Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

# Computer Networks

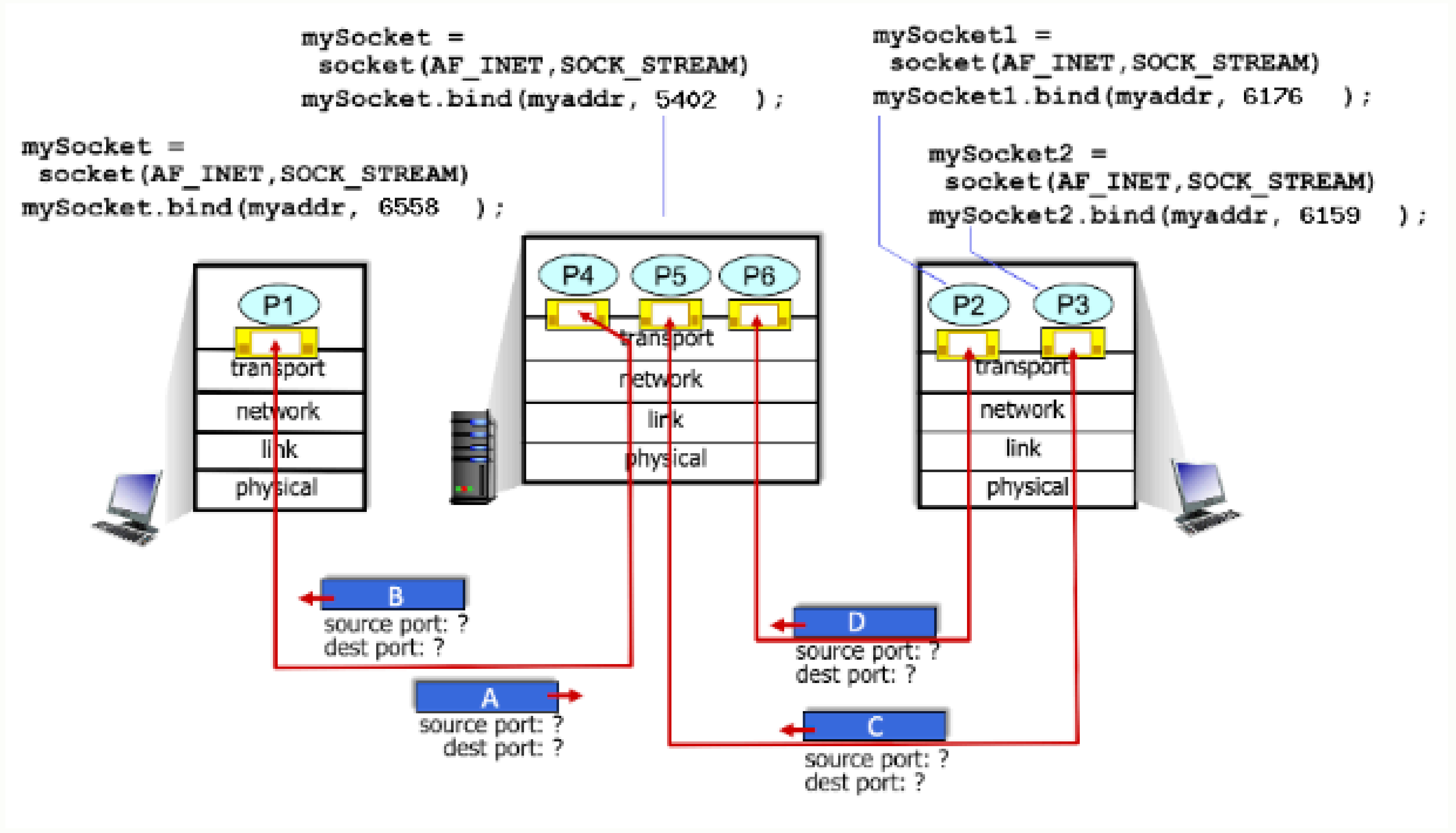
Malik Algazaeery

## Lab-4

## TCP Multiplexing and Demultiplexing

In the scenario below, the left and right TCP clients communicate with a TCP server using TCP sockets. The Python code used to create a single welcoming socket in the server is shown in the figure (the welcoming socket itself is not shown graphically); code is also shown for the client sockets as well. The three sockets shown in server were created as a result of the server accepting connection requests on this welcoming socket from the two clients (one connection from the client on the left, and two connections from the client on the right).

1. What is the source port # for packet A?
2. What is the destination port # for packet A?
3. What is the source port # for packet B?
4. What is the destination port # for packet B?
5. What is the source port # for packet C?
6. What is the destination port # for packet C?
7. What is the source port # for packet D?
8. What is the destination port # for packet D?



```
mySocket =
  socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr, 5402   );
```

```
mySocket1 =
  socket(AF_INET,SOCK_STREAM)
mySocket1.bind(myaddr, 6176   );
```

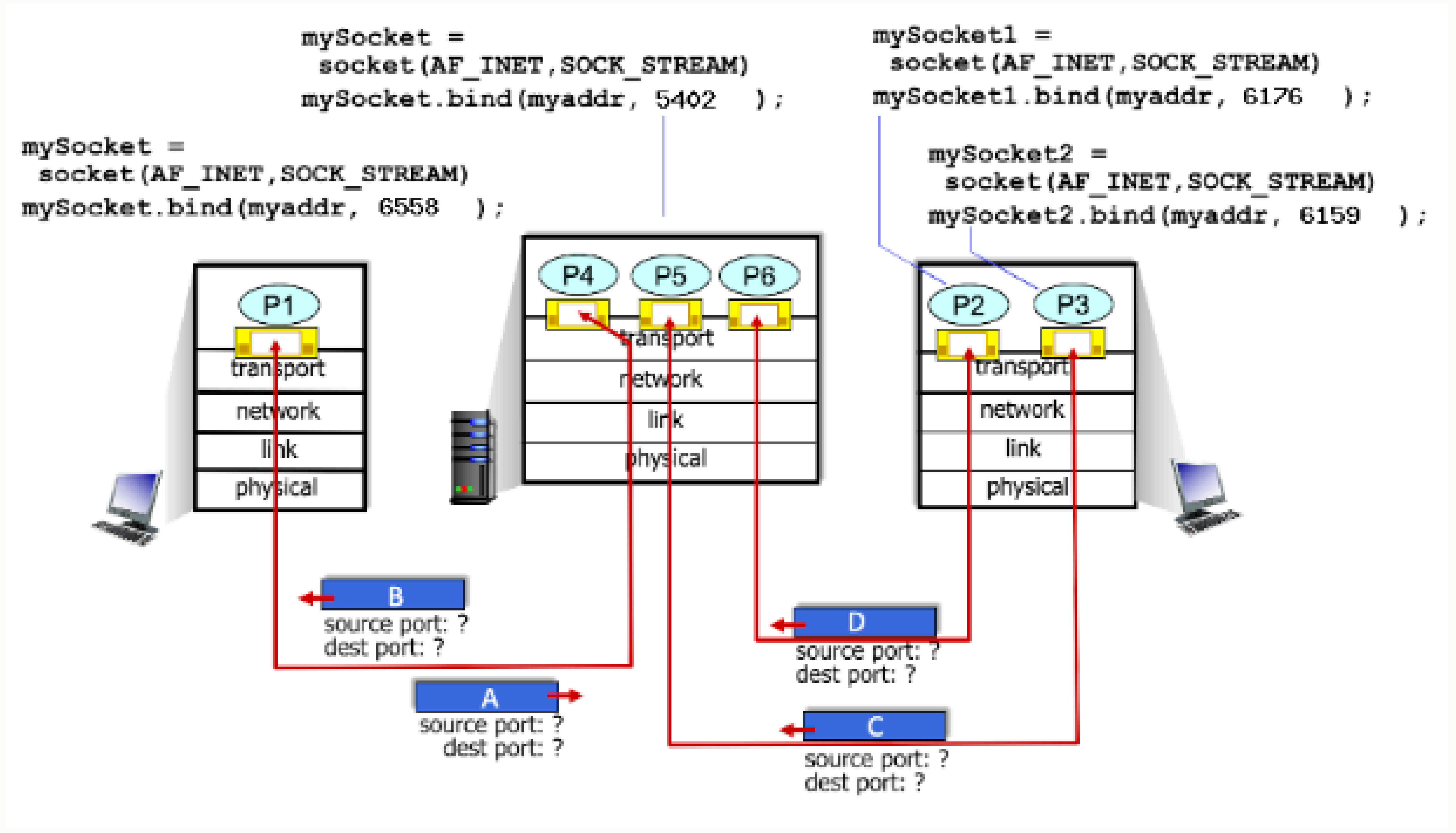
```
mySocket =
  socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr, 6558   );
```

```
mySocket2 =
  socket(AF_INET,SOCK_STREAM)
mySocket2.bind(myaddr, 6159   );
```

# TCP Multiplexing and Demultiplexing

In the scenario below, the left and right TCP clients communicate with a TCP server using TCP sockets. The Python code used to create a single welcoming socket in the server is shown in the figure (the welcoming socket itself is not shown graphically); code is also shown for the client sockets as well. The three sockets shown in server were created as a result of the server accepting connection requests on this welcoming socket from the two clients (one connection from the client on the left, and two connections from the client on the right).

1. What is the source port # for packet A? 6558
2. What is the destination port # for packet A?  5402
3. What is the source port # for packet B? 5402
4. What is the destination port # for packet B? 6558
5. What is the source port # for packet C? 6159
6. What is the destination port # for packet C? 5402
7. What is the source port # for packet D? 6176
8. What is the destination port # for packet D? 5402



```
mySocket =
 socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr, 5402  );
```

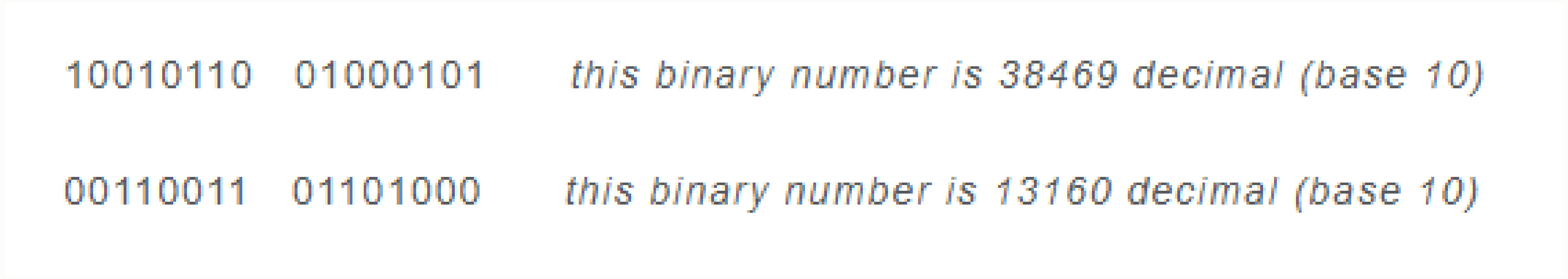
```
mySocket1 =
 socket(AF_INET,SOCK_STREAM)
mySocket1.bind(myaddr, 6176  );
```

```
mySocket =
 socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr, 6558  );
```

```
mySocket2 =
 socket(AF_INET,SOCK_STREAM)
mySocket2.bind(myaddr, 6159  );
```

## Computing an Internet checksum

Consider the two 16-bit words (shown in binary) below. Recall that to compute the Internet checksum of a set of 16-bit words, we compute the one's complement sum [1] of the two words. That is, we add the two numbers together, making sure that any carry into the 17th bit of this initial sum is added back into the 1's place of the resulting sum); we then take the one's complement of the result. Compute the Internet checksum value for these two 16-bit words:

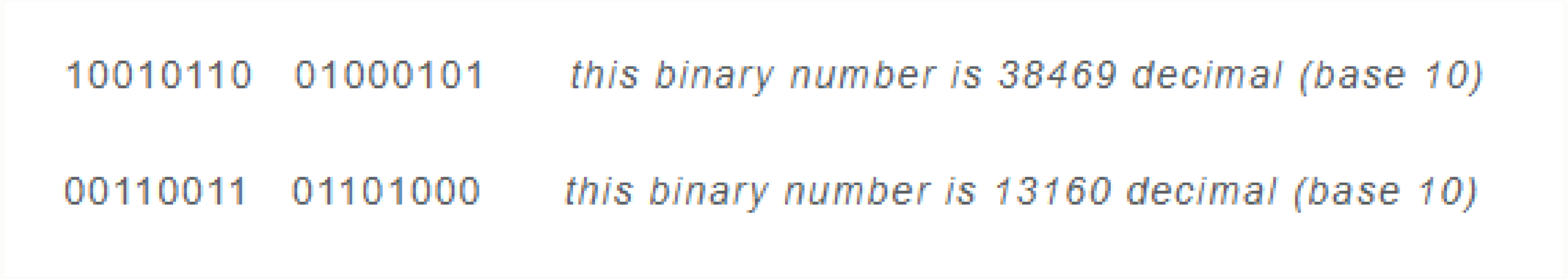


| | |
|---|---|
| 10010110 01000101 | *this binary number is 38469 decimal (base 10)* |
| 00110011 01101000 | *this binary number is 13160 decimal (base 10)* |

1. What is the sum of these two 16 bit numbers?
2. Using the sum from question 1, what is the checksum?

## Computing an Internet checksum

Consider the two 16-bit words (shown in binary) below. Recall that to compute the Internet checksum of a set of 16-bit words, we compute the one's complement sum [1] of the two words. That is, we add the two numbers together, making sure that any carry into the 17th bit of this initial sum is added back into the 1's place of the resulting sum); we then take the one's complement of the result. Compute the Internet checksum value for these two 16-bit words:



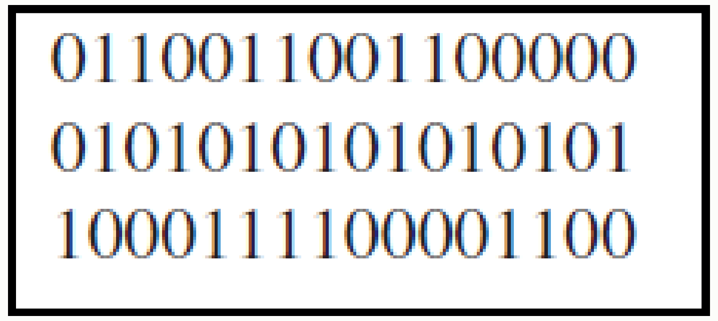
```
10010110   01000101      this binary number is 38469 decimal (base 10)

00110011   01101000      this binary number is 13160 decimal (base 10)
```

1. What is the sum of these two 16 bit numbers?
    - The sum of 10010110 01000101 and 00110011 01101000 = 11001001  10101101
2. Using the sum from question 1, what is the checksum?
    - The internet checksum is the one's complement of the sum: 11001001  10101101 = 00110110  01010010

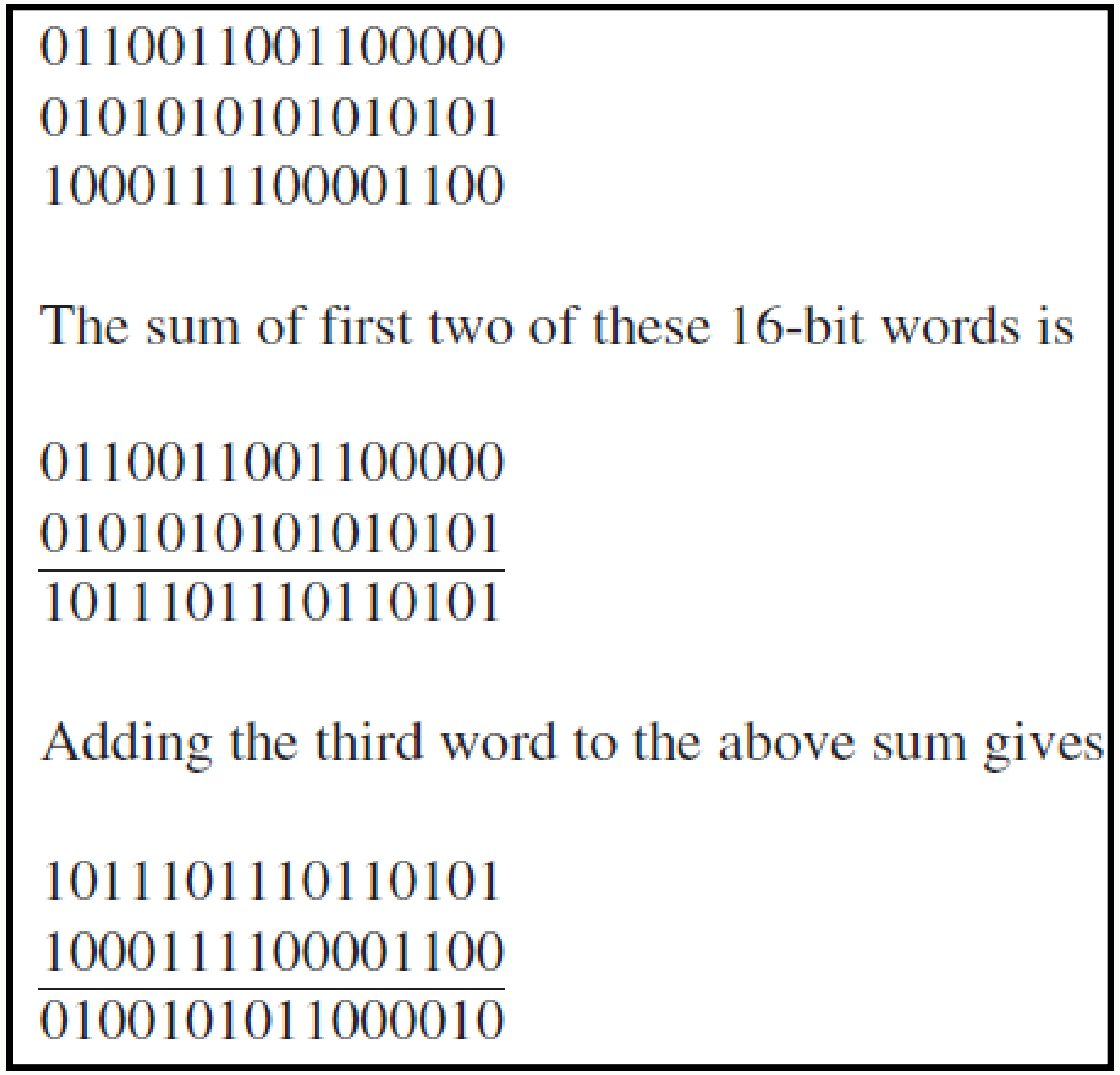
## Computing an Internet checksum 2

Consider the three 16-bit words (shown in binary) below.  Compute the Internet checksum value for these two 16-bit words.



0110011001100000
0101010101010101
1000111100001100

## Solution



0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

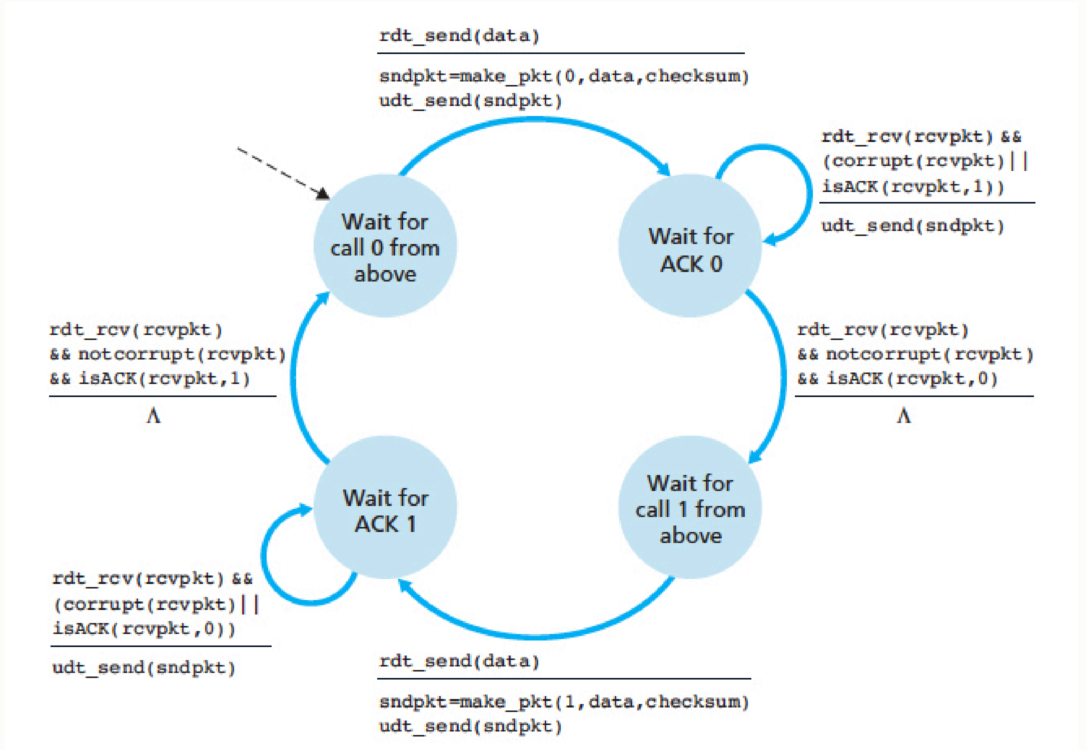
1011101110110101
1000111100001100
0100101011000010

last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus, the 1s complement of the sum 0100101011000010 is **1011010100111101**, which becomes the checksum.

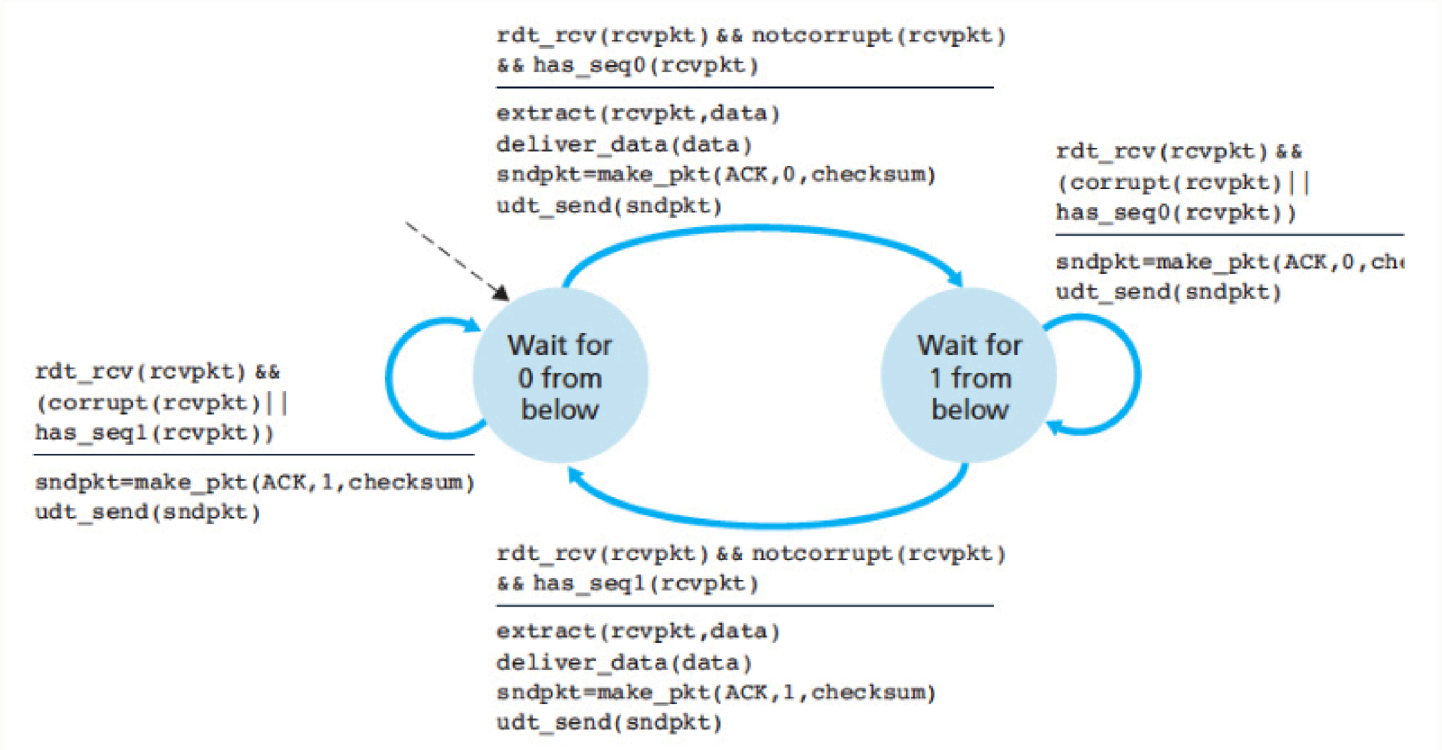
**Checksum = 1011010100111101**

## Reliable Data Transfer: rdt2.2 (sender and receiver actions)

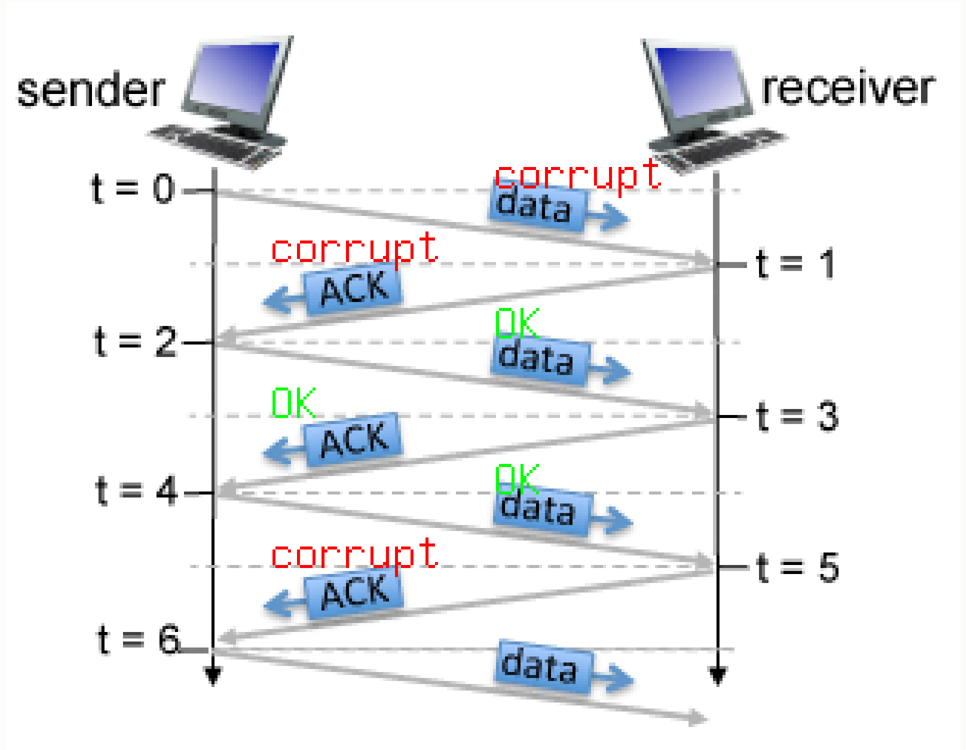
Consider the rdt2.2 protocol from the text (pages 209-212). The FSMs for the sender and receiver are shown below. Suppose that the channel connecting the sender and receiver can corrupt but not lose or reorder packets. Now consider the figure below, which shows four data packets and three corresponding ACKs being exchanged between an rdt 2.2 sender and receiver. The actual corruption or successful transmission/reception of a packet is indicated by the corrupt and OK labels, respectively, shown above the packets in the figure below.



rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
Wait for ACK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)



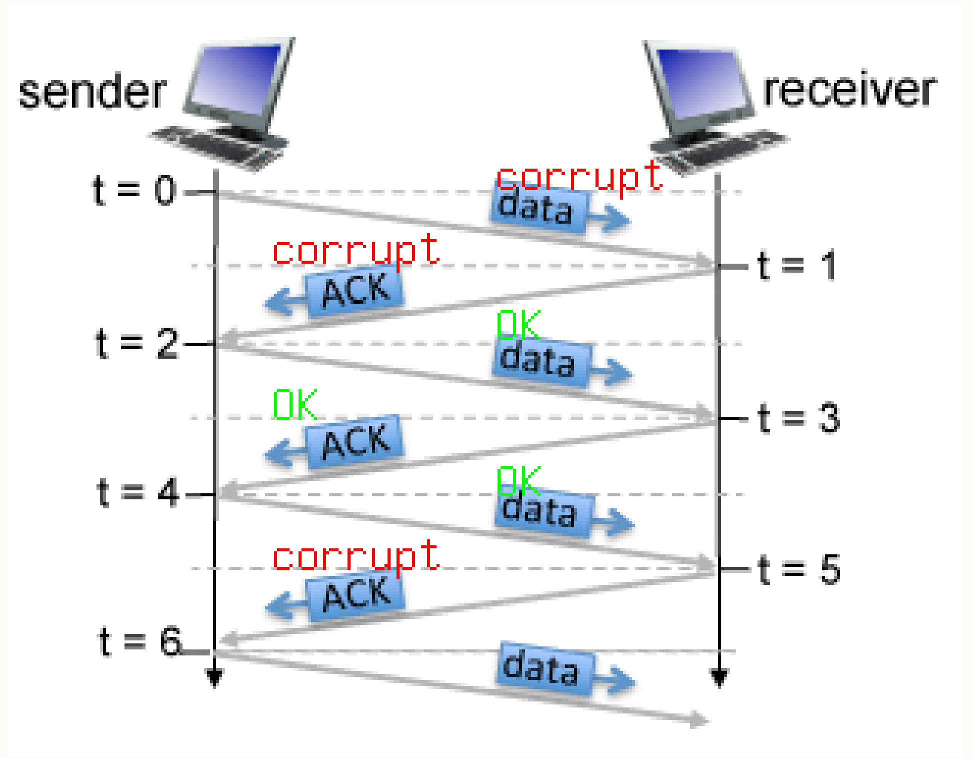
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,ch
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq1(rcvpkt))
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)

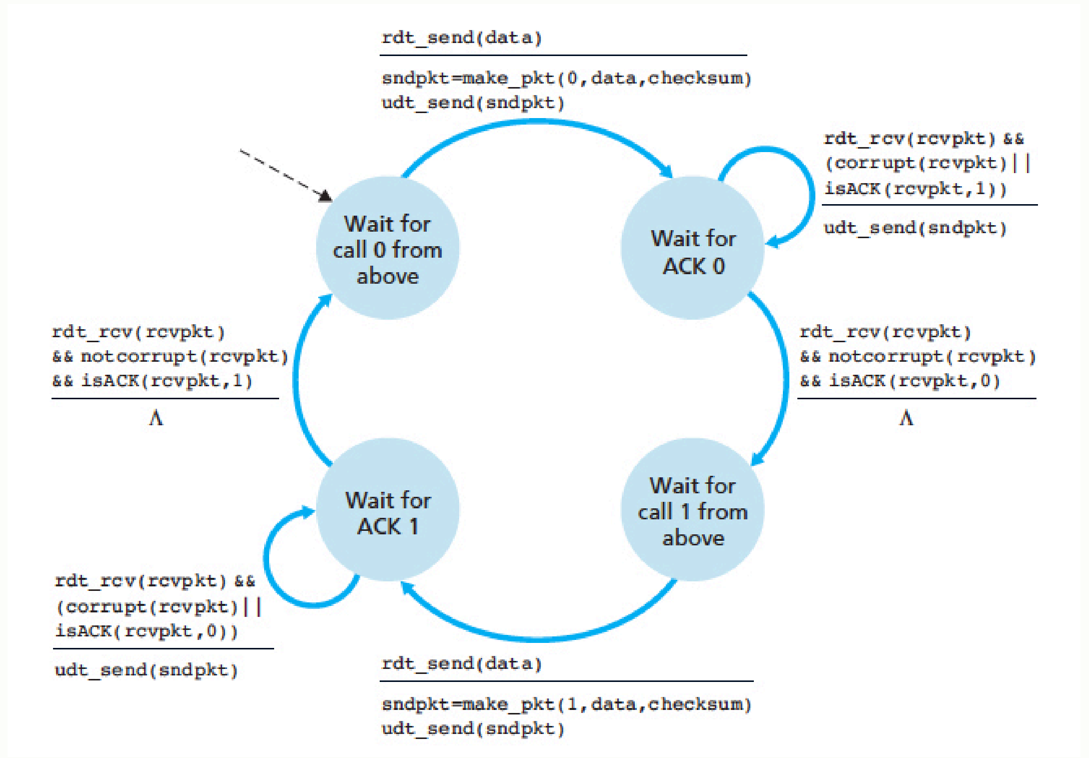


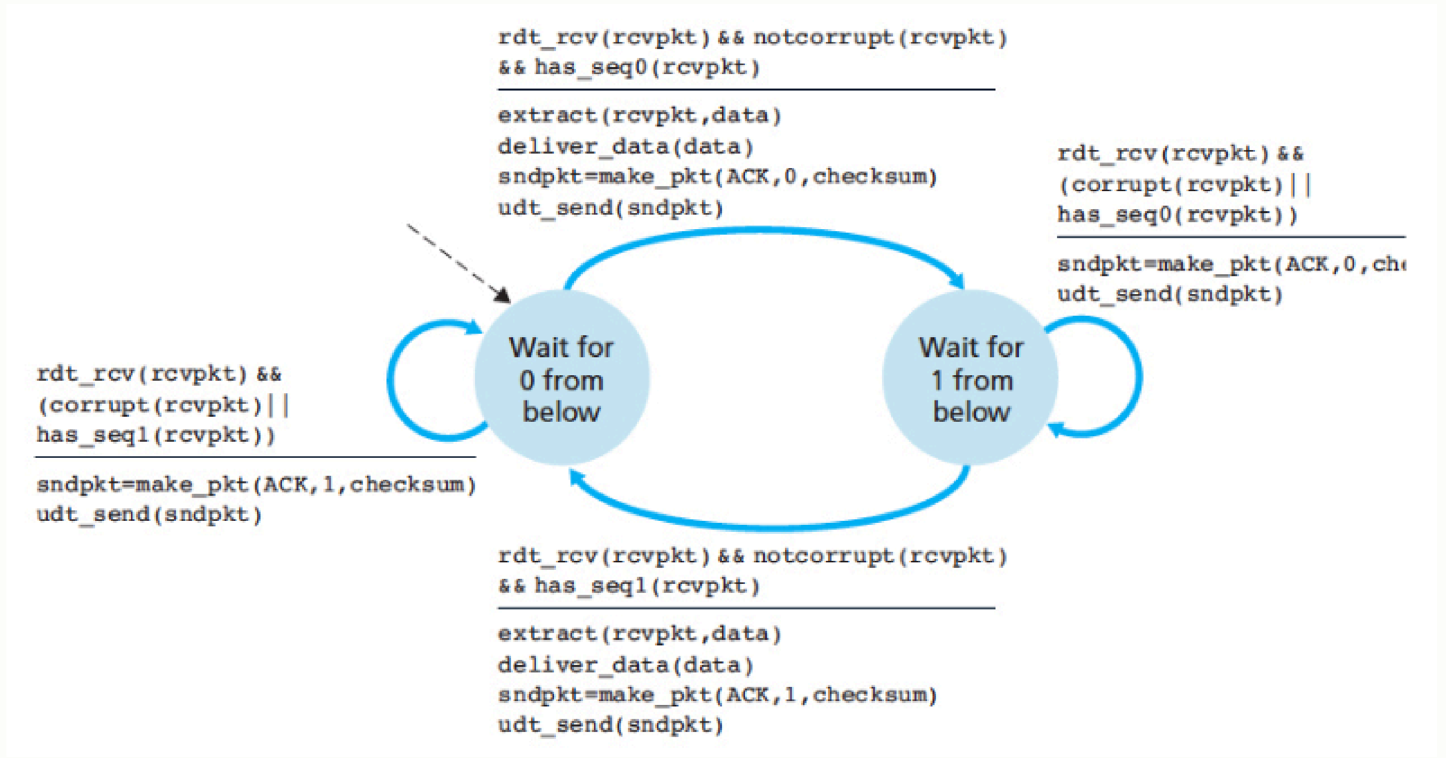
1. At time t=0, what is the sender state? 2. At time t=0, what is the receiver state? 3. At time t=0, what is the sequence/ack # of the packet? 4. At time t=1, what is the sender state? 5. At time t=1, what is the receiver state? 6. At time t=1, what is the sequence/ack # of the packet? 7. At time t=2, what is the sender state? 8. At time t=2, what is the receiver state? 9. At time t=2, what is the sequence/ack # of the packet? 10. At time t=3, what is the sender state? 11. At time t=3, what is the receiver state? 12. At time t=3, what is the sequence/ack # of the packet? 13. How many times is the payload of the received packet passed up to the higher layer?

Solution

1. At time t=0, the sender state is: Wait for ACK 0
2. At time t=0, the receiver state is: Wait for 0 from below
3. At time t=0, the sequence # is: 0
4. At time t=1, the sender state is: Wait for ACK 0
5. At time t=1, the receiver state is: Wait for 0 from below
6. At time t=1, the ACK # is: 1
7. At time t=2, the sender state is: Wait for ACK 0
8. At time t=2, the receiver state is: Wait for 0 from below
9. At time t=2, the sequence # is: 0
10. At time t=3, the sender state is: Wait for ACK 0
11. At time t=3, the receiver state is: Wait for 1 from below
12. At time t=3, the ACK # is: 0
13. 2 packets were passed up to the higher layer by the receiver.



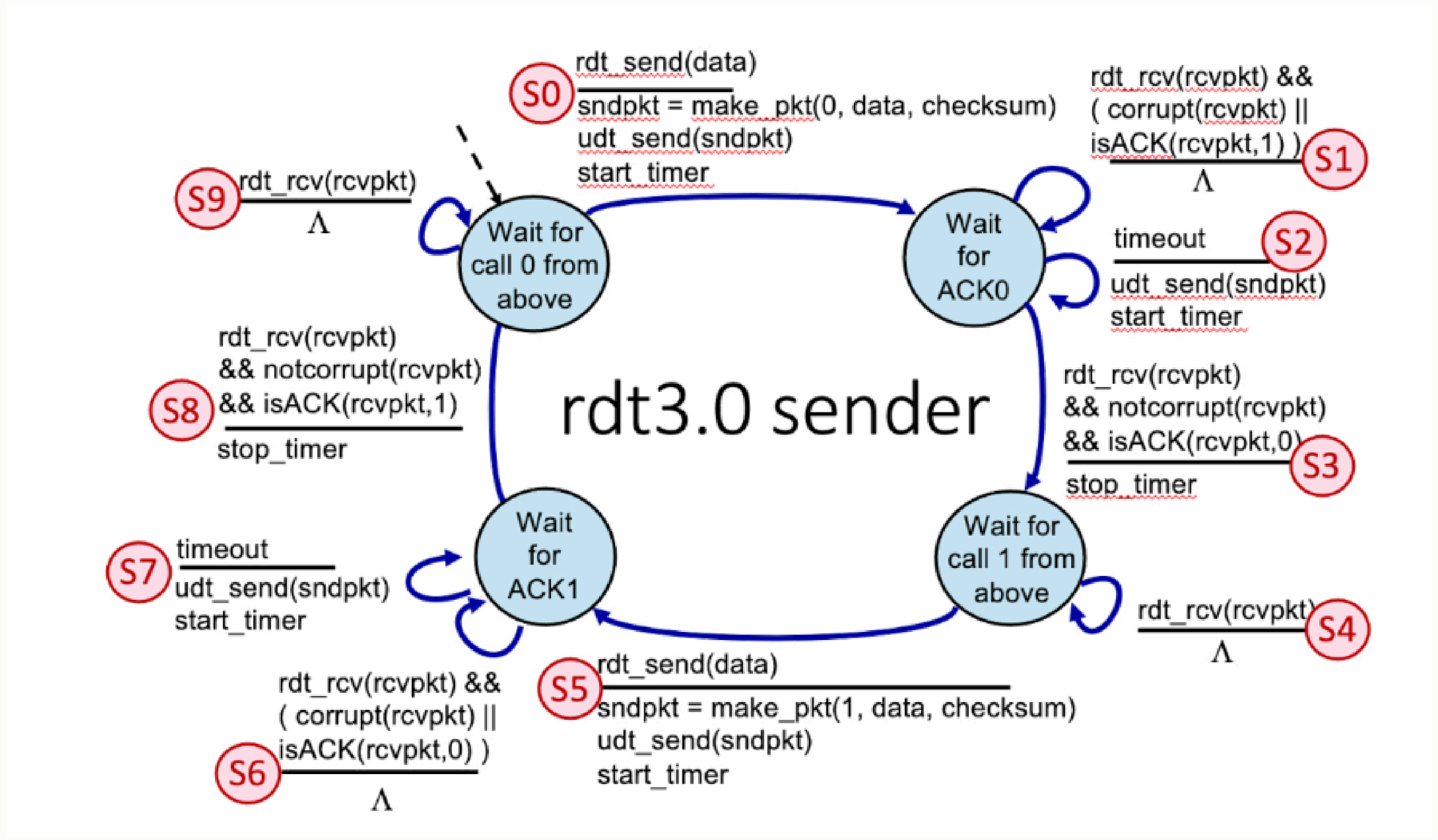

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
Wait for ACK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)

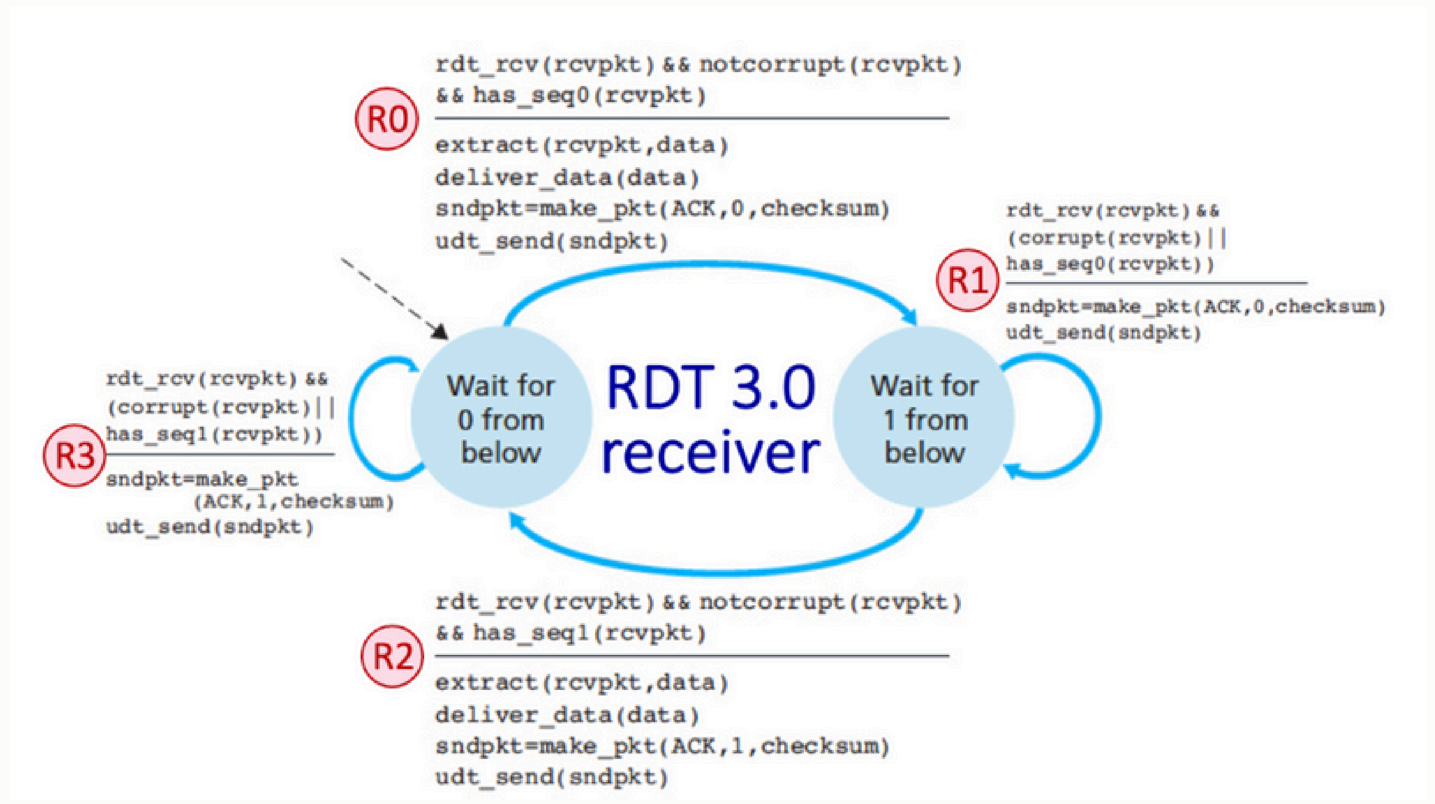

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,ch
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq1(rcvpkt))
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)

## Reliable Data Transfer: RDT 3.0

Consider the RDT 3.0 protocol, for reliably communicating data from a sender to receiver over a channel that can lose or corrupt packets in either direction, and when the maximum delay from sender to receiver and back is not known. The FSMs for the sender and receiver are shown below, with their transitions labeled as SX and RY, respectively.
Now let's consider the sequence of sender and receiver transitions that would happen when one or more of the following complications occur: a packet (data or ACK) is lost, a timer times out (prematurely or not), or a message is corrupted. One or more of these events has occurred to produce the sequence of transitions below. In the sequence below, one transition has been omitted and replaced with a "*".

1. Transition Sequence: S0, R3, *, S2, R0, S3, S5, R2, S6, S7, R3, S8.
2. Transition Sequence: S0, R0, S2, R1, S3, S5, S7, *, S6, S7, R3, S6, S7, R3, S6, S7, R3, S6, S7, R3, S8.
3. Transition Sequence: S0, R0, S2, *, S3, S4, S5, R1, S6, S7, R2, S8.
4. Transition Sequence: S0, *, S1, S2, R1, S3, S5, S7, R2, S8.
5. Transition Sequence: S0, R0, S1, S2, R1, *, S2, R1, S3, S5, S7, R2, S6, S7, R3, S6, S7, R3, S8.

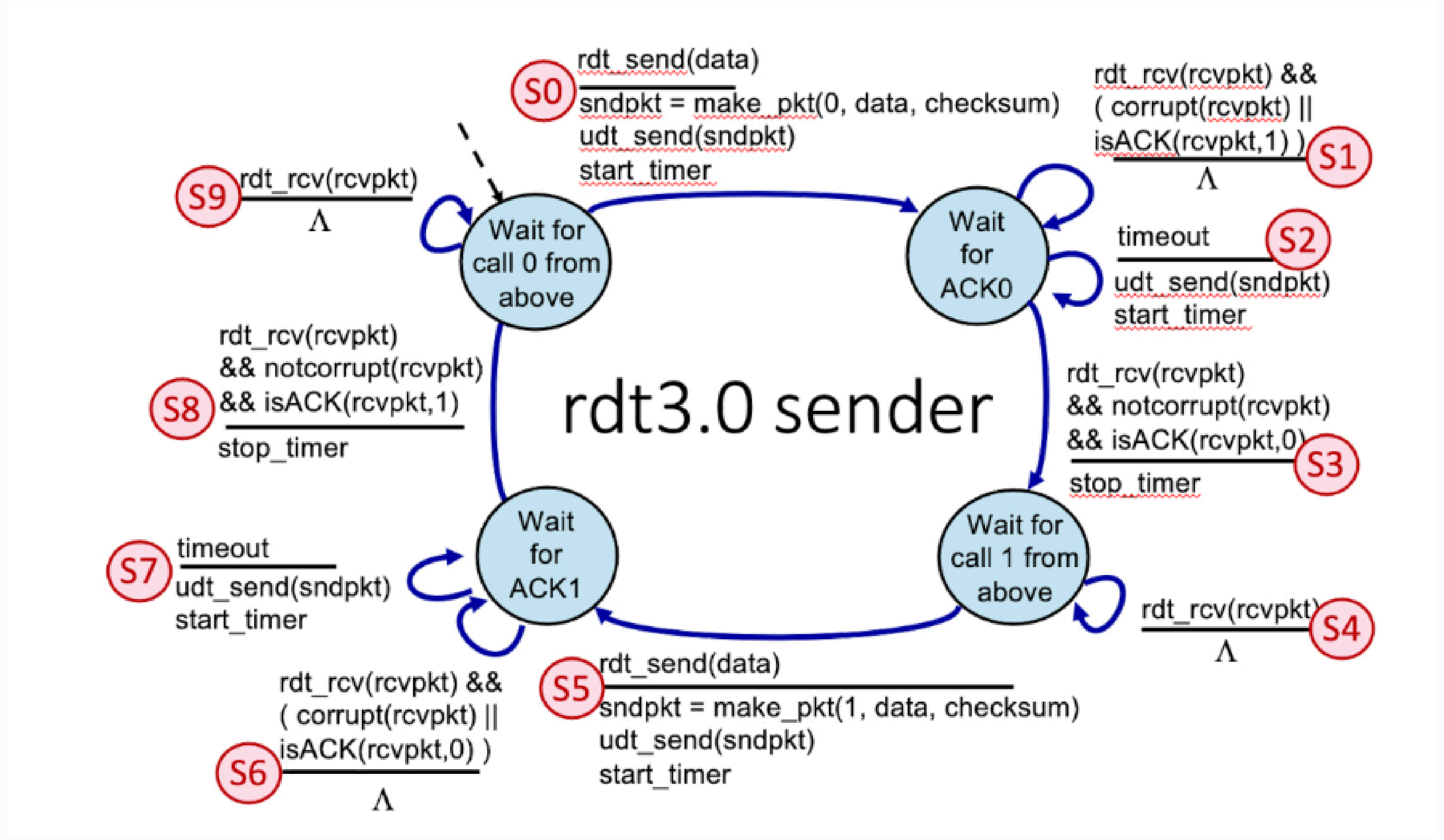




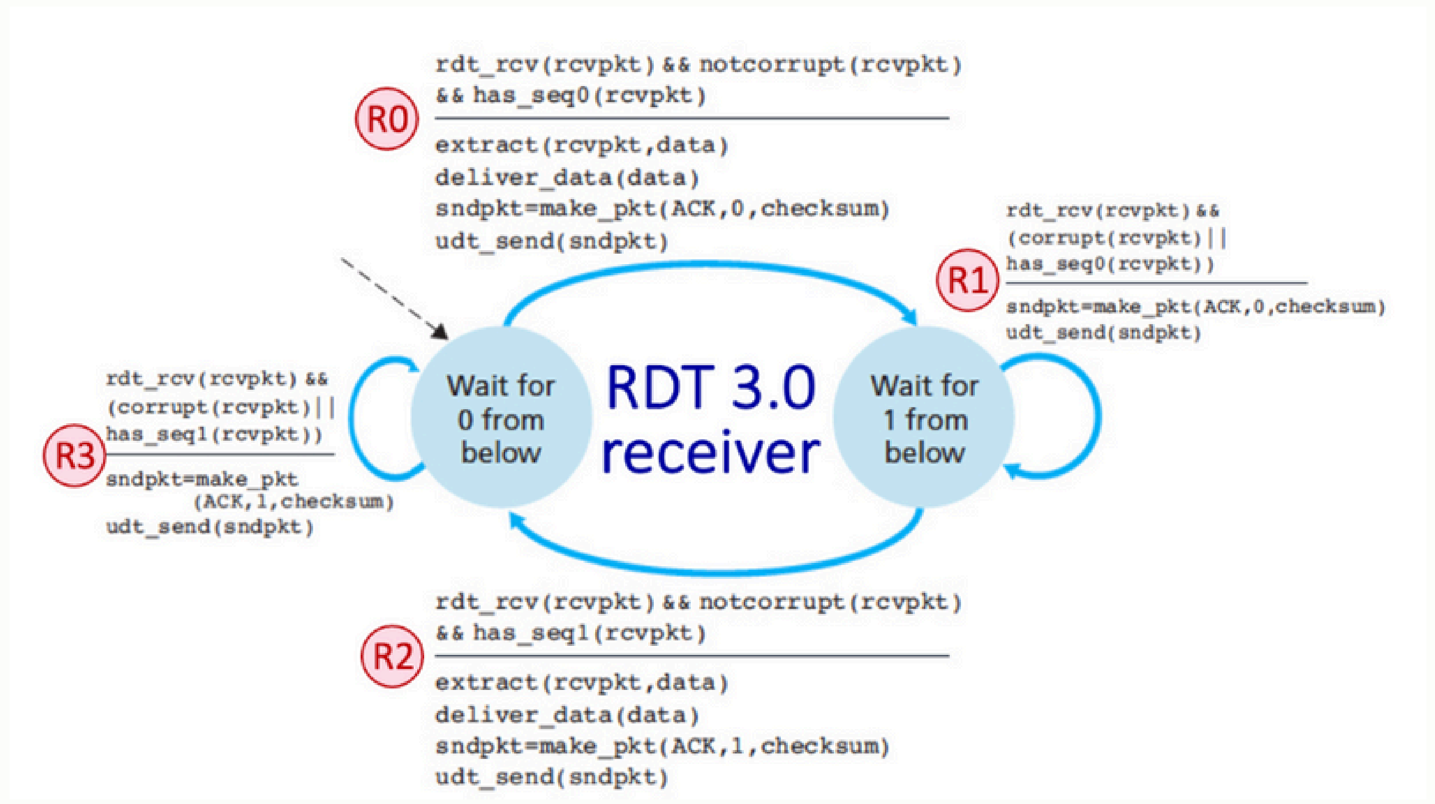
## Reliable Data Transfer: RDT 3.0

Consider the RDT 3.0 protocol, for reliably communicating data from a sender to receiver over a channel that can lose or corrupt packets in either direction, and when the maximum delay from sender to receiver and back is not known. The FSMs for the sender and receiver are shown below, with their transitions labeled as SX and RY, respectively.
Now let’s consider the sequence of sender and receiver transitions that would happen when one or more of the following complications occur: a packet (data or ACK) is lost, a timer times out (prematurely or not), or a message is corrupted. One or more of these events has occurred to produce the sequence of transitions below. In the sequence below, one transition has been omitted and replaced with a "*".

1. Transition Sequence: S0, R3, *, S2, R0, S3, S5, R2, S6, S7, R3, S8. The missing transition is: S1
2. Transition Sequence: S0, R0, S2, R1, S3, S5, S7, *, S6, S7, R3, S6, S7, R3, S6, S7, R3, S6, S7, R3, S8. The missing transition is: R2
3. Transition Sequence: S0, R0, S2, *, S3, S4, S5, R1, S6, S7, R2, S8. The missing transition is: R1
4. Transition Sequence: S0, *, S1, S2, R1, S3, S5, S7, R2, S8. The missing transition is: R0
5. Transition Sequence: S0, R0, S1, S2, R1, *, S2, R1, S3, S5, S7, R2, S6, S7, R3, S6, S7, R3, S8. The missing transition is: S1



S0
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
S1
rdt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,1) )
Λ
S9
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK0
S2
timeout
udt_send(sndpkt)
start_timer
S8
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt3.0 sender
S3
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
S7
timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
S4
rdt_rcv(rcvpkt)
Λ
S6
rdt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,0) )
Λ
S5
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer



R0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
R1
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt)||
has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
R3
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt)||
has_seq1(rcvpkt))
sndpkt=make_pkt
(ACK,1,checksum)
udt_send(sndpkt)
Wait for 0 from below
RDT 3.0 receiver
Wait for 1 from below
R2
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)