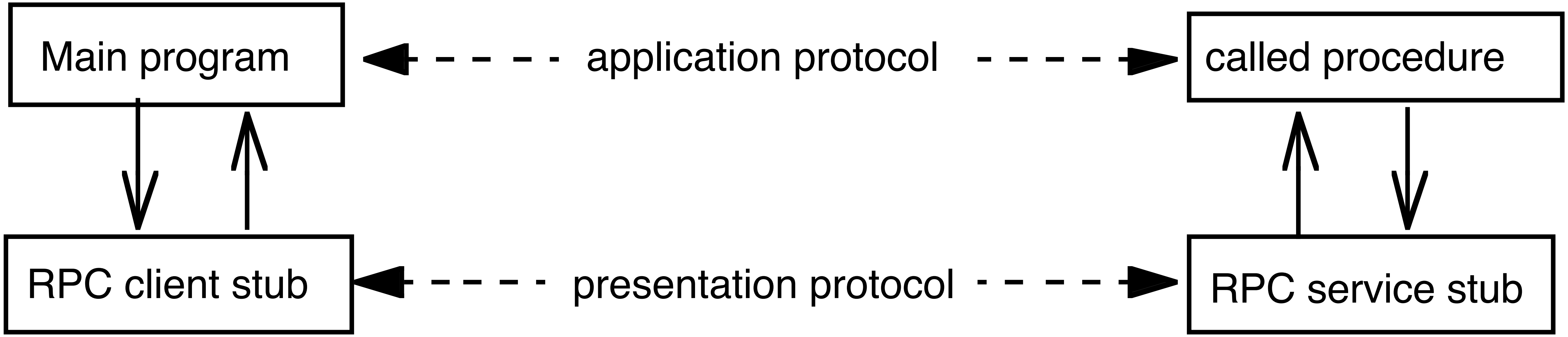
```
proc: FIRE
args: 3
type: integer
value: 2
type: string
value: "Lucifer"
type: procedure
value: EVADE
```

Service stub

Receive request message. Call requested procedure. Prepare response message. Send to client.

response:

```
acknowledgment
type: string
value: "destroyed"
```



Main program

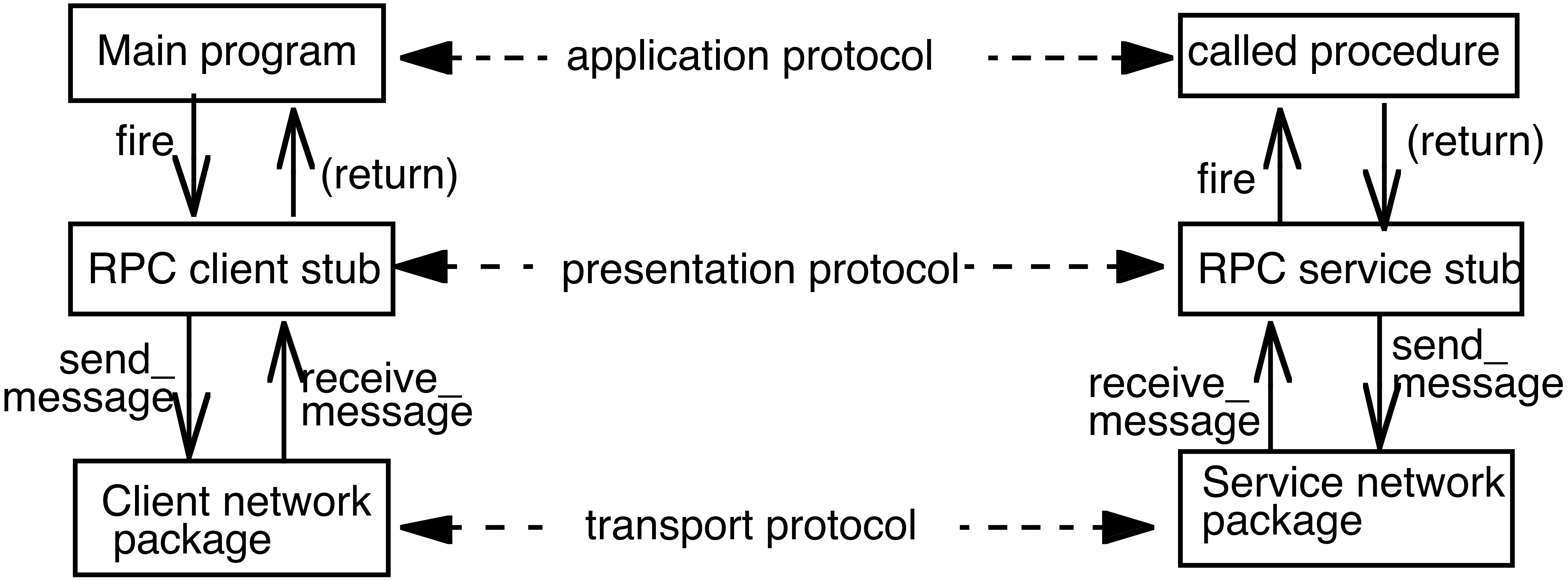
application protocol

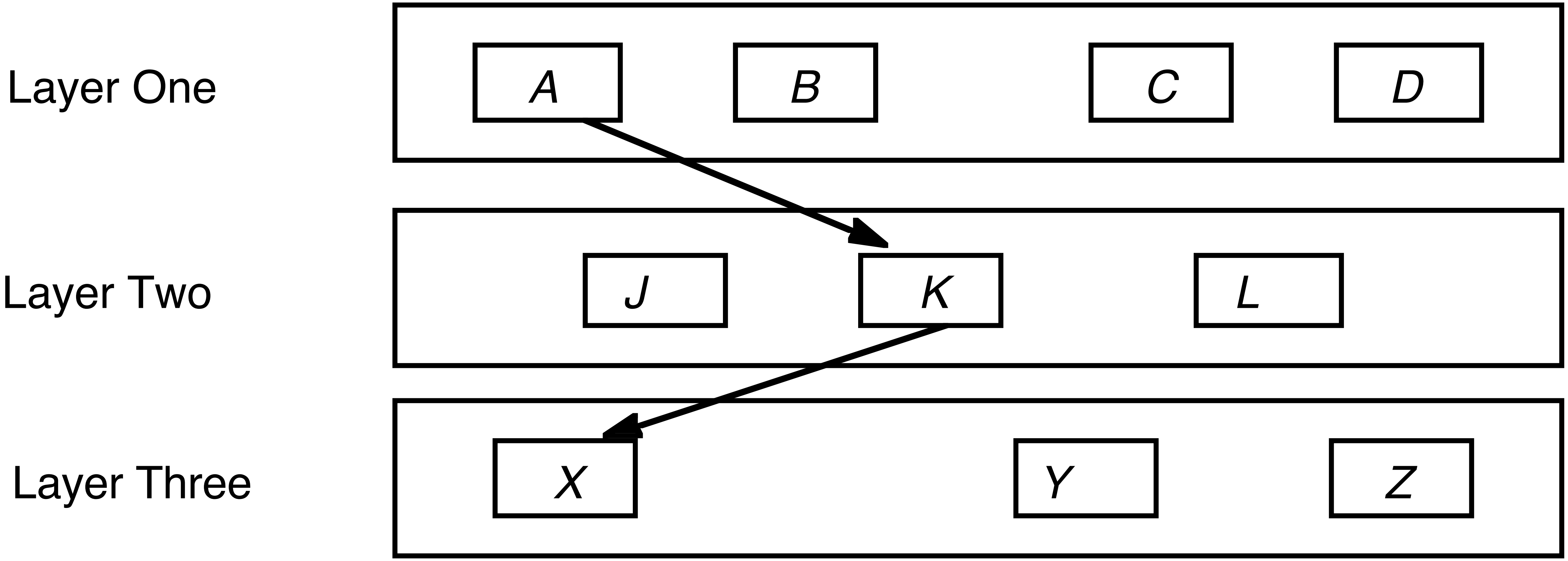
called procedure

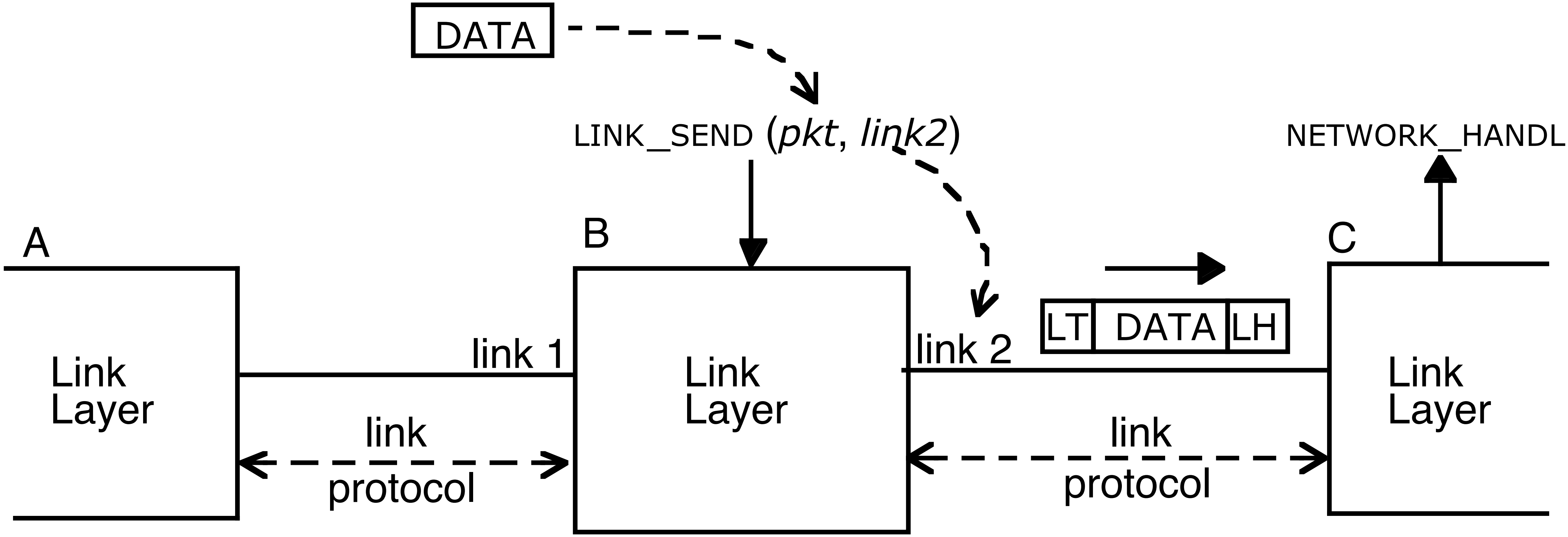
RPC client stub

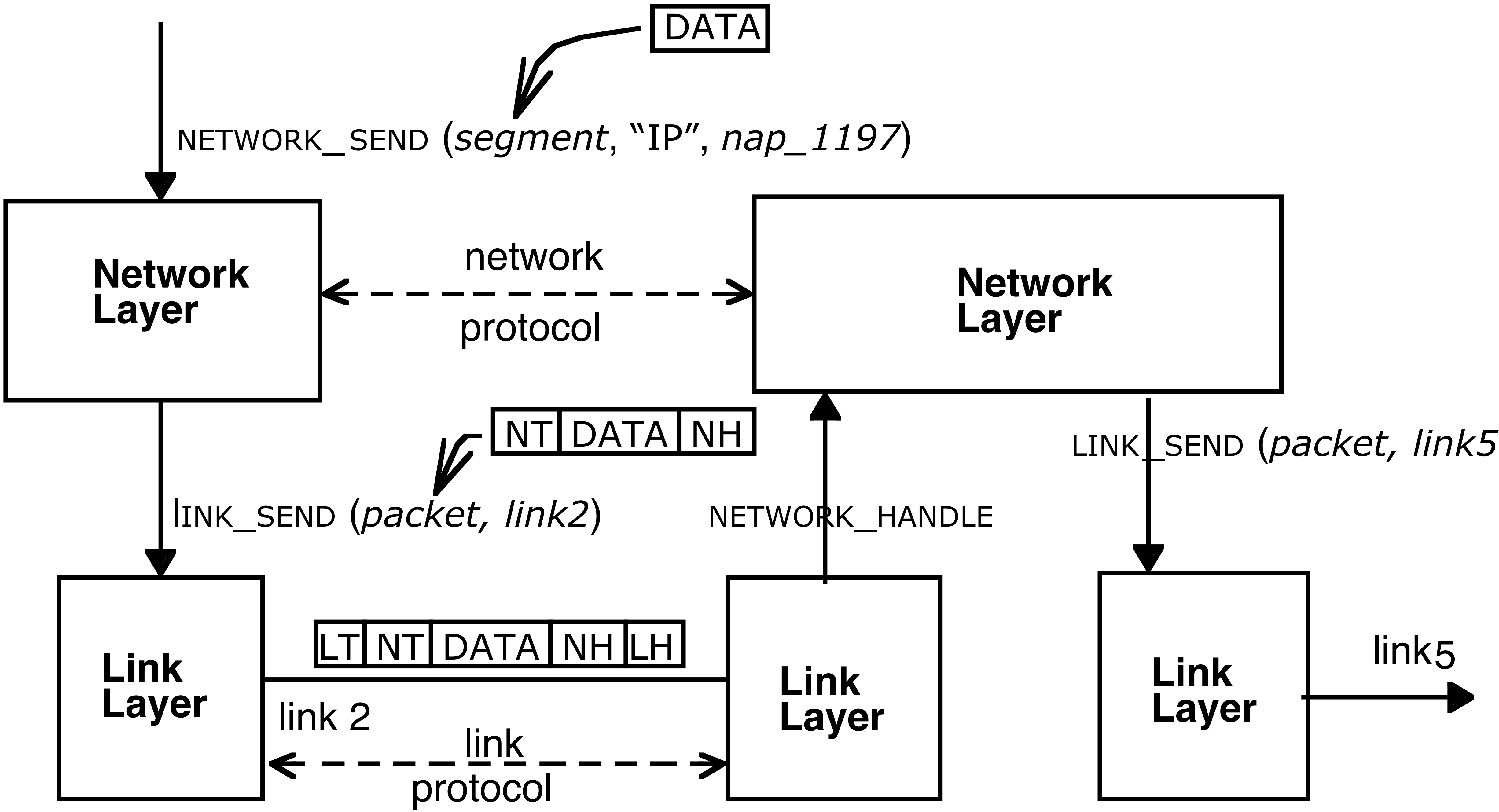
presentation protocol

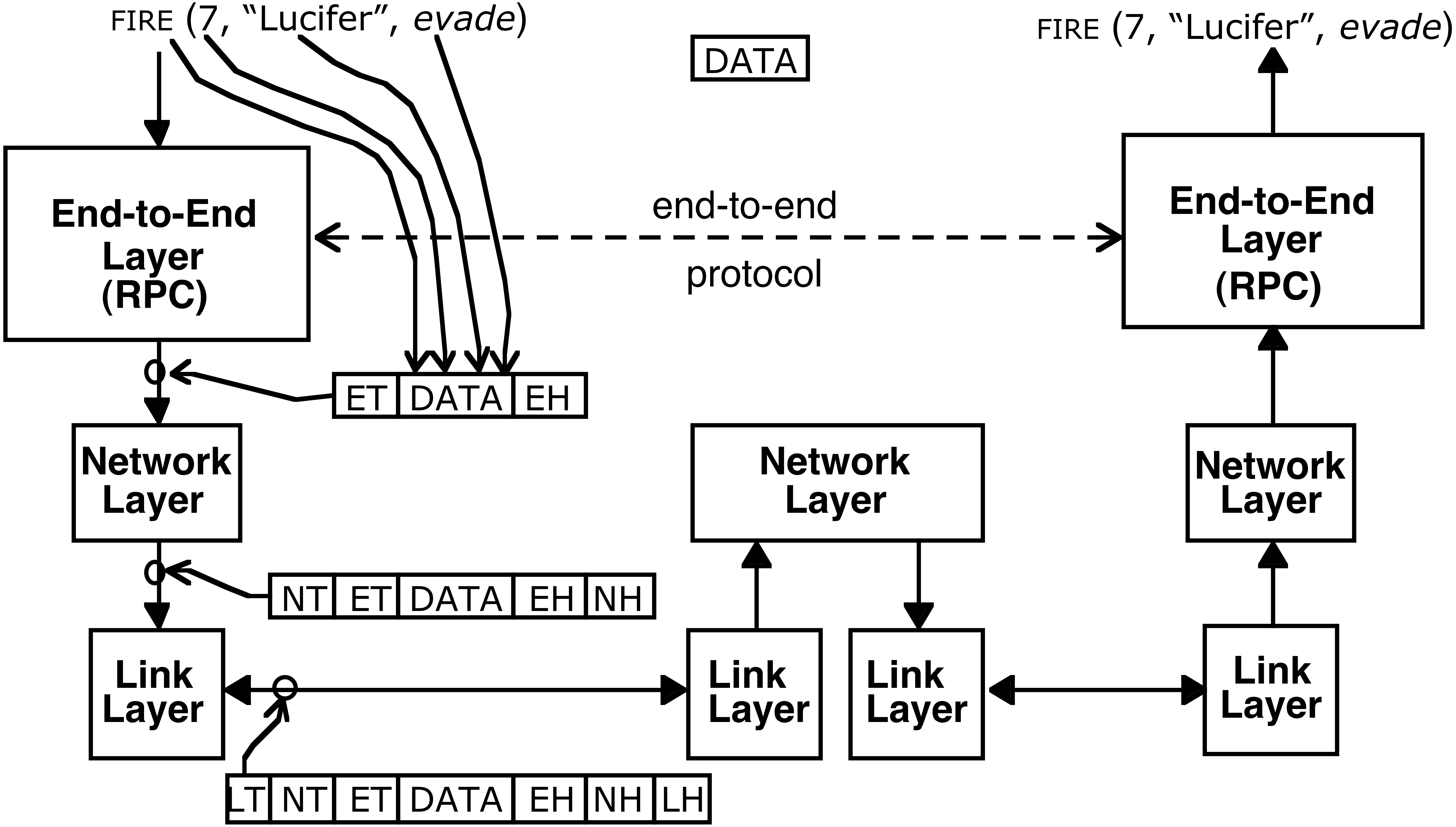
RPC service stub


Main program
application protocol
called procedure
fire
(return)
(return)
fire
RPC client stub
presentation protocol
RPC service stub
send_
message
receive_
message
send_
message
receive_
message
Client network package
transport protocol
Service network package


Layer One
A
B
C
D
Layer Two
J
K
L
Layer Three
X
Y
Z


DATA
LINK_SEND (*pkt*, *link2*)
NETWORK_HANDLE
A
B
C
Link Layer
Link Layer
Link Layer
link 1
link 2
LT
DATA
LH
link protocol
link protocol


DATA
NETWORK_SEND (*segment*, “IP”, *nap_1197*)
Network Layer
network
protocol
Network Layer
NT DATA NH
lINK_SEND (*packet, link2*)
NETWORK_HANDLE
LINK_SEND (*packet, link5*)
Link Layer
LT NT DATA NH LH
link 2
link
protocol
Link Layer
Link Layer
link5


FIRE (7, “Lucifer”, *evade*)

DATA

FIRE (7, “Lucifer”, *evade*)

End-to-End Layer (RPC)

end-to-end

protocol

End-to-End Layer (RPC)

ET DATA EH

Network Layer

Network Layer

Network Layer

NT ET DATA EH NH

Link Layer

Link Layer

Link Layer

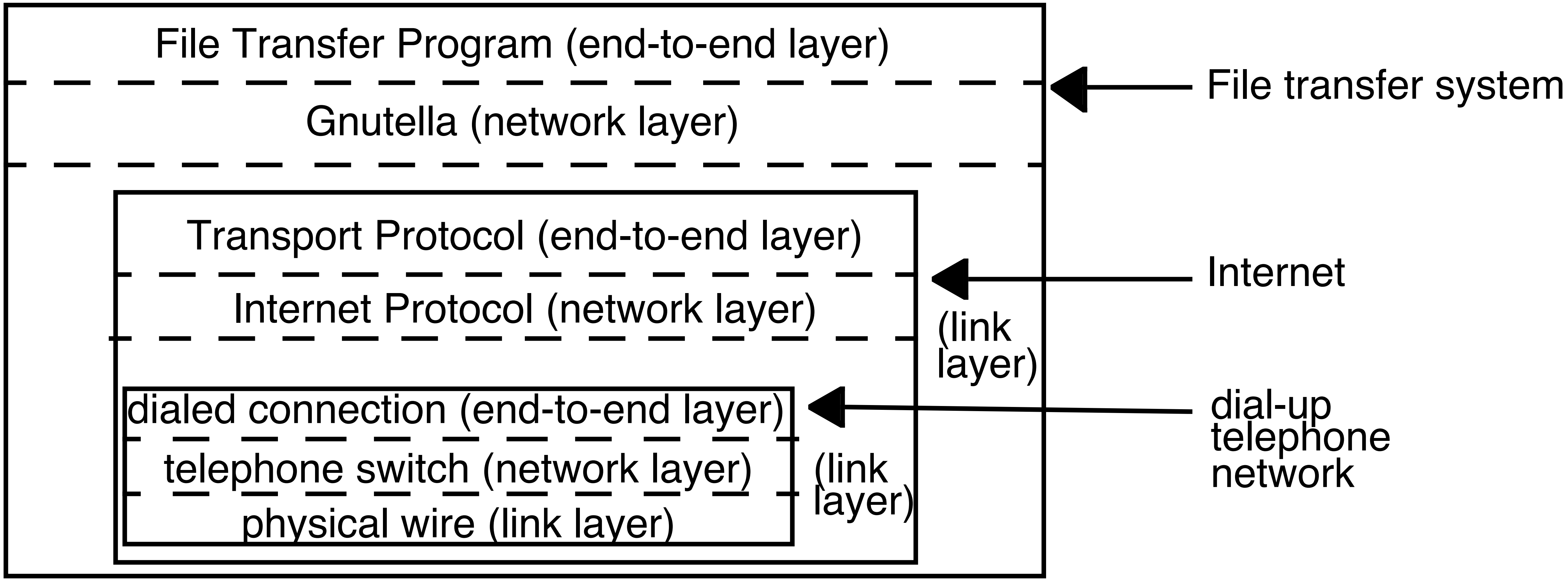
Link Layer

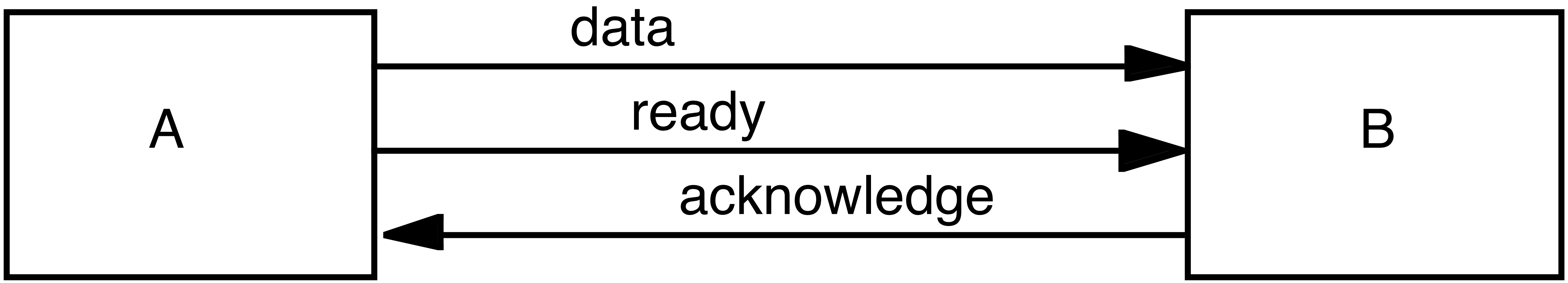
LT NT ET DATA EH NH LH

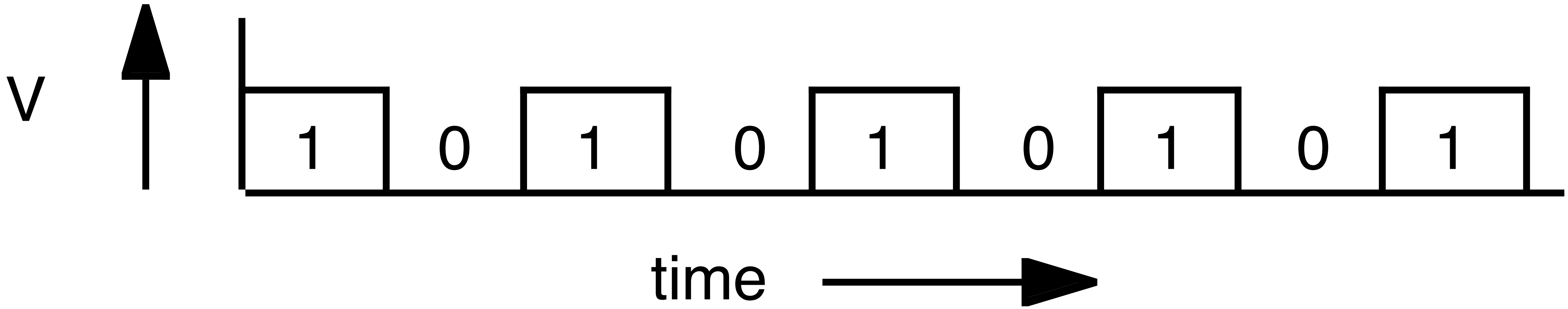
Principles of Computer System Design © Saltzer & Kaashoek 2009

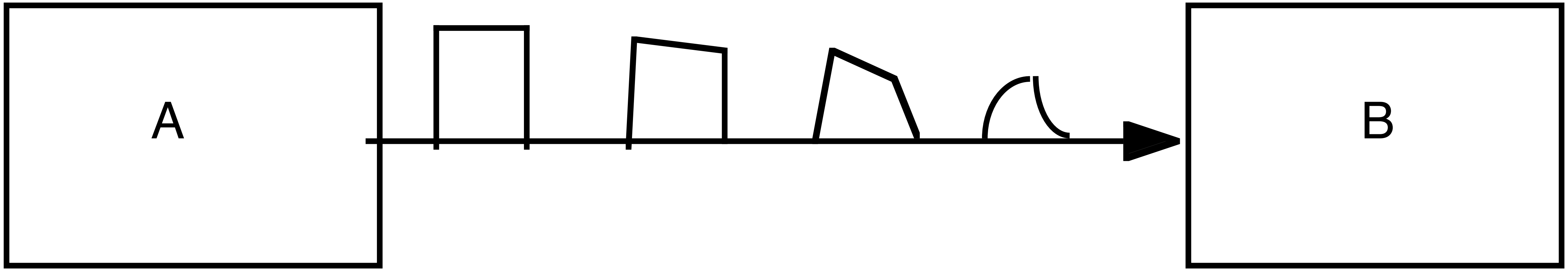
The end-to-end argument

The application knows best.


File Transfer Program (end-to-end layer)
Gnutella (network layer)
Transport Protocol (end-to-end layer)
Internet Protocol (network layer)
(link layer)
dialed connection (end-to-end layer)
telephone switch (network layer)
physical wire (link layer)
(link layer)
File transfer system
Internet
dial-up telephone network

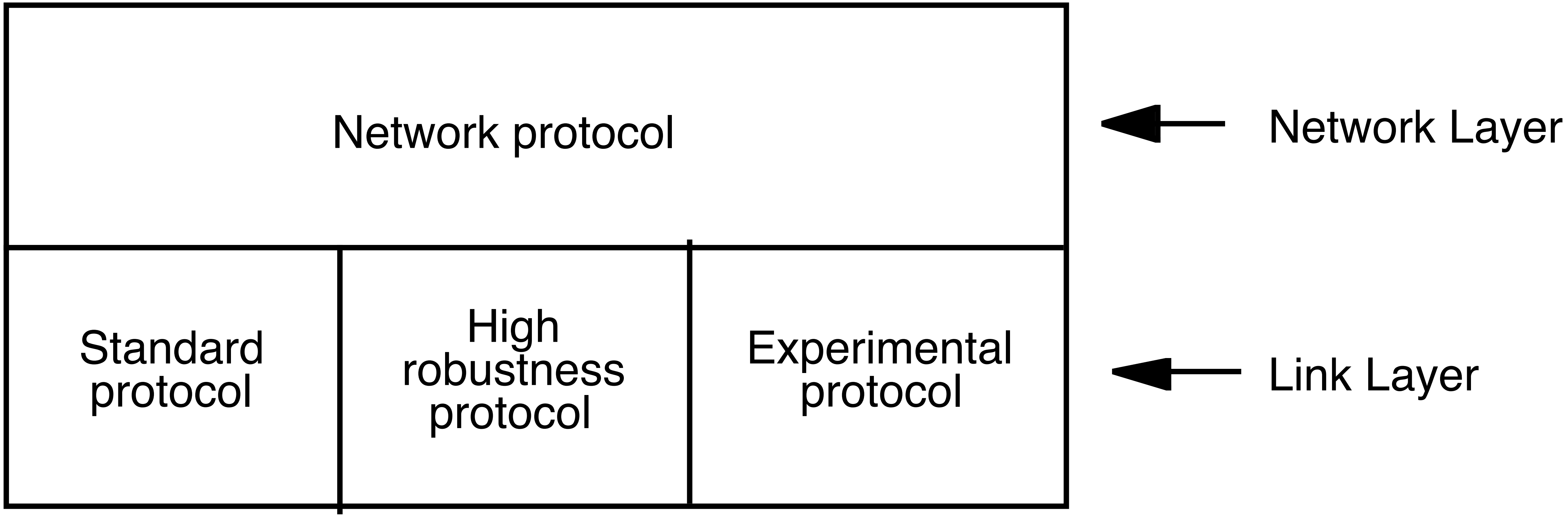

data
A
ready
B
acknowledge

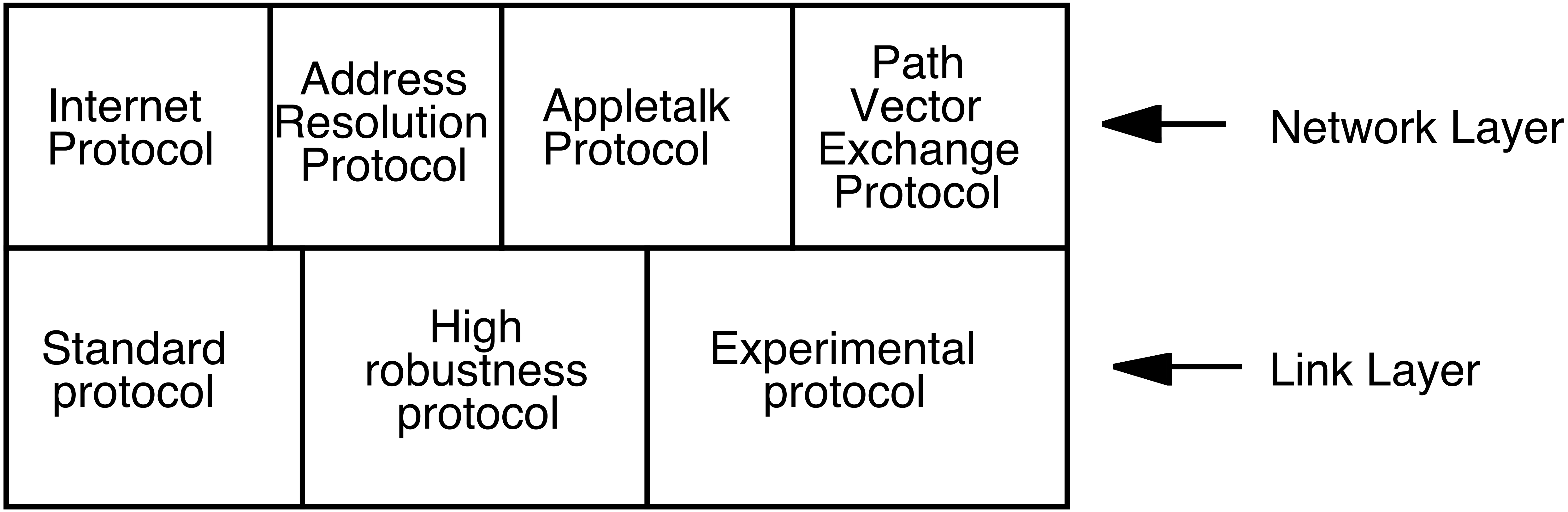

V
1
0
1
0
1
0
1
0
1
time


A
B

```
procedure FRAME_TO_BIT (frame_data, length)
    ones_in_a_row = 0
    for i from 1 to length do                   // First send frame contents
        SEND_BIT (frame_data[i]);
        if frame_data[i] = 1 then
            ones_in_a_row ← ones_in_a_row + 1;
            if ones_in_a_row = 6 then
                SEND_BIT (0);                   // Stuff a zero so that data doesn't
                ones_in_a_row ← 0;              // look like a framing marker
        else
            ones_in_a_row ← 0;
    for i from 1 to 7 do                        // Now send framing marker.
        SEND_BIT (1)
```

```
procedure BIT_TO_FRAME (rcvd_bit)
    ones_in_a_row integer initially 0
    if ones_in_a_row < 6 then
        bits_in_frame ← bits_in_frame + 1
        frame_data[bits_in_frame] ← rcvd_bit
        if rcvd_bit = 1 then ones_in_a_row ← ones_in_a_row + 1
        else ones_in_a_row ← 0
    else                    // This may be a seventh one-bit in a row, check it out.
        if rcvd_bit = 0 then
            ones_in_a_row ← 0               // Stuffed bit, don't use it.
        else                                // This is the end-of-frame marker
            LINK_RECEIVE (frame_data, (bits_in_frame - 6), link_id)
            bits_in_frame ← 0
            ones_in_a_row ← 0
```


Network protocol
Network Layer
Standard protocol
High robustness protocol
Experimental protocol
Link Layer



Internet Protocol

Address Resolution Protocol

Appletalk Protocol

Path Vector Exchange Protocol

Network Layer

Standard protocol

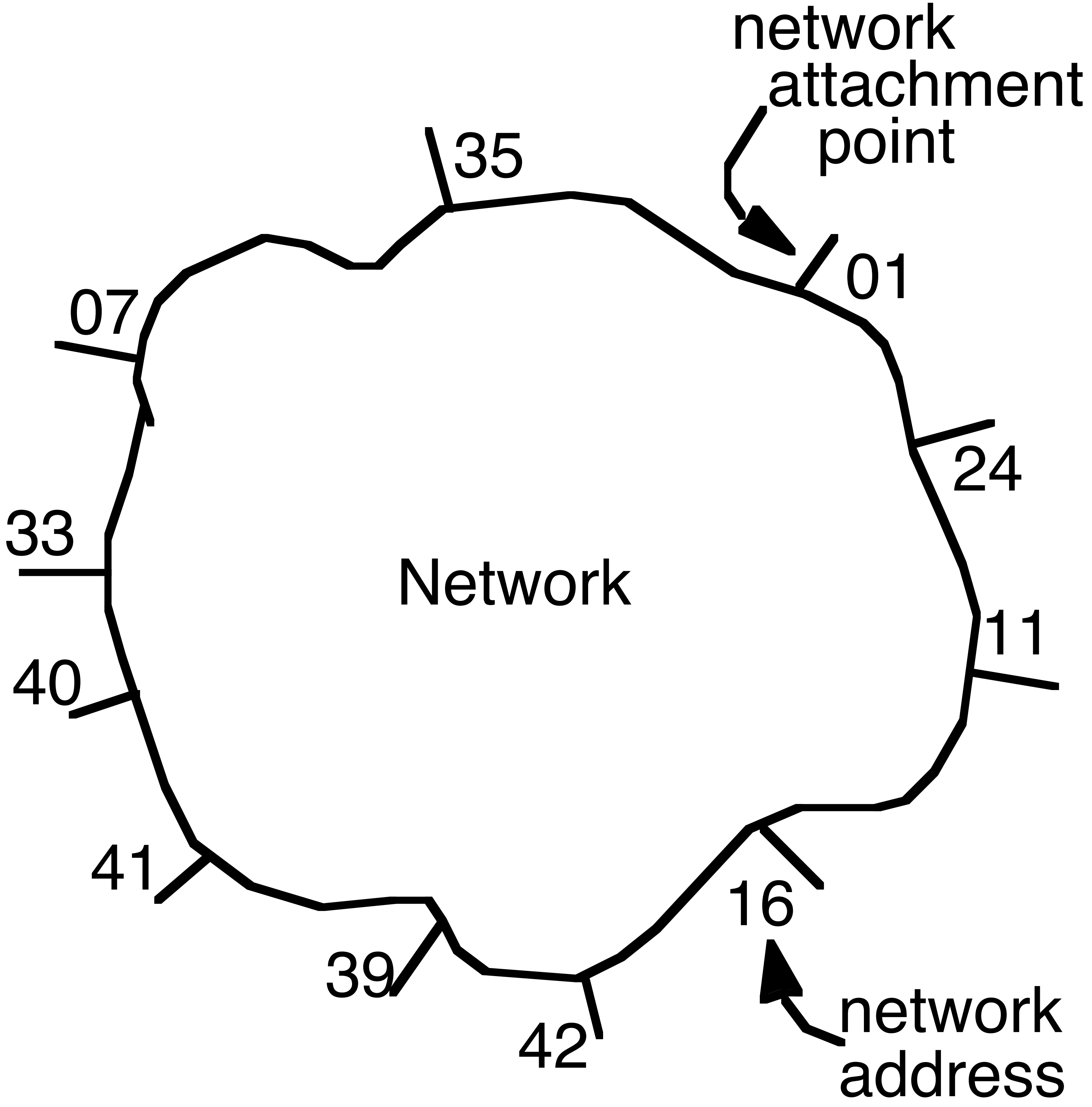
High robustness protocol

Experimental protocol

Link Layer

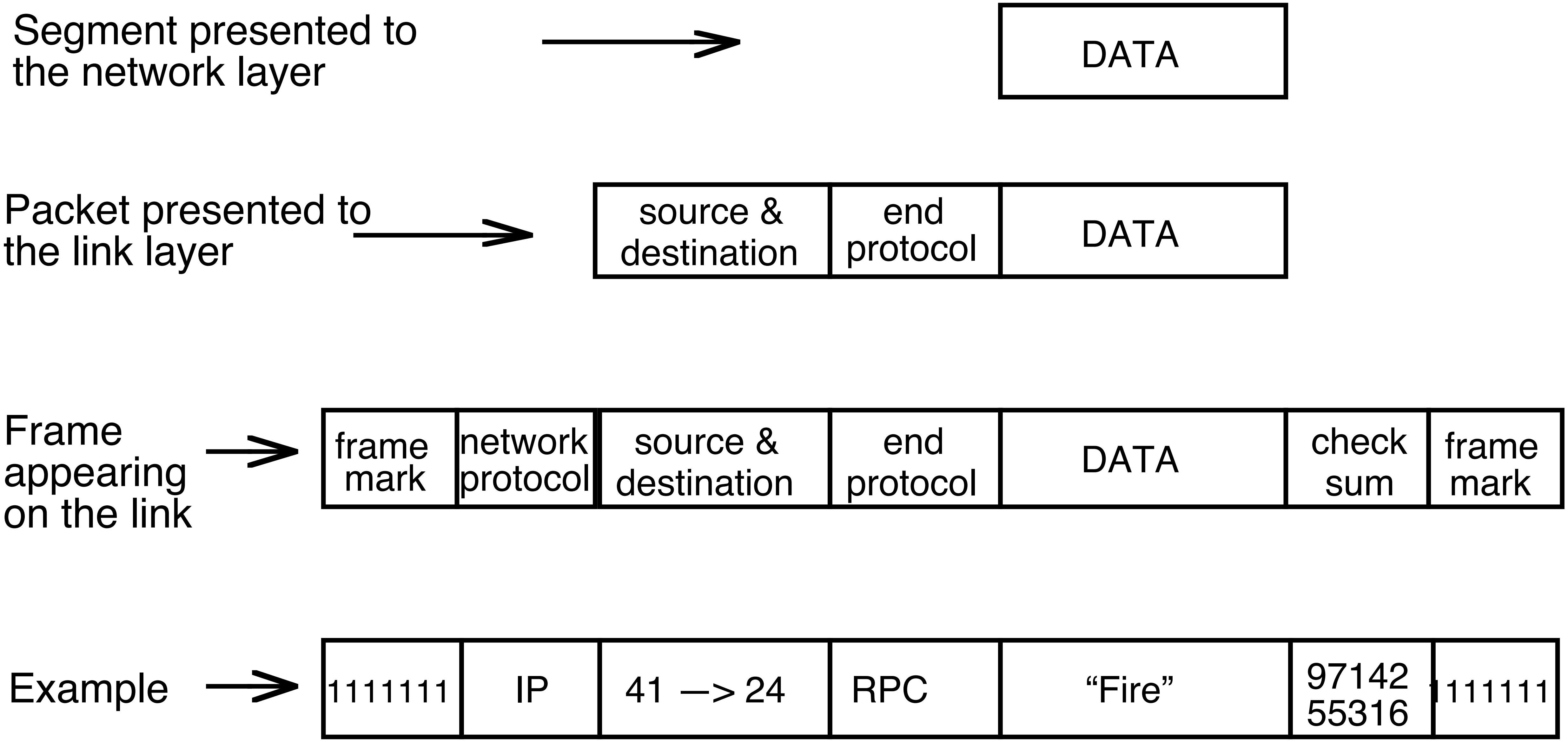
```
 structure frame
    structure checked_contents
        bit_string net_protocol                          // multiplexing parameter
        bit_string payload                               // payload data
    bit_string checksum

procedure LINK_SEND (data_buffer, link_identifier, link_protocol, network_protocol)
    frame instance outgoing_frame
    outgoing_frame.checked_contents.payload ← data_buffer
    outgoing_frame.checked_contents.net_protocol ← data_buffer.network_protocol
    frame_length ← LENGTH (data_buffer) + header_length
    outgoing_frame.checksum ← CHECKSUM (frame.checked_contents, frame_length)
    sendproc ← link_protocol[that_link.protocol]                // Select link protocol.
    sendproc (outgoing_frame, frame_length, link_identifier)    // Send frame.
```

```
procedure LINK_RECEIVE (received_frame, length, link_id)
    frame instance received_frame
    if CHECKSUM (received_frame.checked_contents, length) =
                                                      received_frame.checksum
        then                                          // Pass good packets up to next layer.
        good_frame_count ← good_frame_count + 1;
        GIVE_TO_NETWORK_HANDLER (received_frame.checked_contents.payload,
                            received_frame.checked_contents.net_protocol);
    else bad_frame_count ← bad_frame_count + 1     // Just count damaged frame.

// Each network layer protocol handler must call SET_HANDLER before the first packet
// for that protocol arrives…

procedure SET_HANDLER (handler_procedure, handler_protocol)
    net_handler[handler_protocol] ← handler_procedure

procedure GIVE_TO_NETWORK_HANDLER (received_packet, network_protocol)
    handler ← net_handler[network_protocol]
    if (handler ≠ NULL) call handler(received_packet, network_protocol)
    else unexpected_protocol_count ← unexpected_protocol_count + 1
```


network
attachment
point
35
01
07
24
33
Network
11
40
16
41
39
42
network
address

```
  structure packet
     bit_string source
     bit_string destination
     bit_string end_protocol
     bit_string payload

procedure NETWORK_SEND (segment_buffer, destination,
                                        network_protocol, end_protocol)
    packet instance outgoing_packet
    outgoing_packet.payload ← segment_buffer
    outgoing_packet.end_protocol ← end_protocol
    outgoing_packet.source ← MY_NETWORK_ADDRESS
    outgoing_packet.destination ← destination
    NETWORK_HANDLE (outgoing_packet, net_protocol)
```

```
procedure NETWORK_HANDLE (net_packet, net_protocol)
    packet instance net_packet
    if net_packet.destination ≠ MY_NETWORK_ADDRESS then
        next_hop ← LOOKUP (net_packet.destination, forwarding_table)
        LINK_SEND (net_packet, next_hop, link_protocol, net_protocol)
    else
        GIVE_TO_END_LAYER (net_packet.payload,
                    net_packet.end_protocol, net_packet.source)
```

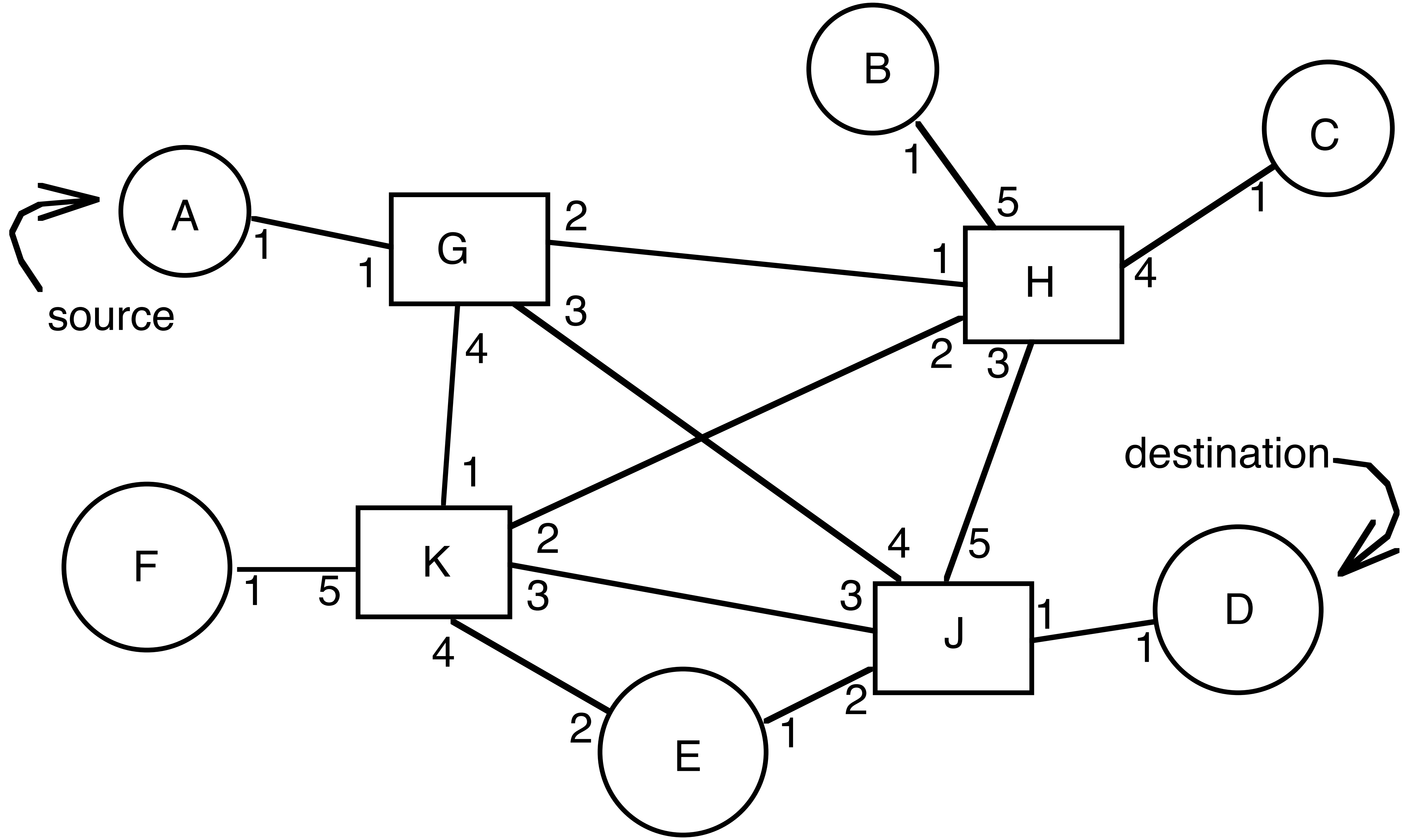


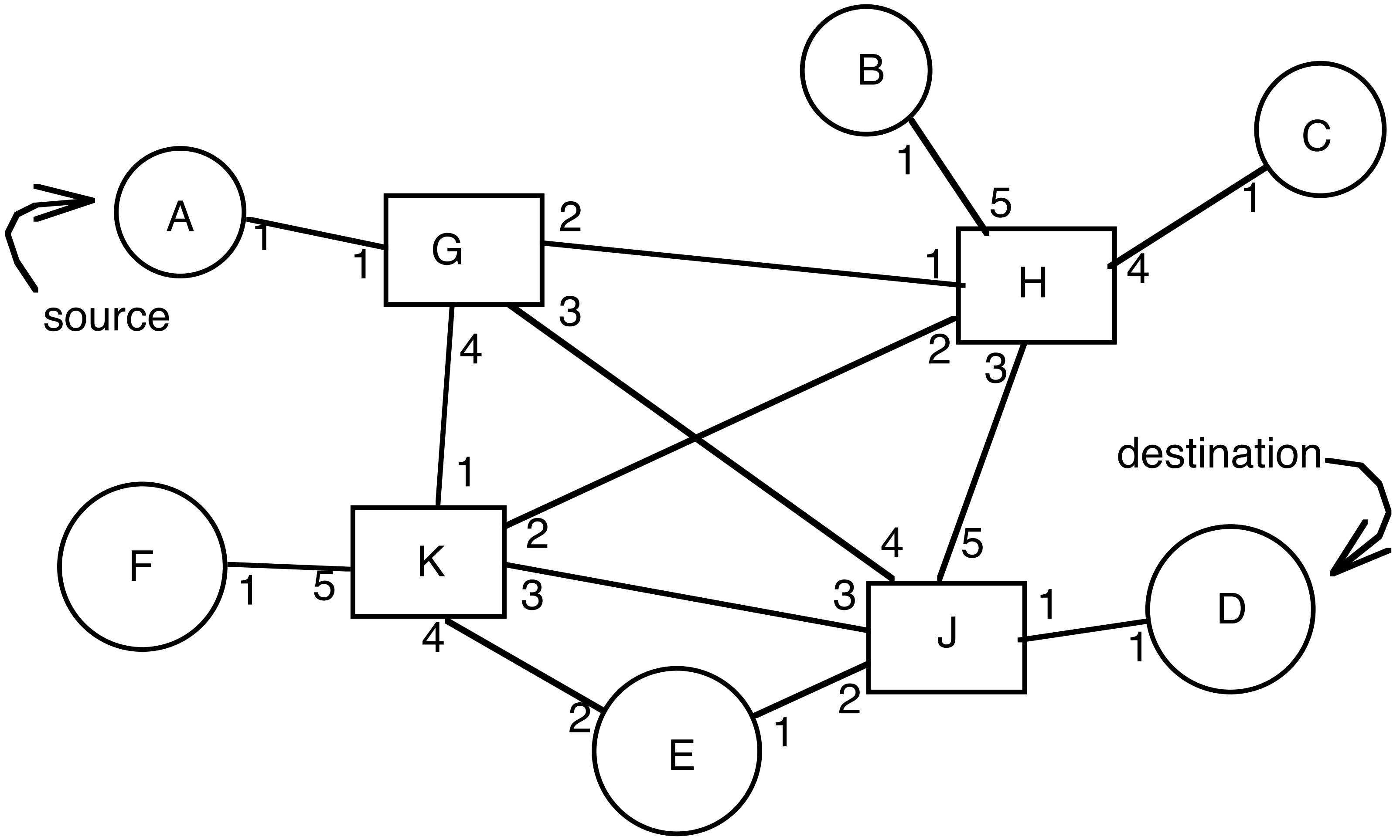
Segment presented to the network layer → DATA

Packet presented to the link layer → source & destination | end protocol | DATA

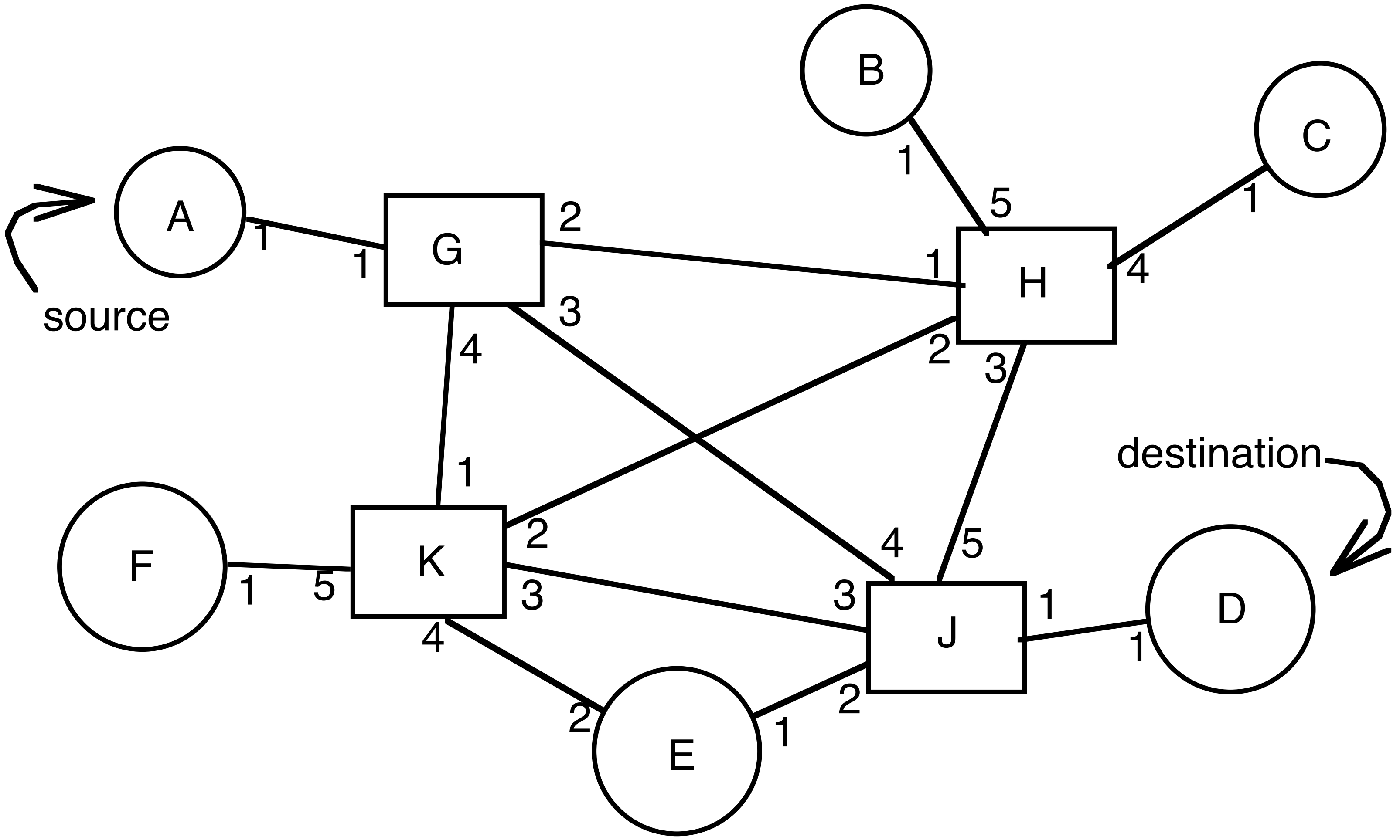
Frame appearing on the link → frame mark | network protocol | source & destination | end protocol | DATA | check sum | frame mark

Example → 1111111 | IP | 41 —> 24 | RPC | “Fire” | 97142 55316 | 1111111

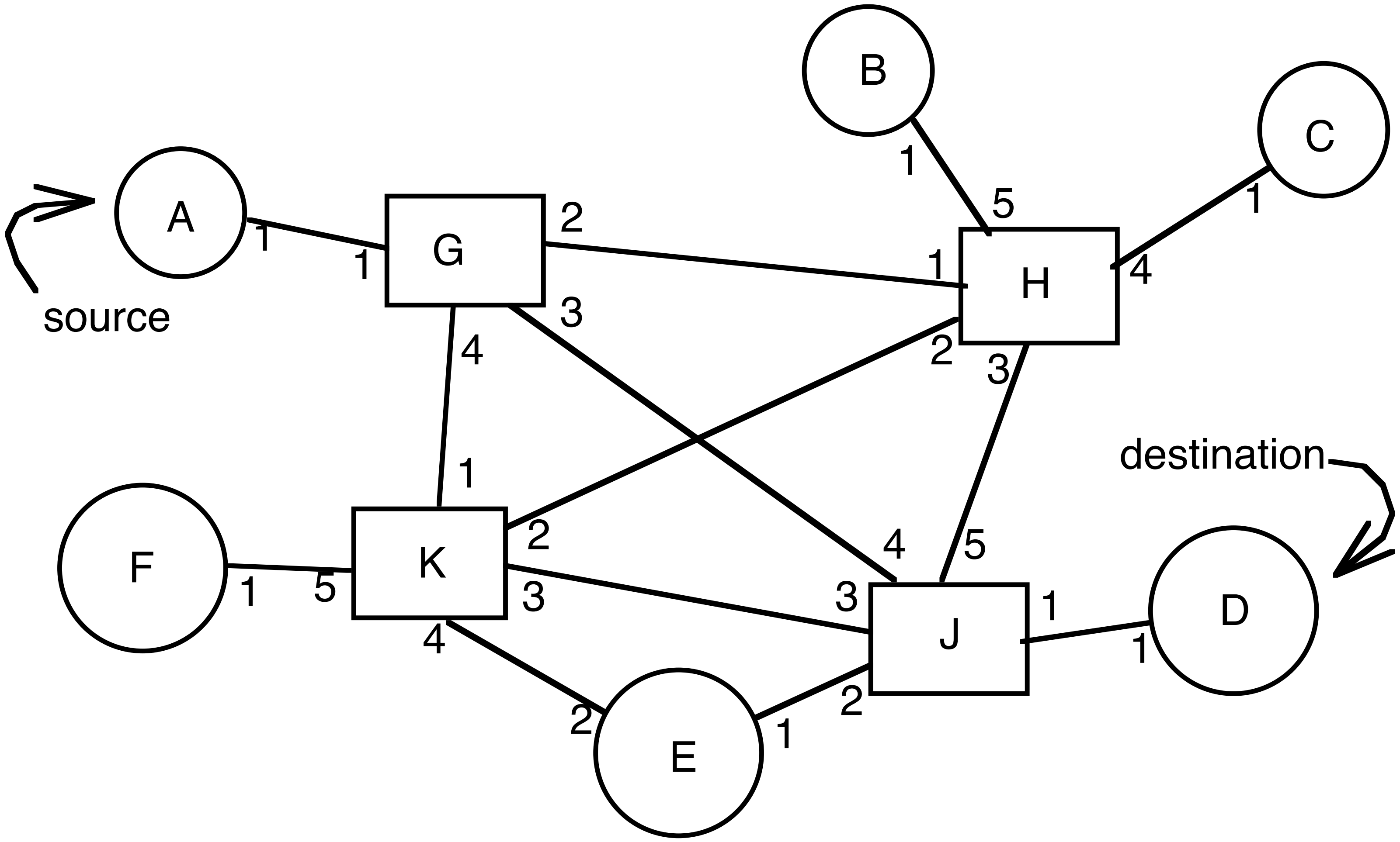

source
A
1
1
G
2
3
4
B
1
5
C
1
1
H
4
2
3
destination
F
1
5
K
1
2
3
4
4
5
3
J
1
1
D
2
2
E
1



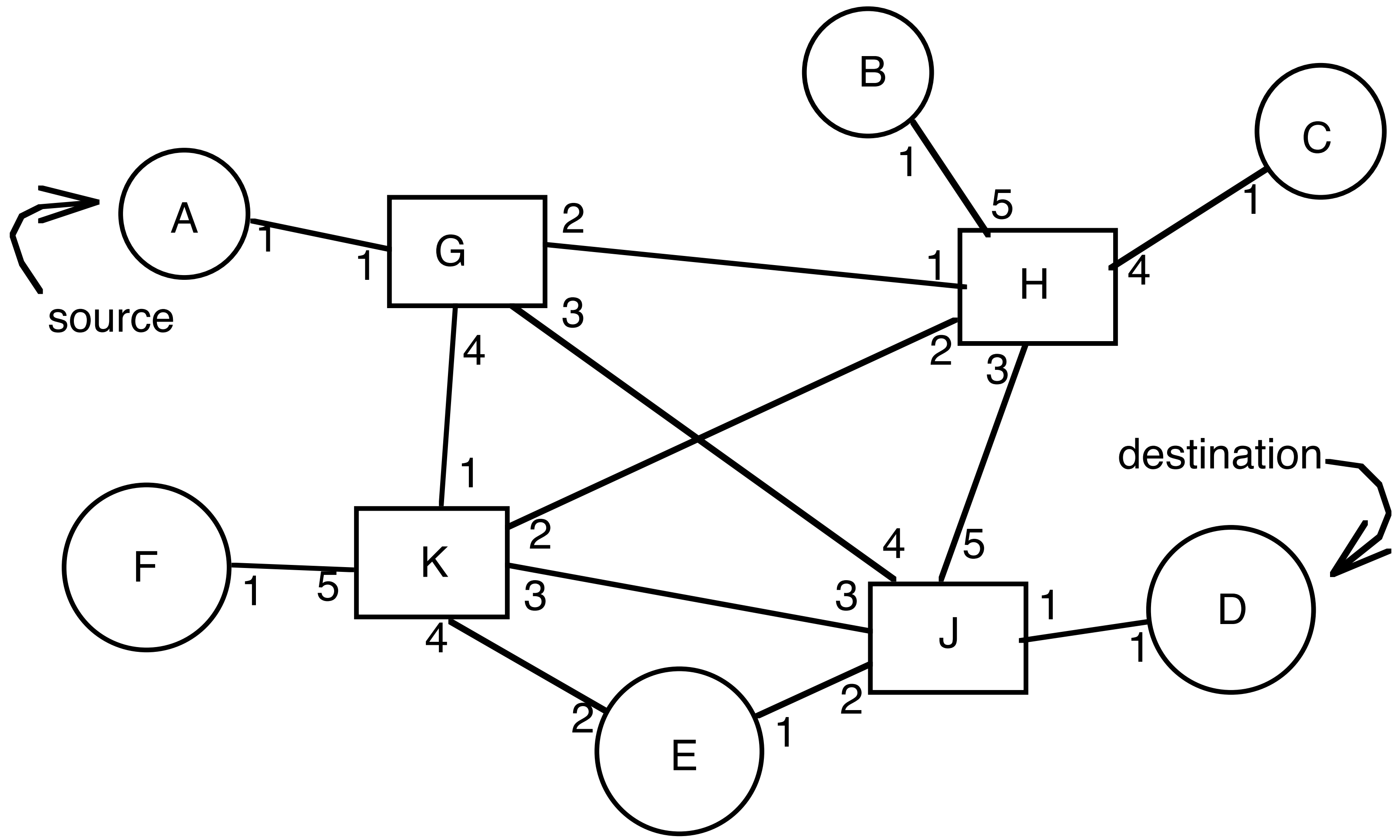
destination	link
A	end-layer
all other	1



destination	link
A	1
B	2
C	2
D	3
E	4
F	4
G	end-layer
H	2
J	3
K	4


B
C
1
5
1
A
1
G
2
1
H
4
1
source
3
4
2
3
destination
1
F
K
2
4
5
1
5
3
3
J
1
D
4
1
2
E
1
2

to	path
G	< >



From A,
via link 1

to	path
A	< >

From H,
via link 2:

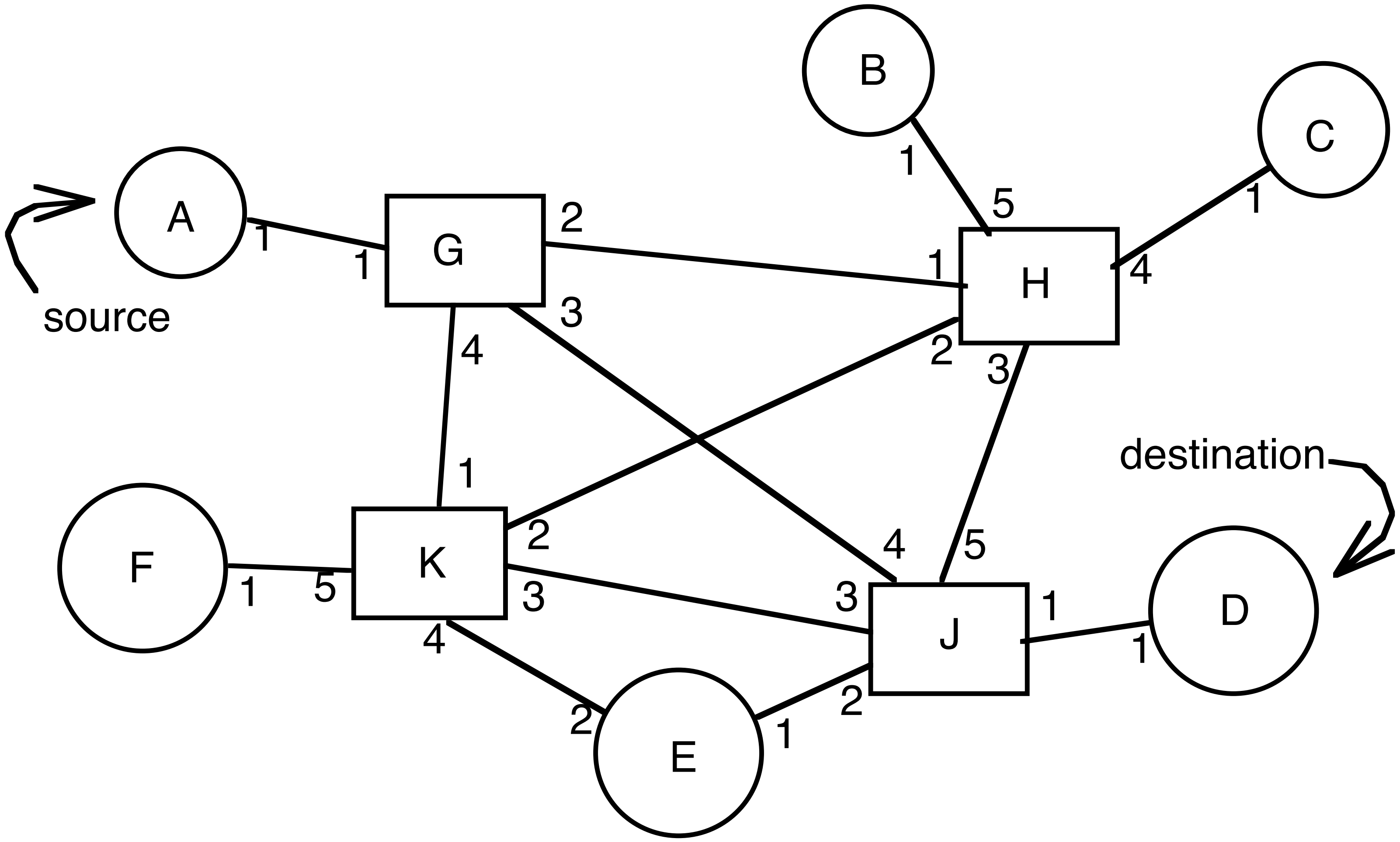
to	path
H	< >

From J,
via link 3:

to	path
J	< >

From K,
via link 4:

to	path
K	< >

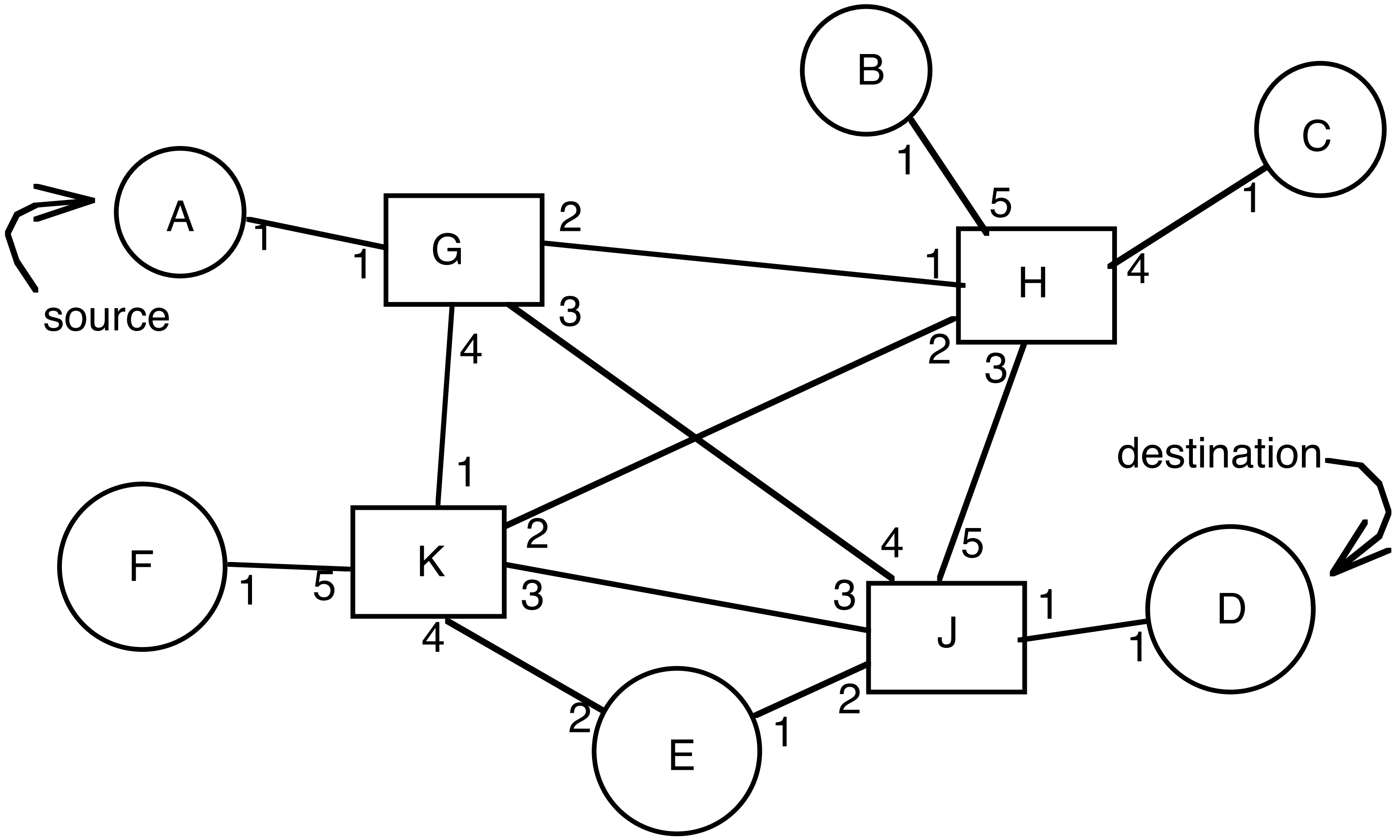


path vector

to	path
A	<A>
G	< >
H	<H>
J	<J>
K	<K>

forwarding table

to	link
A	1
G	end-layer
H	2
J	3
K	4



From A,
via link 1

to	path
A	< >
G	<G>

From H,
via link 2:

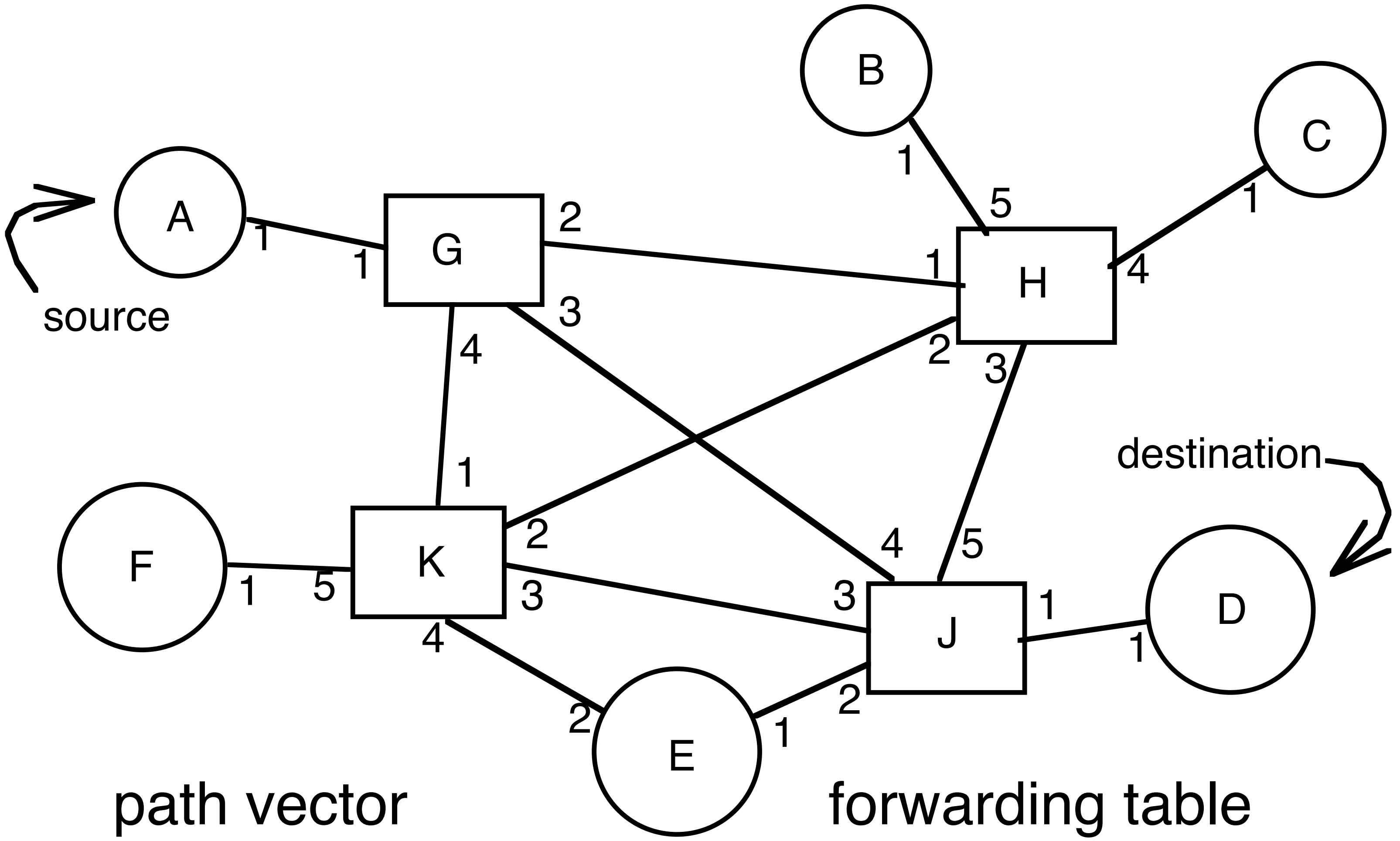
to	path
B	<B>
C	<C>
G	<G>
H	< >
J	<J>
K	<K>

From J,
via link 3:

to	path
D	<D>
E	<E>
G	<G>
H	<H>
J	< >
K	<K>

From K,
via link 4:

to	path
E	<E>
F	<F>
G	<G>
H	<H>
J	<J>
K	< >



path vector

to	path
A	<A>
B	<H, B>
C	<H, C>
D	<J, D>
E	<J, E>
F	<K, F>
G	< >
H	<H>
J	<J>
K	<K>

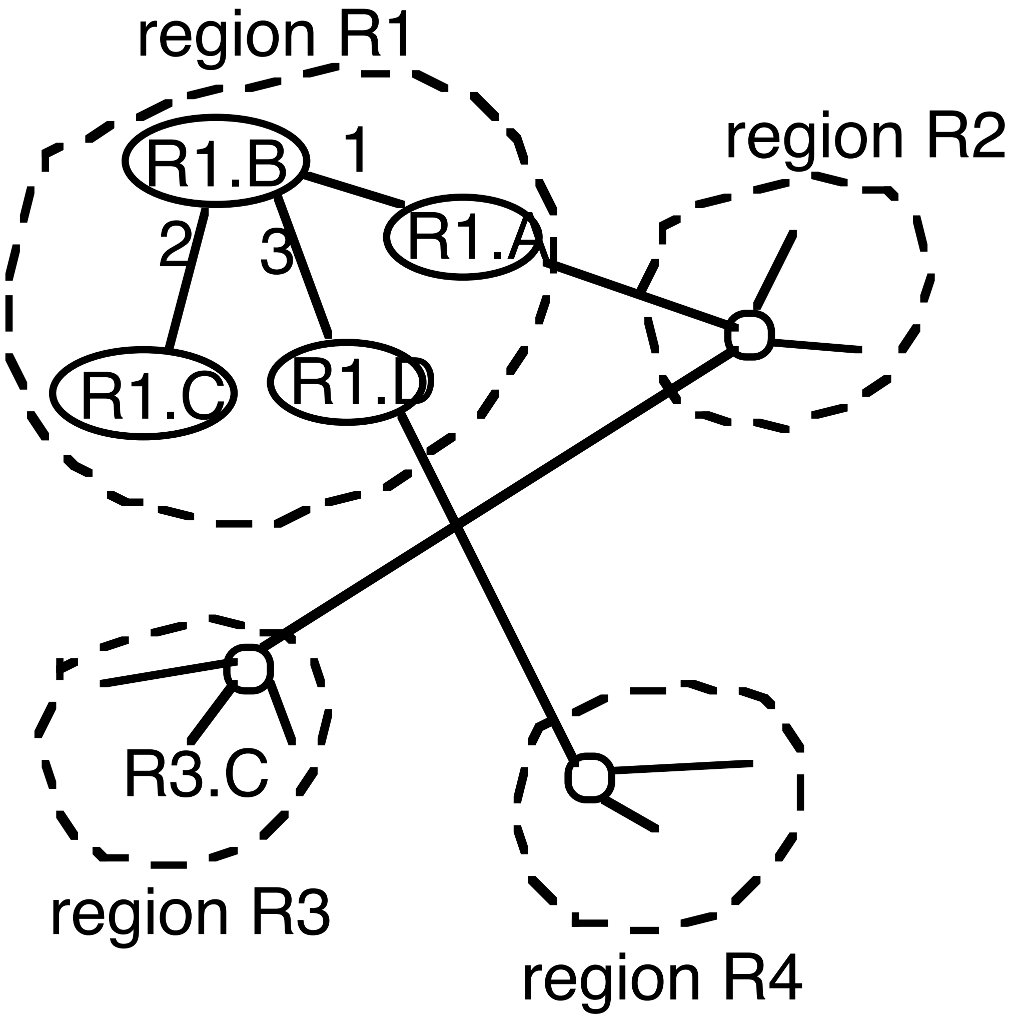
forwarding table

to	link
A	1
B	2
C	2
D	3
E	3
F	4
G	end-layer
H	2
J	3
K	4

```
// Maintain routing and forwarding tables.

vector associative array            // vector[d_addr] contains path to destination d_addr
neighbor_vector instance of vector   // A path vector received from some neighbor
my_vector  instance of vector       // My current path vector.
addr associative array              // addr[j] is the address of the network attachment
                                    // point at the other end of link j.
                                    // my_addr is address of my network attachment point.
                                    // A path is a parsable list of addresses, e.g. {a,b,c,d}

procedure main()                              // Initialize, then start advertising.
    SET_TYPE_HANDLER (HANDLE_ADVERTISEMENT, exchange_protocol)
    clear my_vector;                          // Listen for advertisements
    do occasionally                           // and advertise my paths
        for each j in link_ids do             // to all of my neighbors.
            status ← SEND_PATH_VECTOR (j, my_addr, my_vector, exch_protocol)
            if status ≠ 0 then                // If the link was down,
                clear new_vector              // forget about any paths
                FLUSH_AND_REBUILD (j)         // that start with that link.
```

```
procedure HANDLE_ADVERTISEMENT (advt, link_id)          // Called when an advt arrives.
    addr[link_id] ← GET_SOURCE (advt)                   // Extract neighbor’s address
    neighbor_vector ← GET_PATH_VECTOR (advt)            //    and path vector.
    for each neighbor_vector.d_addr do                  // Look for better paths.
        new_path ←{addr[link_id], neighbor_vector[d_addr]}      // Build potential path.
        if my_addr is not in new_path then              // Skip it if I’m in it.
            if my_vector[d_addr] = NULL) then           // Is it a new destination?
                my_vector[d_addr] ← new_path            // Yes, add this one.
            else                                        // Not new; if better, use it.
                my_vector[d_addr] ← SELECT_PATH (new_path, my_vector[d_addr])
    FLUSH_AND_REBUILD (link_id)
```

```
procedure SELECT_PATH (new, old)              // Decide if new path is better than old one.
    if first_hop(new) = first_hop(old) then return new   // Update any path we were
                                                         // already using.
    else if length(new) ≥ length(old) then return old    // We know a shorter path, keep
    else return new                                      // OK, the new one looks better.

procedure FLUSH_AND_REBUILD (link_id)         // Flush out stale paths from this neighbor.
    for each my_vector,d_addr
        if first_hop(my_vector[d_addr]) = addr[link_id] and new_vector[d_addr] = NULL
            then
            delete my_vector[d_addr]          // Delete paths that are not still advertised.
    REBUILD_FORWARDING_TABLE (my_vector, addr)           // Pass info to forwarder.
```



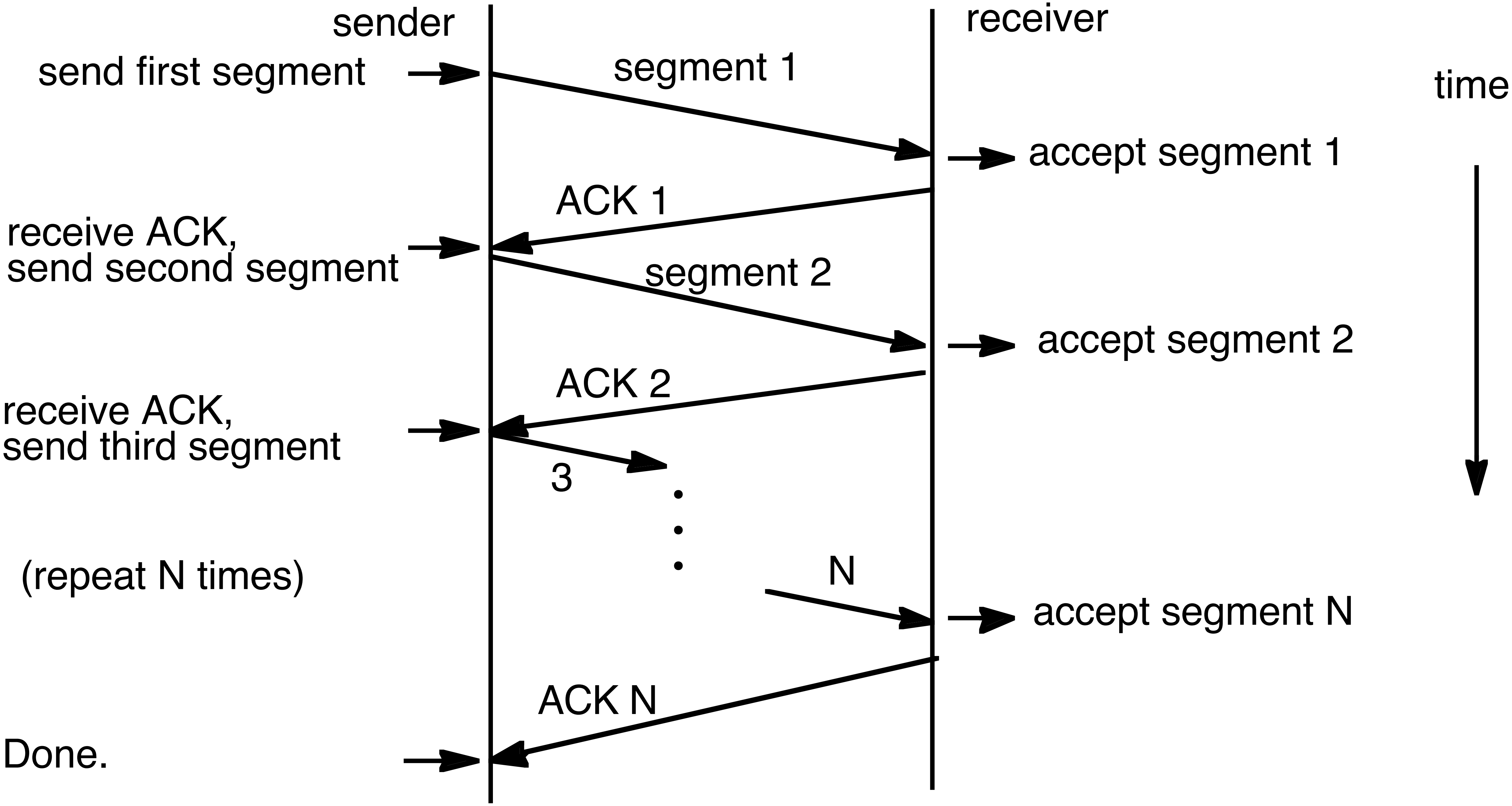
forwarding table in R1.B

region forwarding section

to	link
R1	local
R2	1
R3	1
R4	3

local forwarding section

to	link
R1.A	1
R1.B	end-layer
R1.C	2
R1.D	3



sender

receiver

send first segment

segment 1

time

accept segment 1

ACK 1

receive ACK,
send second segment

segment 2

accept segment 2

ACK 2

receive ACK,
send third segment

3

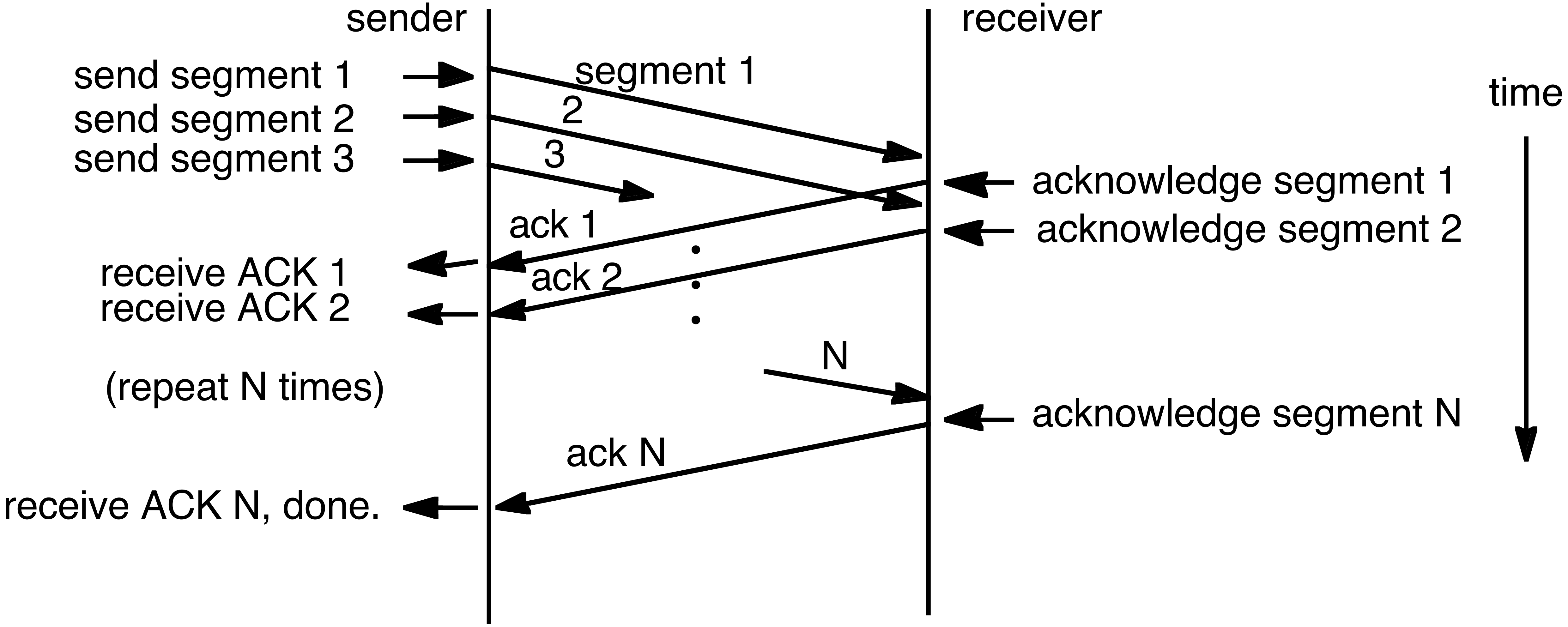
(repeat N times)

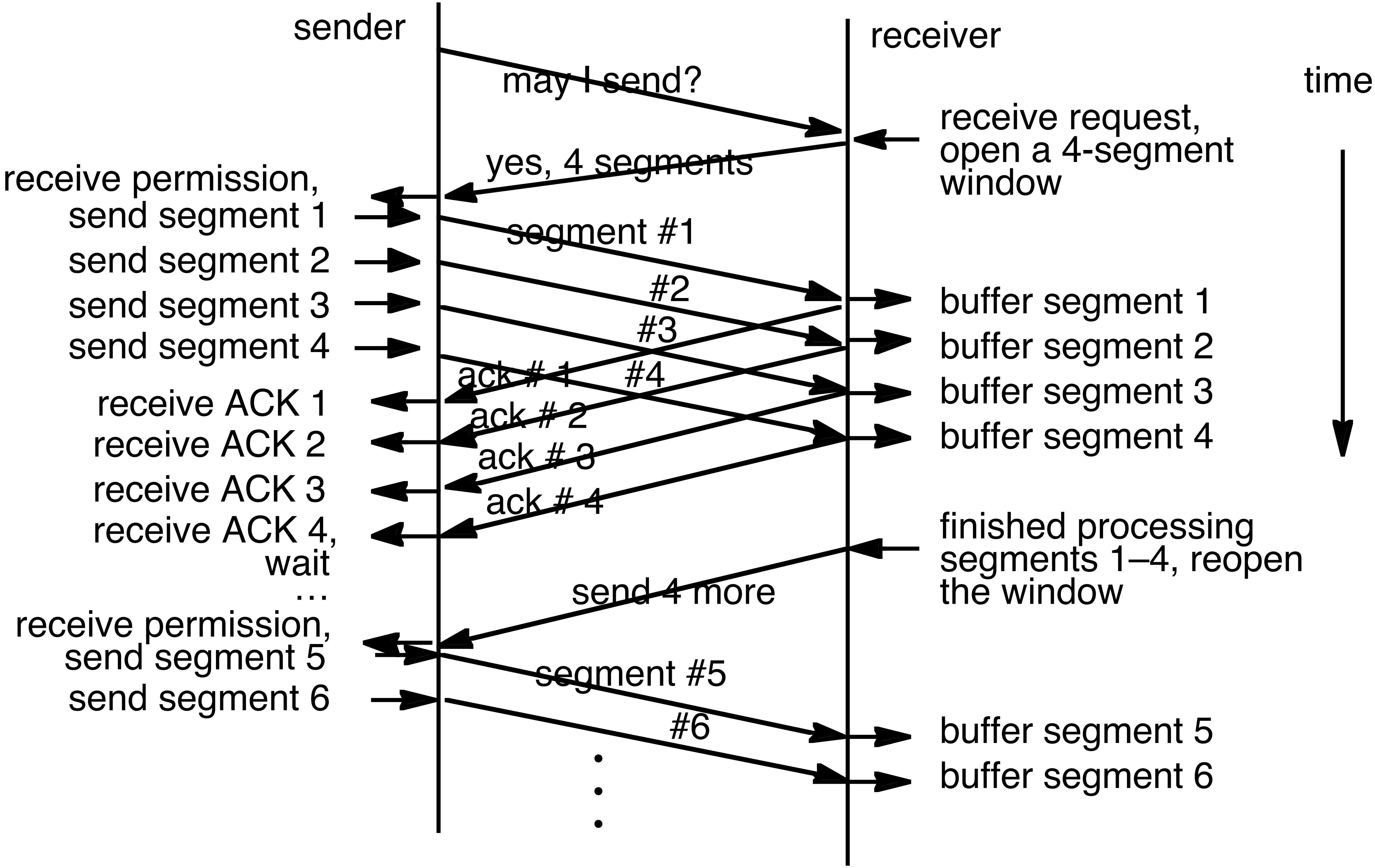
N

accept segment N

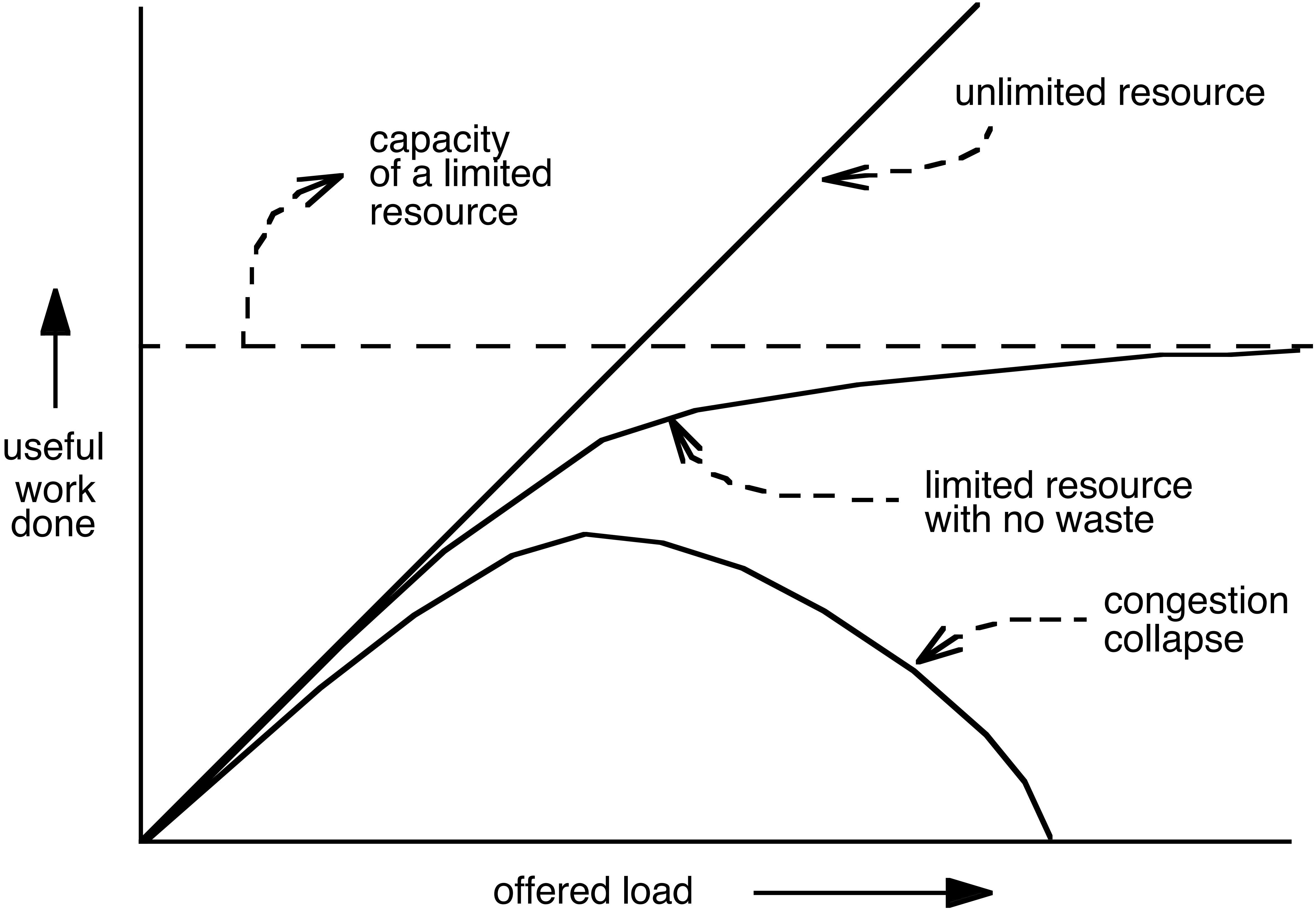
ACK N

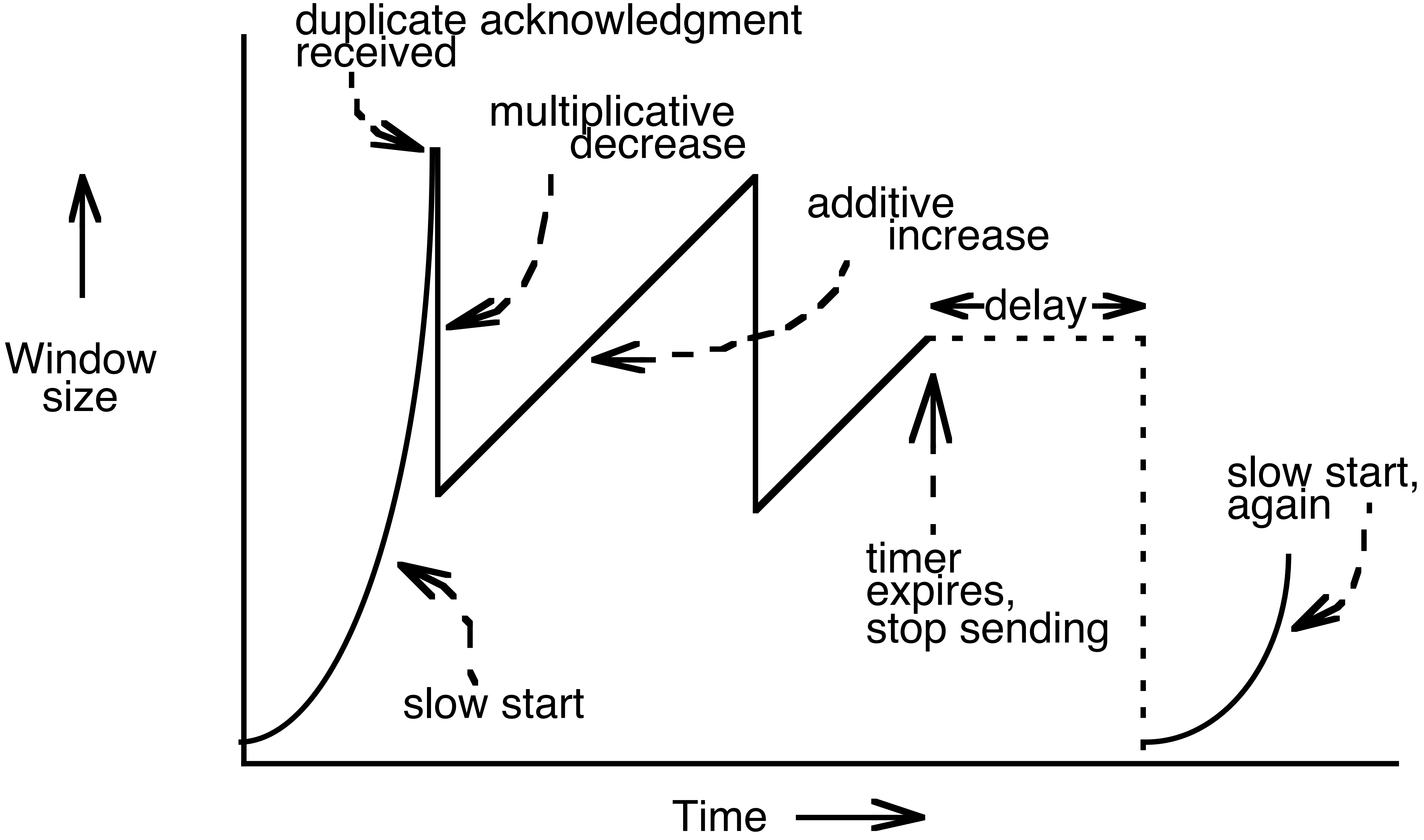
Done.


sender
receiver
send segment 1
send segment 2
send segment 3
segment 1
2
3
time
acknowledge segment 1
acknowledge segment 2
ack 1
receive ACK 1
ack 2
receive ACK 2
N
(repeat N times)
acknowledge segment N
ack N
receive ACK N, done.

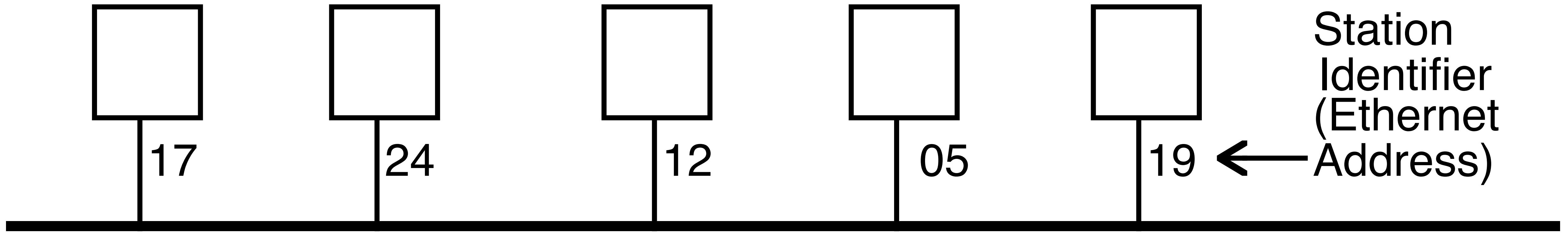


sender
receiver
time
may I send?
receive request, open a 4-segment window
yes, 4 segments
receive permission, send segment 1
segment #1
send segment 2
#2
send segment 3
#3
send segment 4
#4
buffer segment 1
buffer segment 2
buffer segment 3
buffer segment 4
ack # 1
receive ACK 1
ack # 2
receive ACK 2
ack # 3
receive ACK 3
ack # 4
receive ACK 4, wait
…
finished processing segments 1–4, reopen the window
send 4 more
receive permission, send segment 5
segment #5
send segment 6
#6
buffer segment 5
buffer segment 6

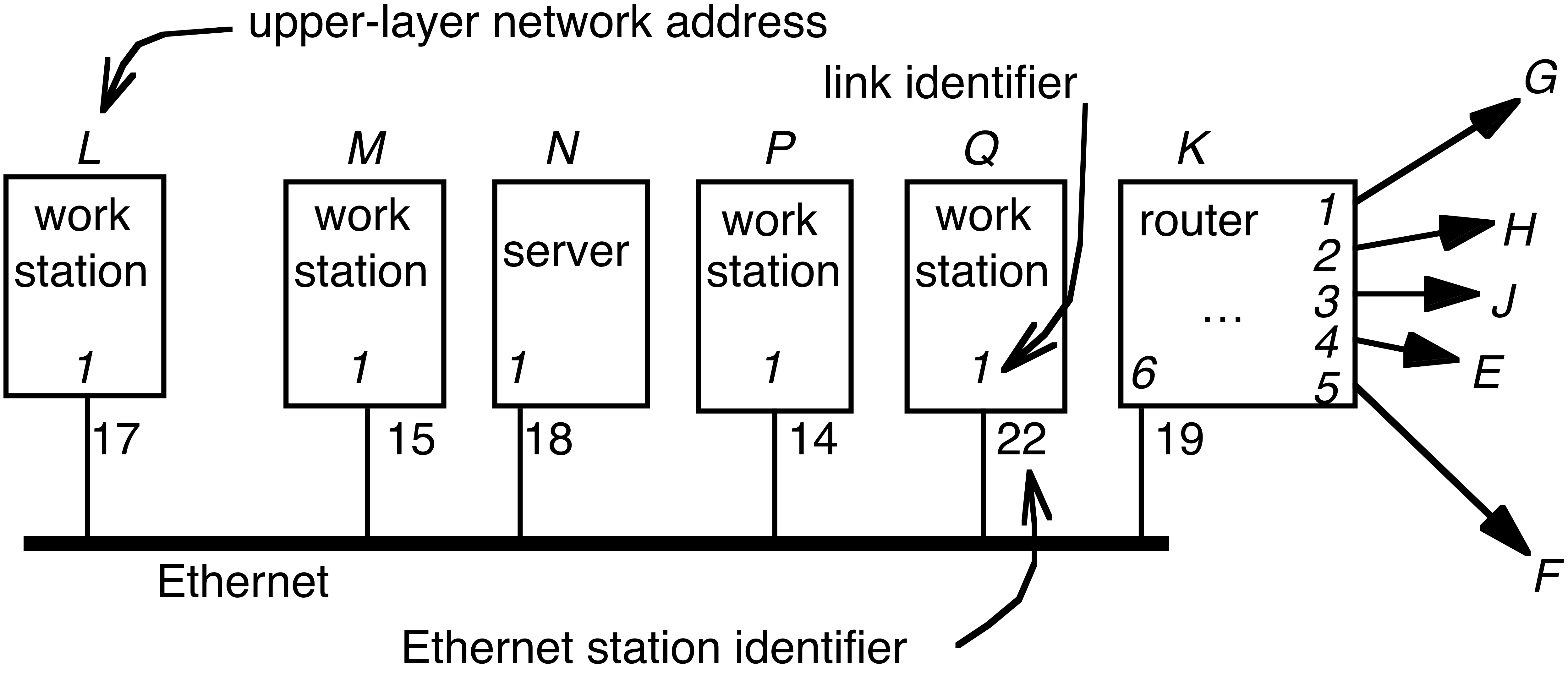

unlimited resource
capacity of a limited resource
useful work done
limited resource with no waste
congestion collapse
offered load

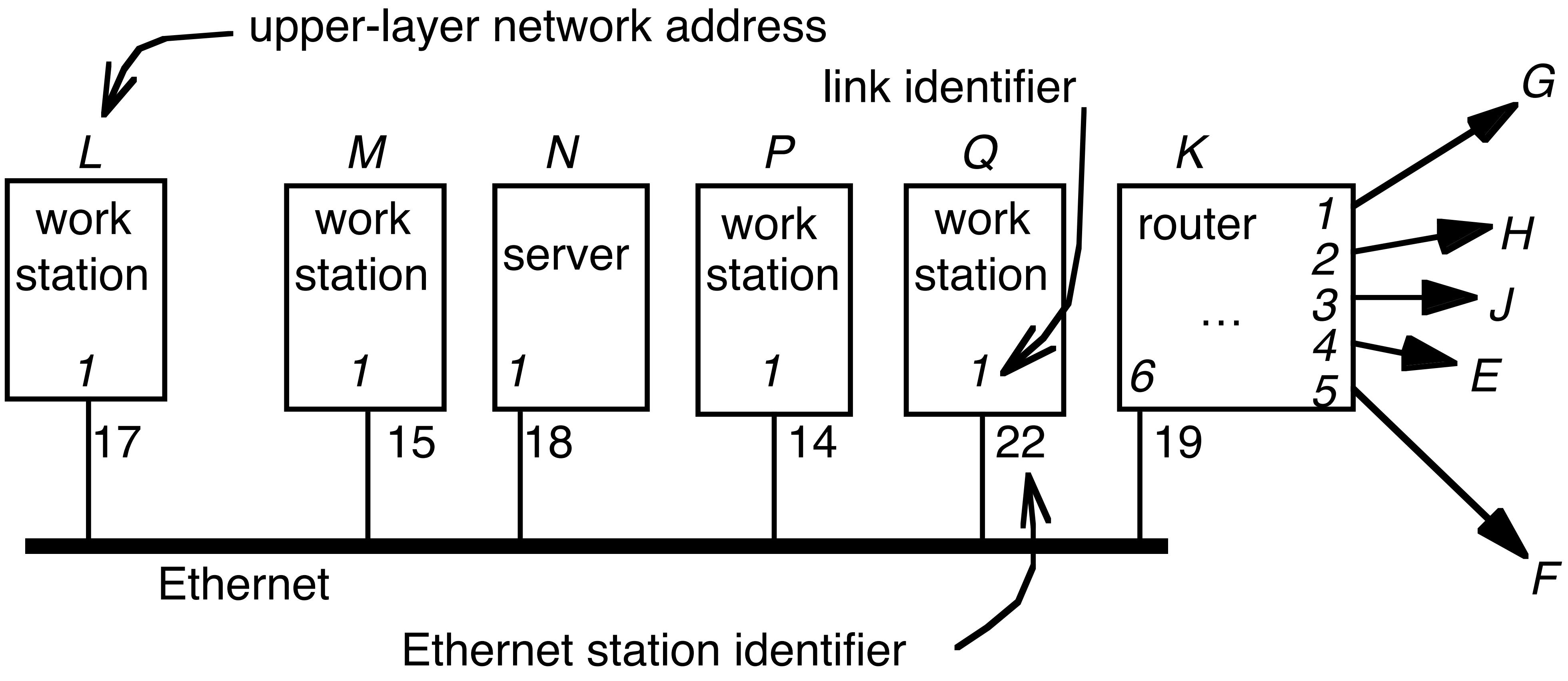

duplicate acknowledgment received
multiplicative decrease
additive increase
delay
Window size
timer expires, stop sending
slow start, again
slow start
Time

leader	destination	source	type	data	checksum
64 bits	48 bits	48 bits	16 bits	368 to 12,000 bits	32 bits

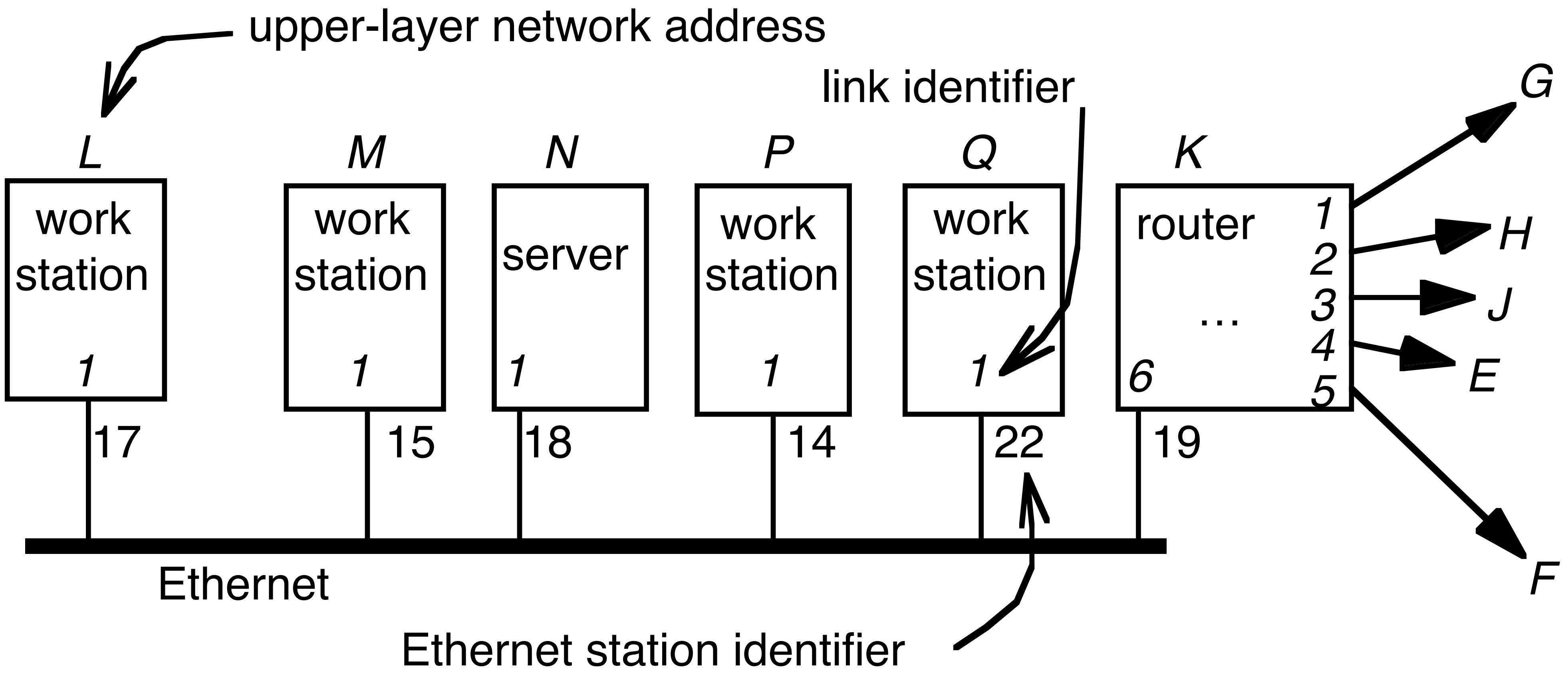

17
24
12
05
19
Station Identifier (Ethernet Address)

```
procedure ETHERNET_HANDLE (net_packet, length)
    destination ← net_packet.target_id
    if destination = my_station_id or destination = BROADCAST_ID then
        GIVE_TO_END_LAYER (net_packet.data,
                           net_packet.end_protocol,
                           net_packet.source_id)
    else
        ignore packet
```


upper-layer network address
link identifier
G
L
M
N
P
Q
K
work station
work station
server
work station
work station
router
1
2
H
3
J
...
4
E
5
1
1
1
1
1
6
17
15
18
14
22
19
F
Ethernet
Ethernet station identifier



internet address	Ethernet/ station
M	enet/15
N	enet/18
P	enet/14
Q	enet/22
K	enet/19
E	enet/19



internet address	Ethernet/ station
M	enet/15

internet address	Ethernet/ station
M	enet/15
E	enet/19