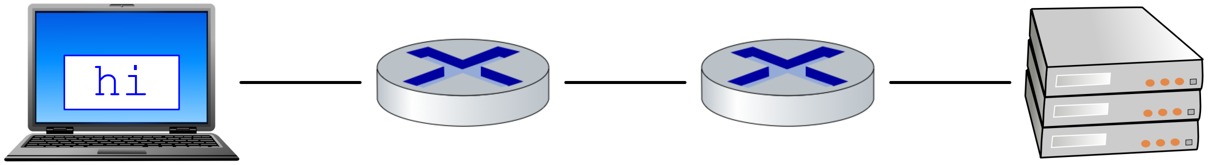
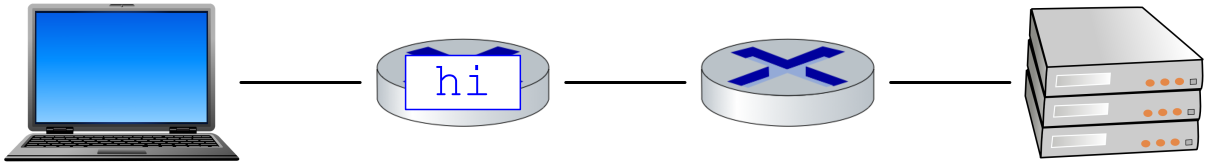
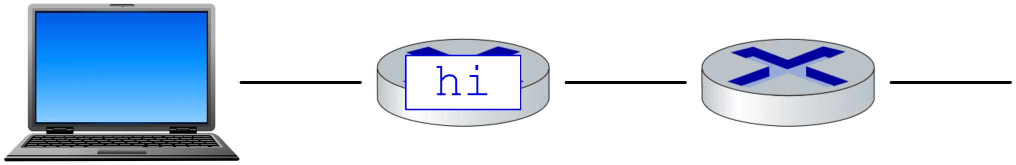
Reliable Data Transfer



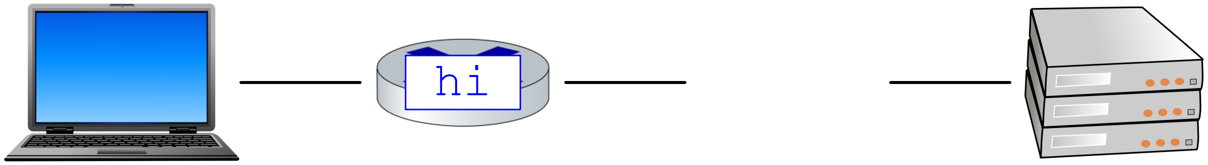
Reliable Data Transfer



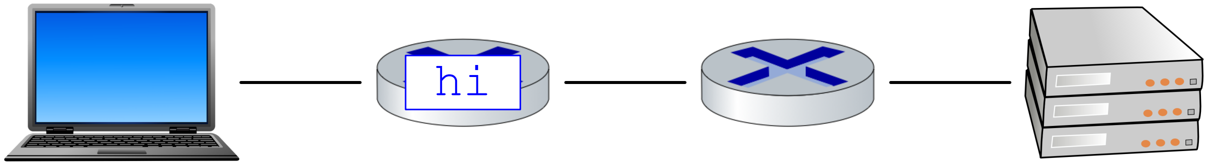
Reliable Data Transfer



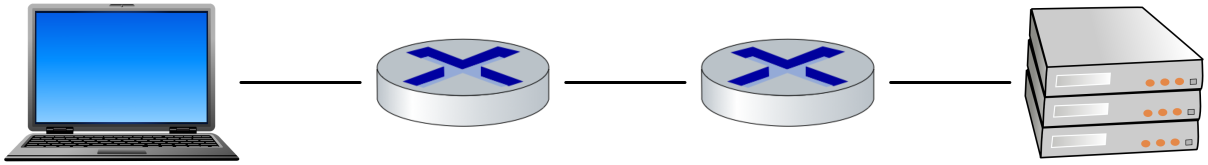
Reliable Data Transfer



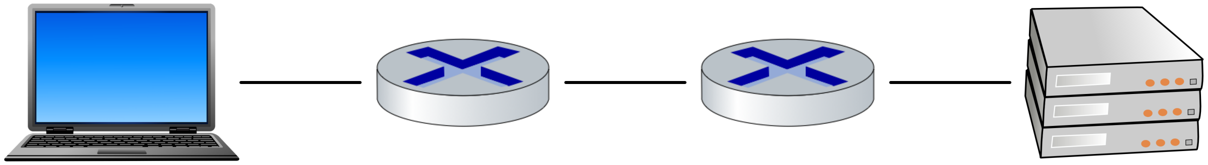
Reliable Data Transfer



Reliable Data Transfer

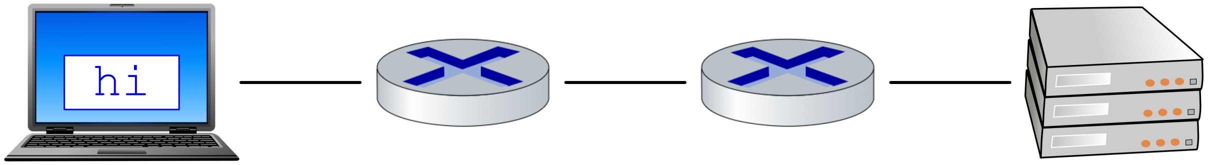


Reliable Data Transfer



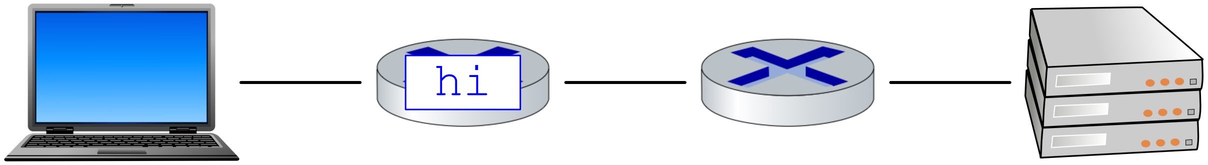
Focus only on packet-delivery problems

Reliable Data Transfer



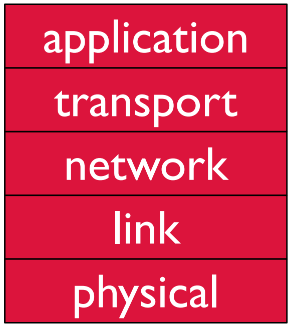
Focus only on packet-delivery problems

Reliable Data Transfer

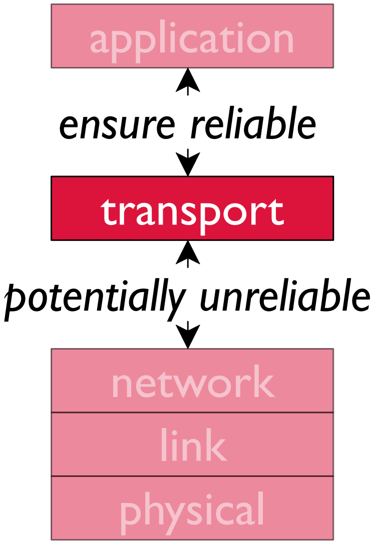


Focus only on packet-delivery problems

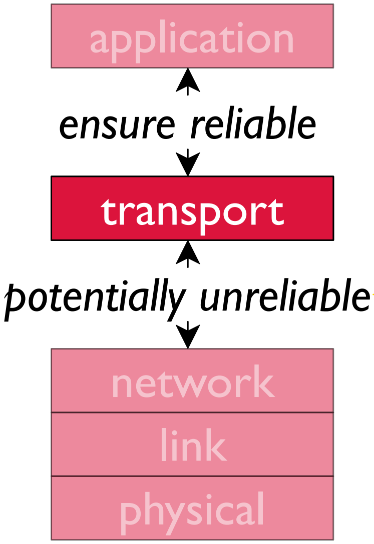
Reliable Data Transfer



Reliable Data Transfer



Reliable Data Transfer



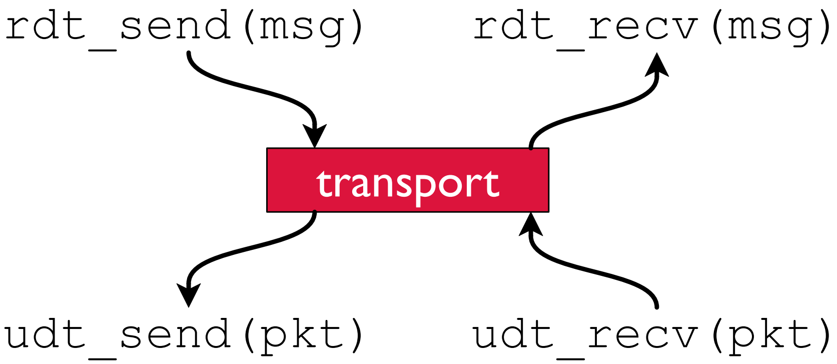
lost packets

reordered packets

duplicated packets

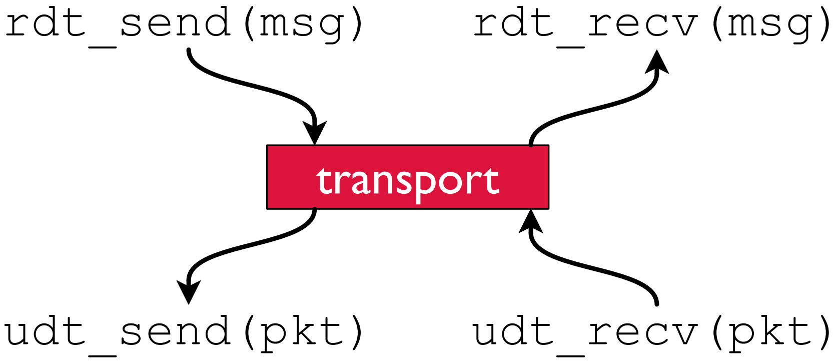
corrupted packets

Reliable Data Transfer



Reliable Data Transfer

“reliable data transfer”



rdt_send(msg)

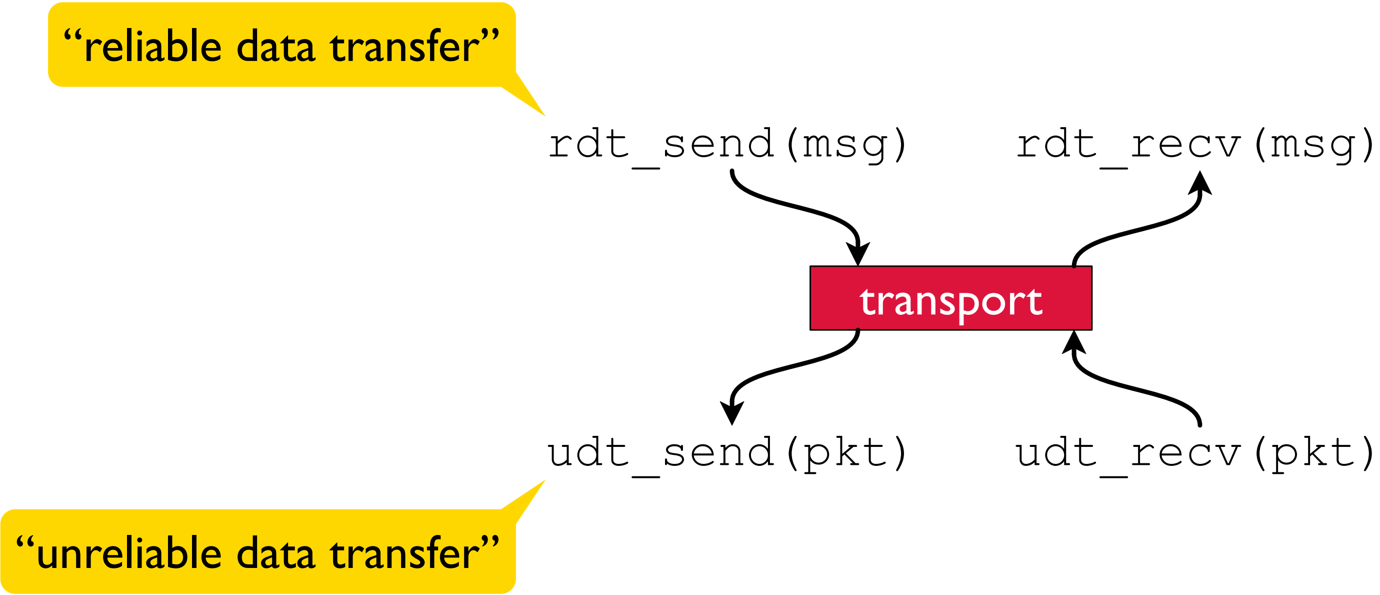
rdt_recv(msg)

transport

udt_send(pkt)

udt_recv(pkt)

Reliable Data Transfer



“reliable data transfer”

rdt_send(msg)

rdt_recv(msg)

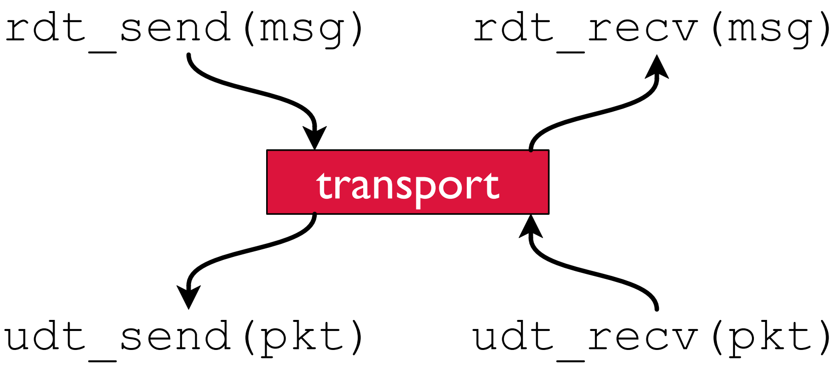
transport

udt_send(pkt)

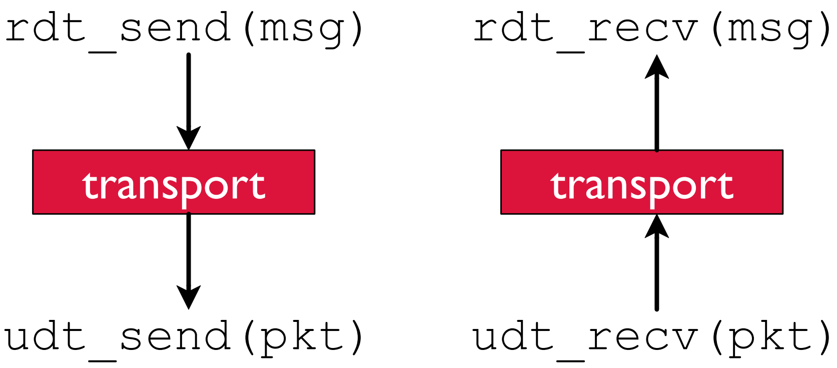
udt_recv(pkt)

“unreliable data transfer”

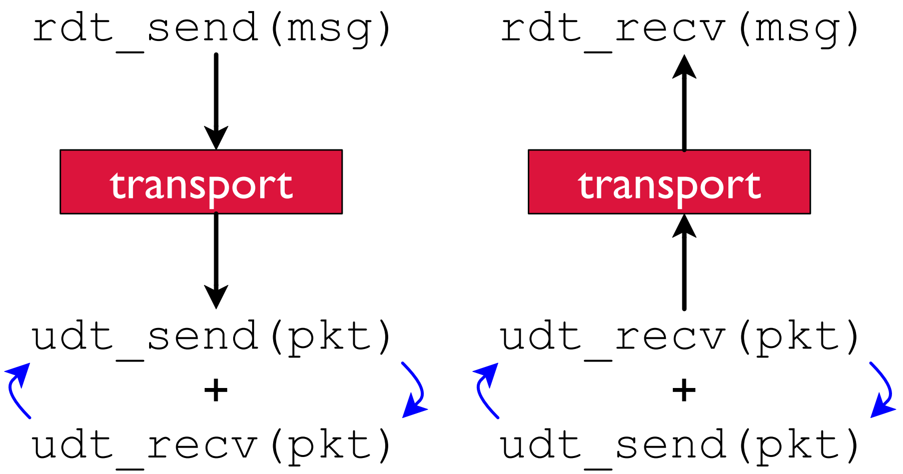
Reliable Data Transfer



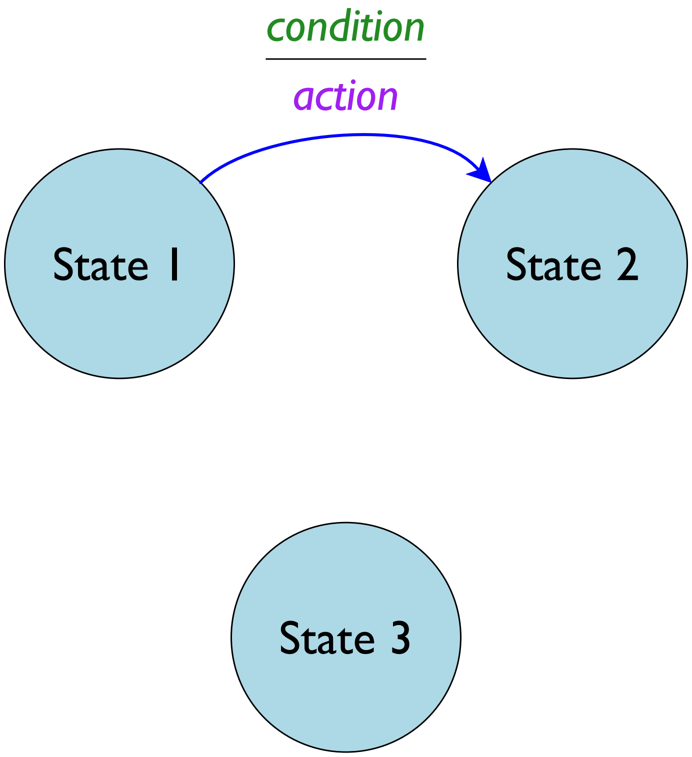
Reliable Data Transfer



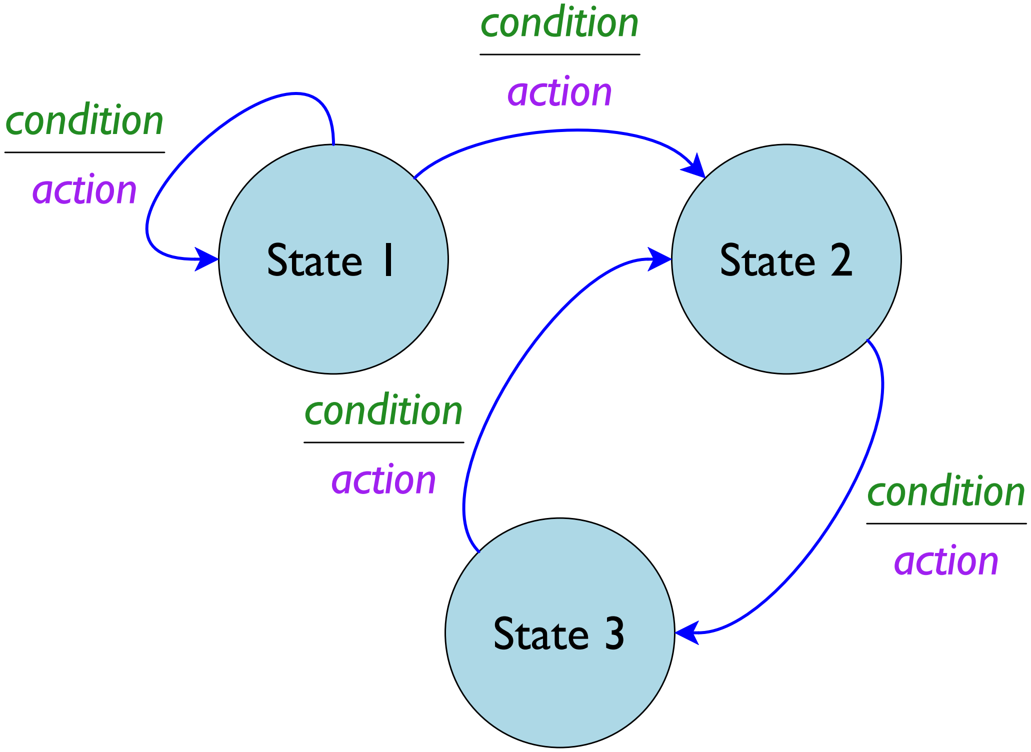
Reliable Data Transfer



State Machines

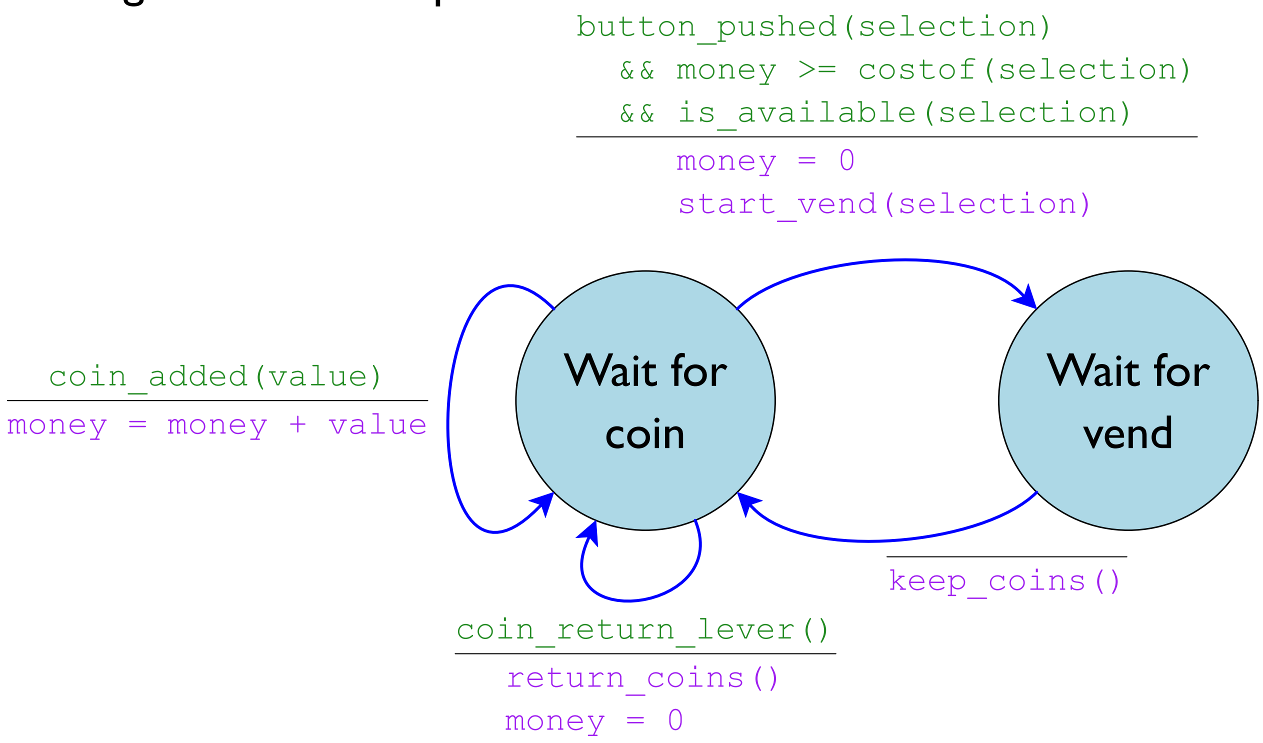


State Machines



State Machines

Vending machine example



```
button_pushed(selection)
  && money >= costof(selection)
  && is_available(selection)
-----------------------------
    money = 0
    start_vend(selection)
```

Wait for coin

Wait for vend

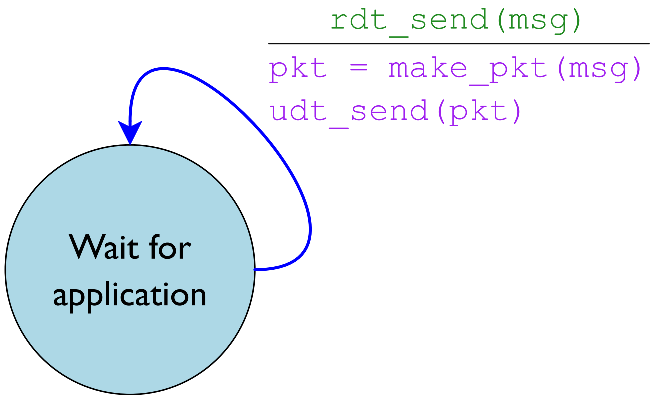
```
coin_added(value)
---------------------
money = money + value
```

```
coin_return_lever()
-------------------
  return_coins()
  money = 0
```

```
------------
keep_coins()
```

Assuming Reliable `udt_send` and `udt_recv`

sending host



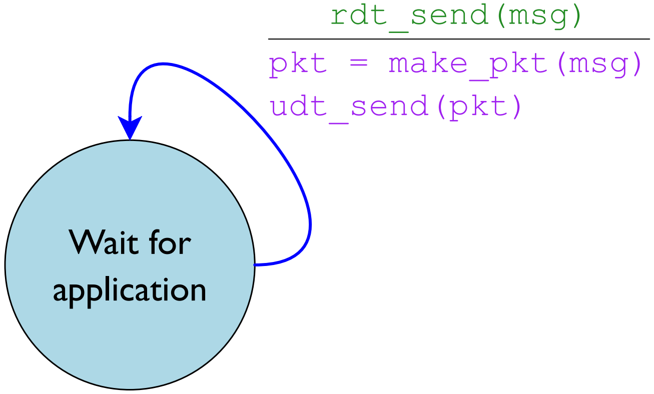
rdt_send(msg)

pkt = make_pkt(msg)
udt_send(pkt)

Wait for application

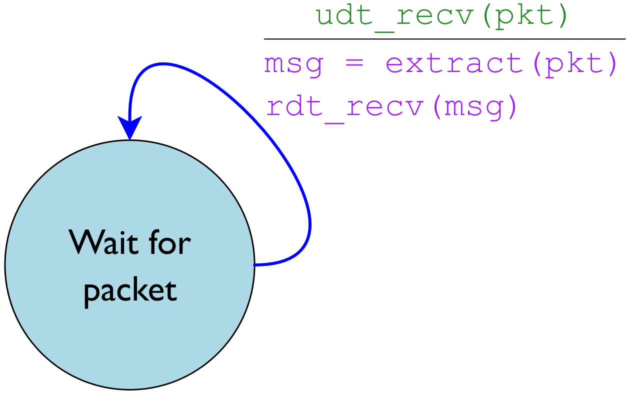
Assuming Reliable `udt_send` and `udt_recv`

sending host



```
rdt_send(msg)
________________
pkt = make_pkt(msg)
udt_send(pkt)
```

receiving host

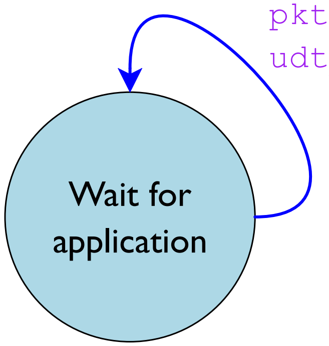


```
udt_recv(pkt)
________________
msg = extract(pkt)
rdt_recv(msg)
```

Assuming Reliable `udt_send` and `udt_recv`

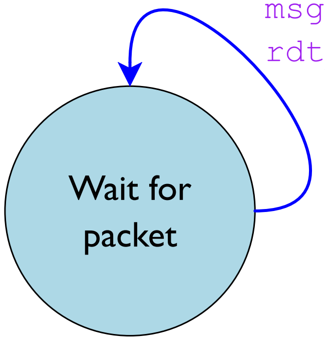
What if `pkt` is corrupted?

sending host



```
rdt_send(msg)
----------------------
pkt = make_pkt(msg)
udt_send(pkt)
```

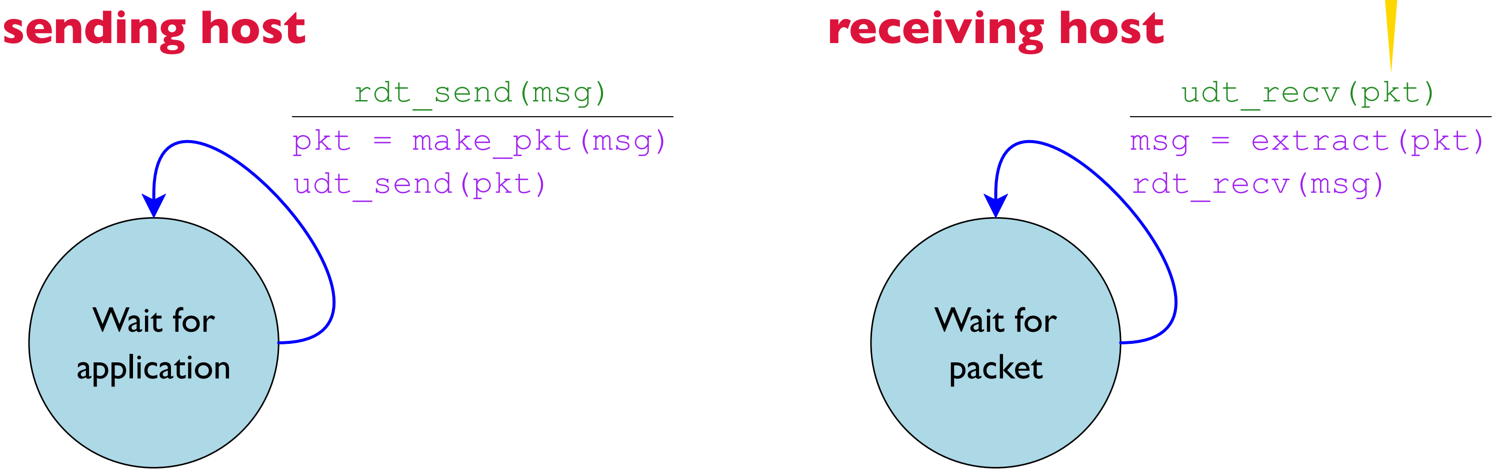
receiving host



```
udt_recv(pkt)
----------------------
msg = extract(pkt)
rdt_recv(msg)
```

Assuming Reliable `udt_send` and `udt_recv`

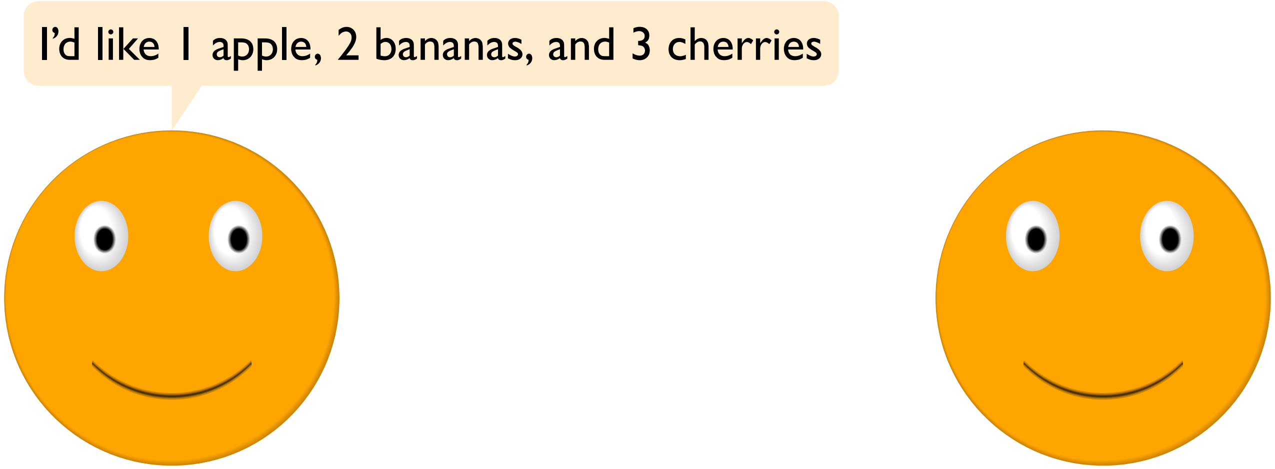
What if `pkt` is corrupted?



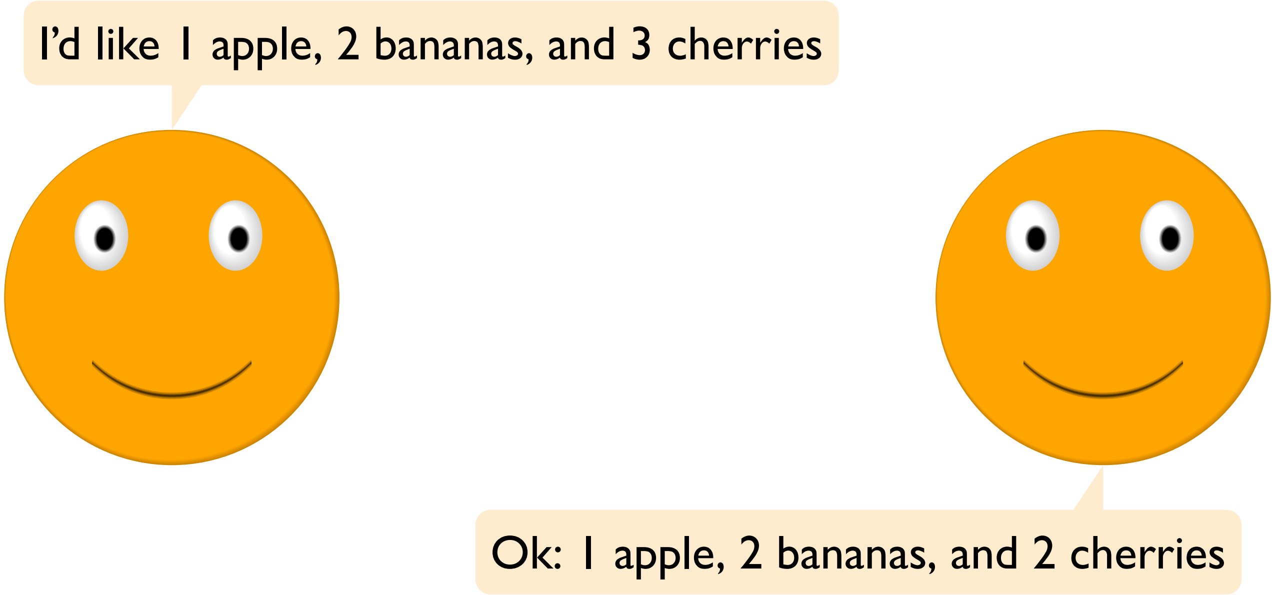
Send twice and check as the same?

Checksum

Checksum



Checksum



I’d like 1 apple, 2 bananas, and 3 cherries

Ok: 1 apple, 2 bananas, and 2 cherries

Checksum

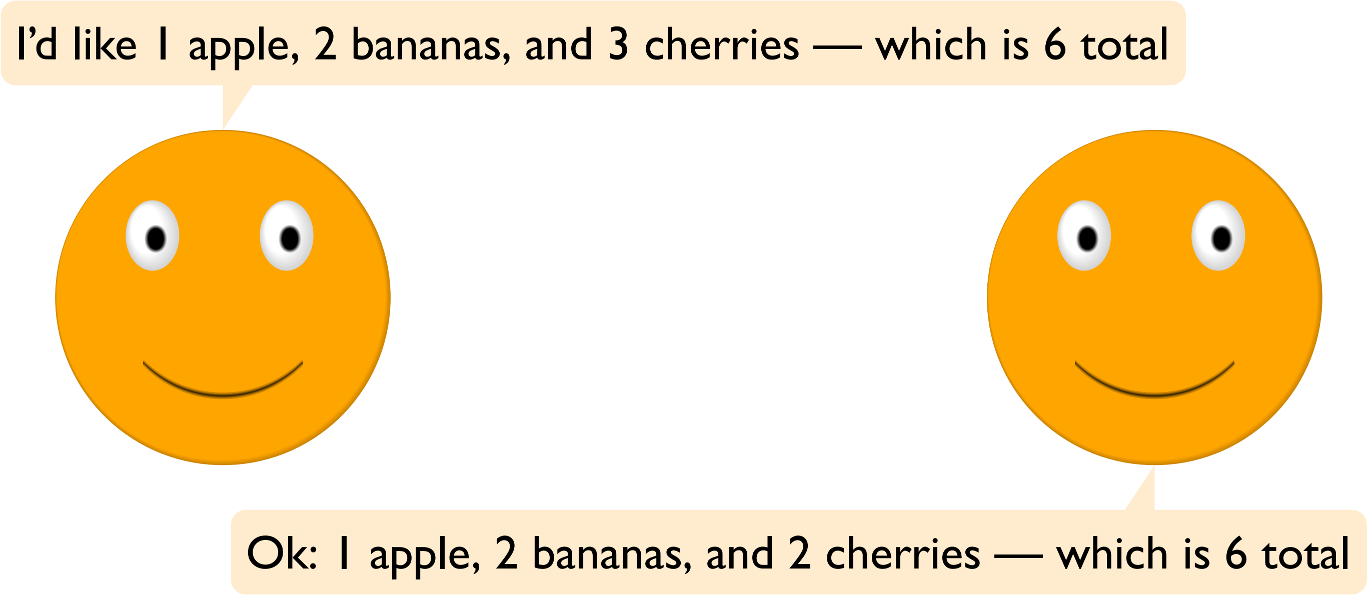
I’d like 1 apple, 2 bananas, and 3 cherries

Ok: 1 apple, 2 bananas, and 2 cherries

Checksum

I’d like 1 apple, 2 bananas, and 3 cherries — which is 6 total

Checksum



I’d like 1 apple, 2 bananas, and 3 cherries — which is 6 total

Ok: 1 apple, 2 bananas, and 2 cherries — which is 6 total

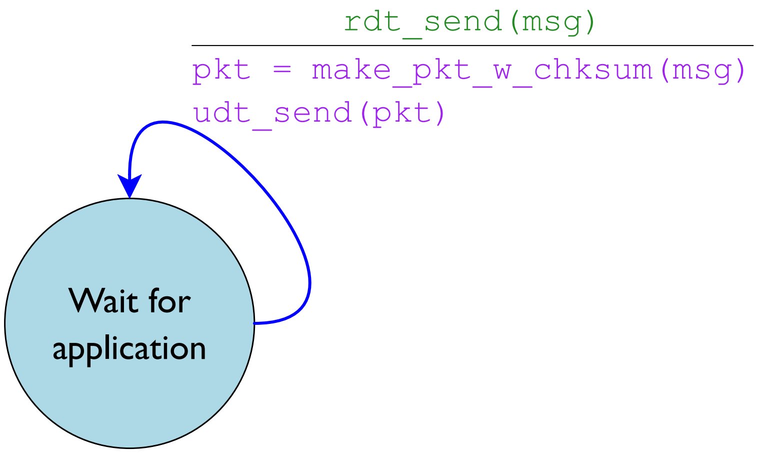
Checksum

I’d like 1 apple, 2 bananas, and 3 cherries — which is 6 total

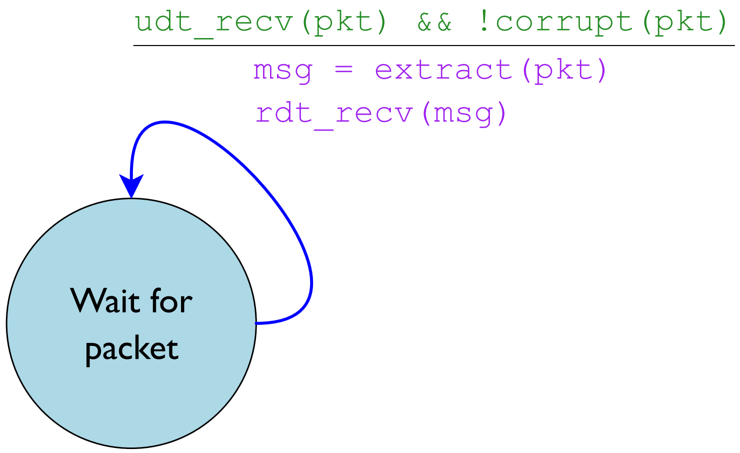
Ok: 1 apple, 2 bananas, and 2 cherries — which is 6 total

To deal with lots of numbers, just keep low bits of the sum

Using a Checksum

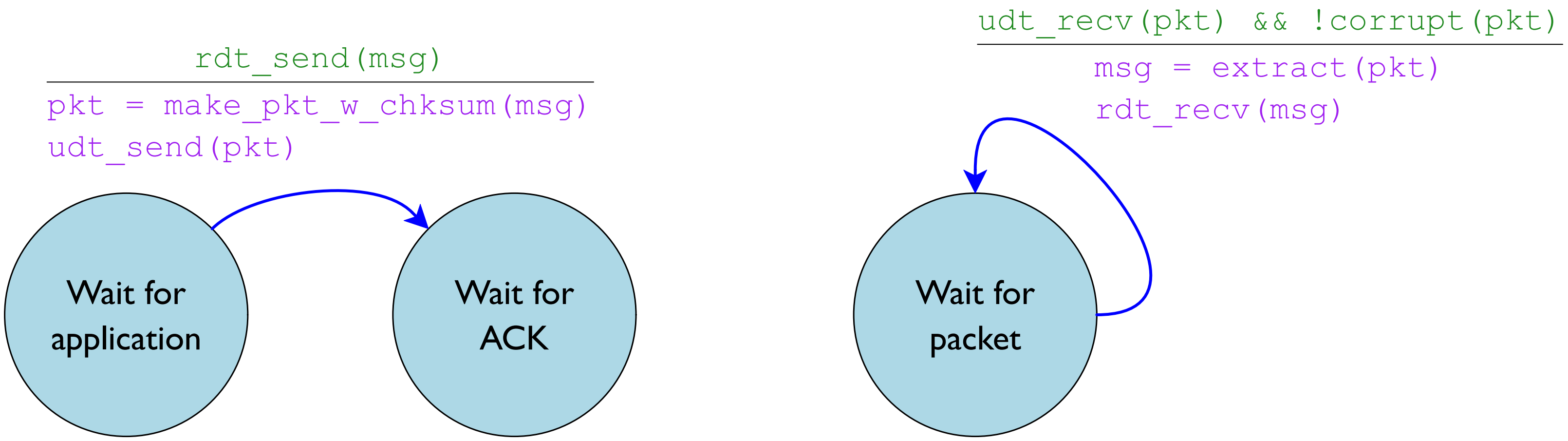


```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg)
udt_send(pkt)
```



```
udt_recv(pkt) && !corrupt(pkt)
msg = extract(pkt)
rdt_recv(msg)
```

Using a Checksum



rdt_send(msg)

pkt = make_pkt_w_chksum(msg)
udt_send(pkt)

Wait for application

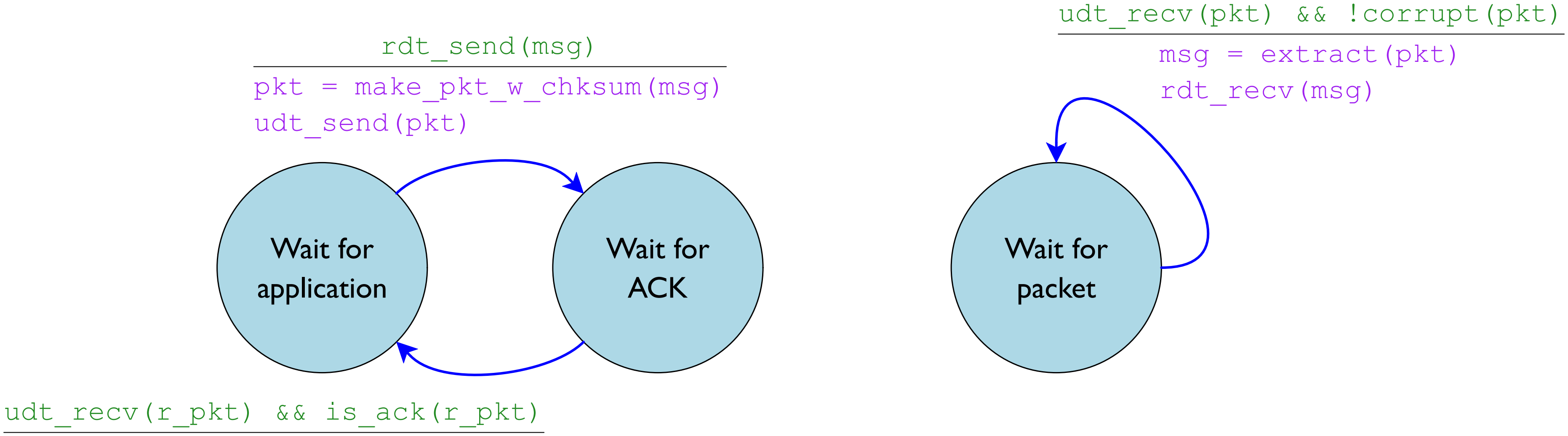
Wait for ACK

udt_recv(pkt) && !corrupt(pkt)

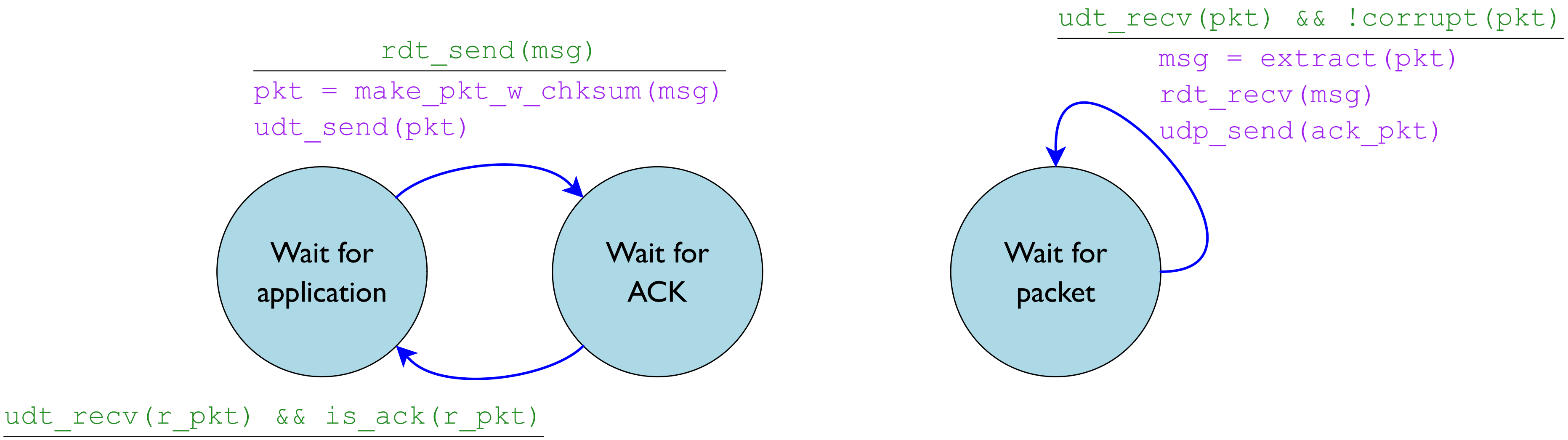
msg = extract(pkt)
rdt_recv(msg)

Wait for packet

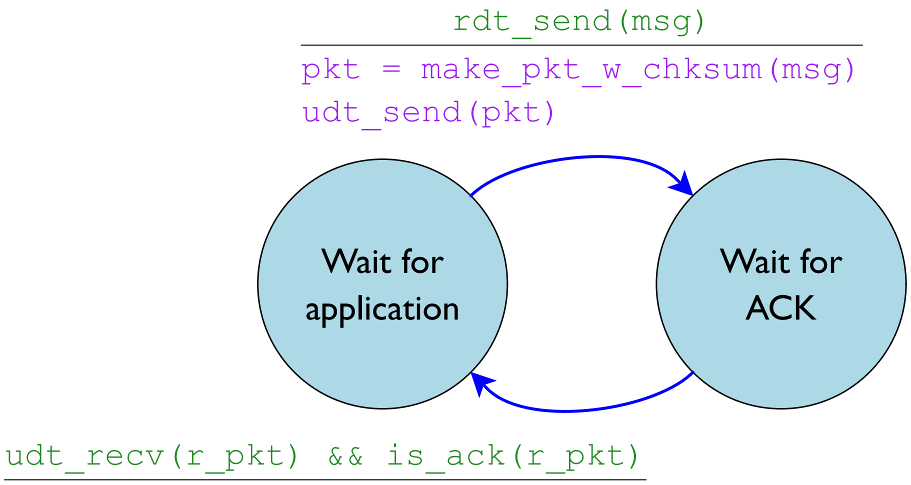
Using a Checksum



Using a Checksum



Using a Checksum



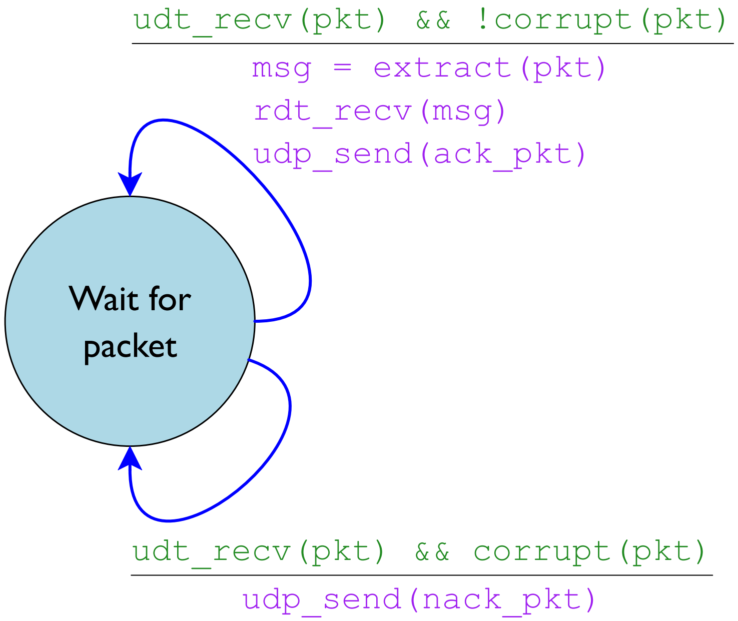
rdt_send(msg)

pkt = make_pkt_w_chksum(msg)
udt_send(pkt)

Wait for application

Wait for ACK

udt_recv(r_pkt) && is_ack(r_pkt)



udt_recv(pkt) && !corrupt(pkt)

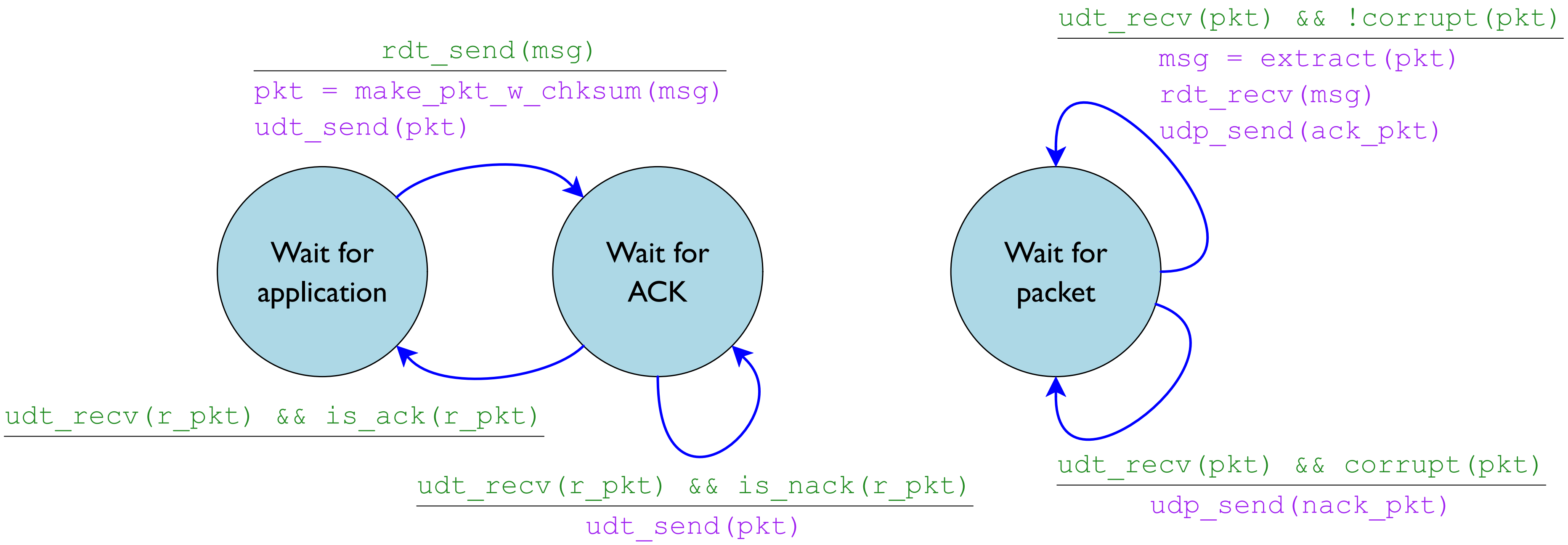
msg = extract(pkt)
rdt_recv(msg)
udp_send(ack_pkt)

Wait for packet

udt_recv(pkt) && corrupt(pkt)

udp_send(nack_pkt)

Using a Checksum



rdt_send(msg)

pkt = make_pkt_w_chksum(msg)
udt_send(pkt)

Wait for application

Wait for ACK

udt_recv(r_pkt) && is_ack(r_pkt)

udt_recv(r_pkt) && is_nack(r_pkt)

udt_send(pkt)

udt_recv(pkt) && !corrupt(pkt)

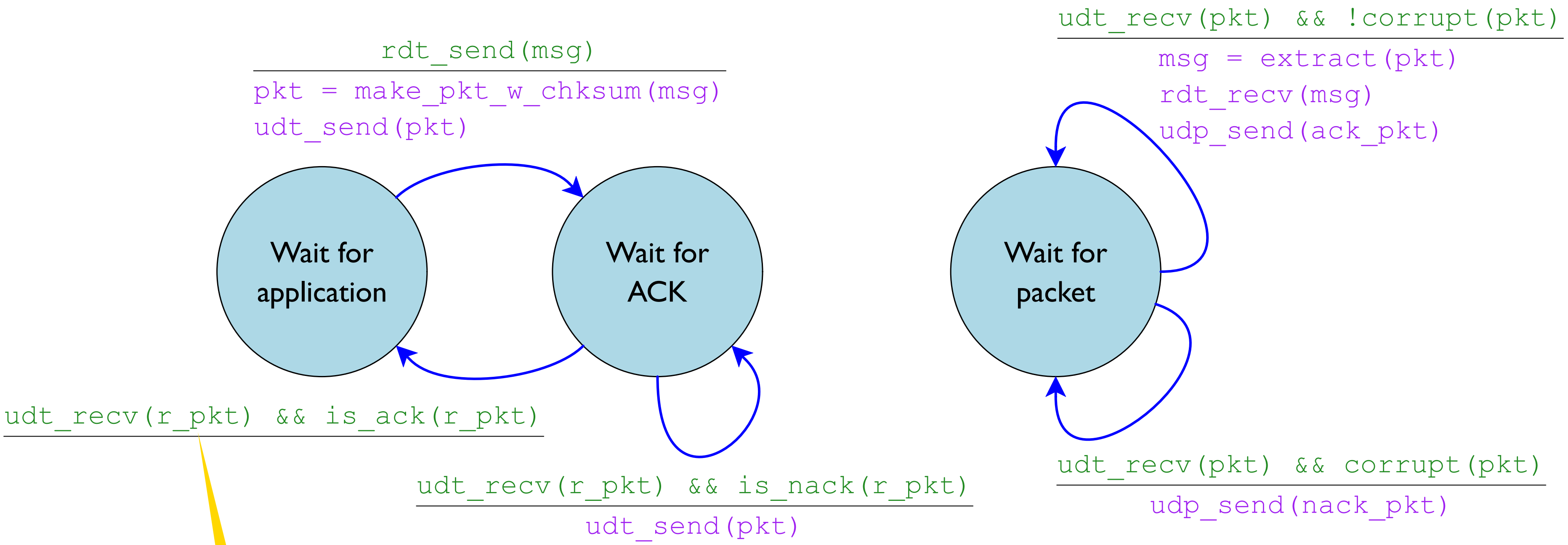
msg = extract(pkt)
rdt_recv(msg)
udp_send(ack_pkt)

Wait for packet

udt_recv(pkt) && corrupt(pkt)

udp_send(nack_pkt)

Using a Checksum



Wait for application

Wait for ACK

Wait for packet

rdt_send(msg)

pkt = make_pkt_w_chksum(msg)
udt_send(pkt)

udt_recv(r_pkt) && is_ack(r_pkt)

udt_recv(r_pkt) && is_nack(r_pkt)

udt_send(pkt)

udt_recv(pkt) && !corrupt(pkt)

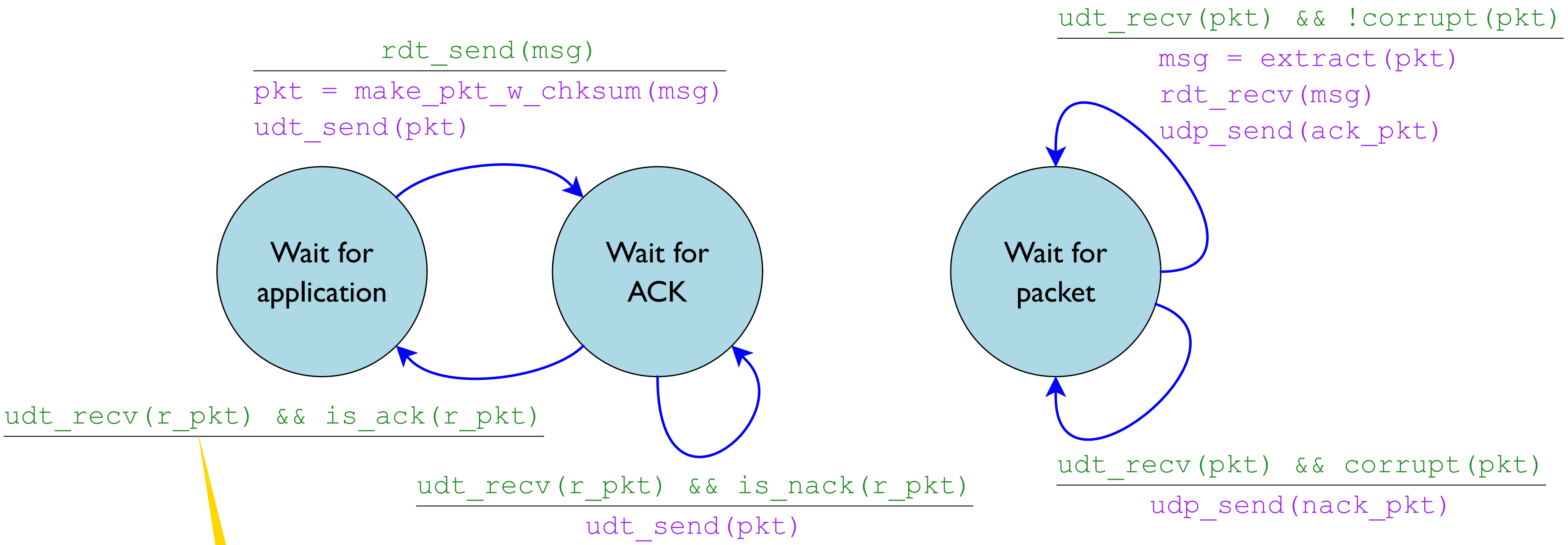
msg = extract(pkt)
rdt_recv(msg)
udp_send(ack_pkt)

udt_recv(pkt) && corrupt(pkt)

udp_send(nack_pkt)

What if `r_pkt` is corrupt?

Using a Checksum



rdt_send(msg)

pkt = make_pkt_w_chksum(msg)
udt_send(pkt)

Wait for application

Wait for ACK

udt_recv(r_pkt) && is_ack(r_pkt)

udt_recv(r_pkt) && is_nack(r_pkt)

udt_send(pkt)

udt_recv(pkt) && !corrupt(pkt)

msg = extract(pkt)
rdt_recv(msg)
udp_send(ack_pkt)

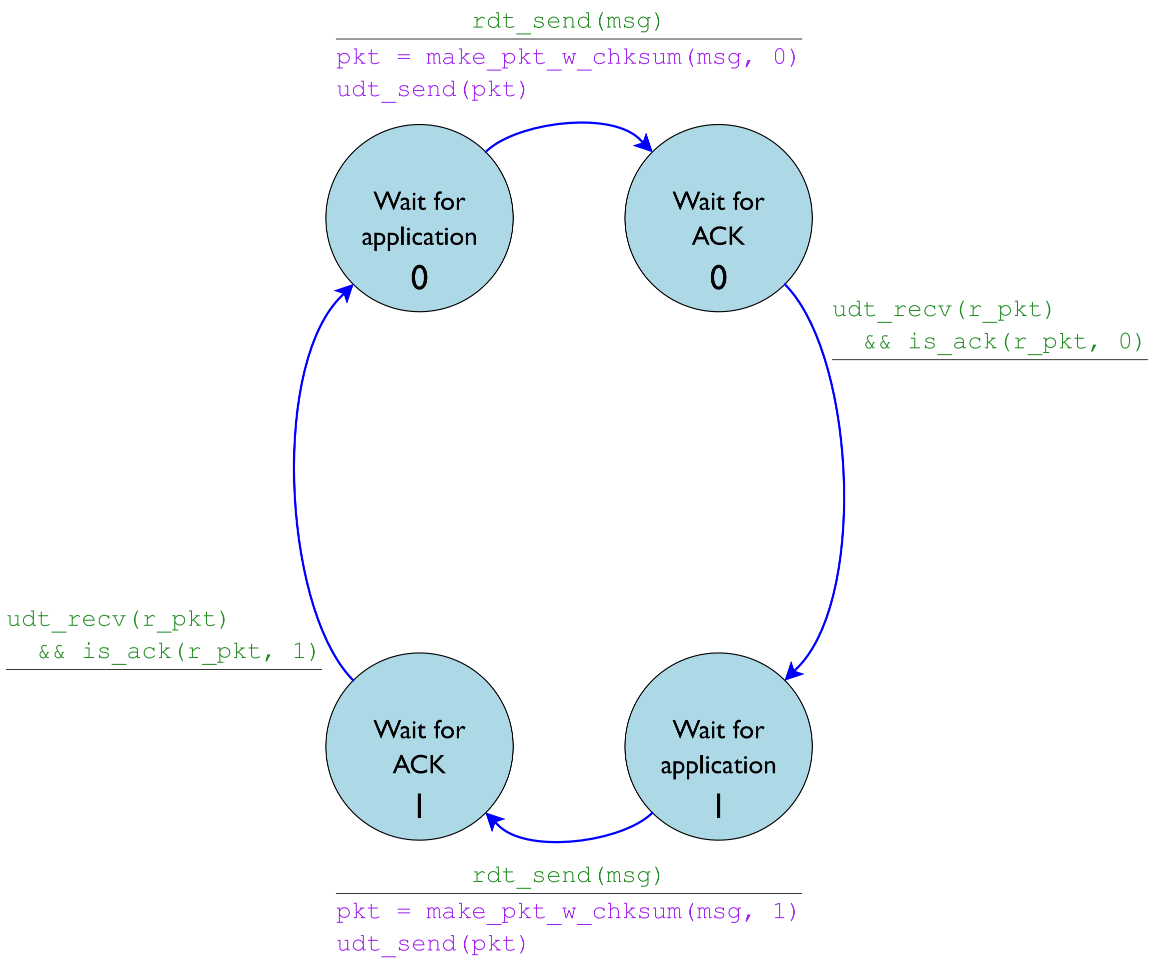
Wait for packet

udt_recv(pkt) && corrupt(pkt)

udp_send(nack_pkt)

How do we avoid an ACK of ACK of ACK...?

Handling ACK Corruption

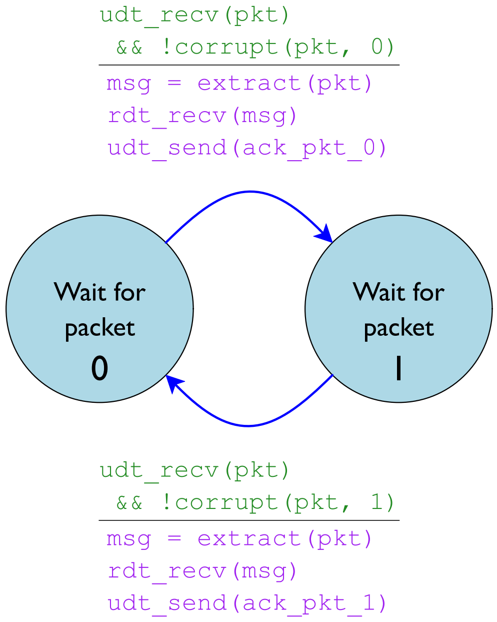


```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 0)
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 0)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 1)
```

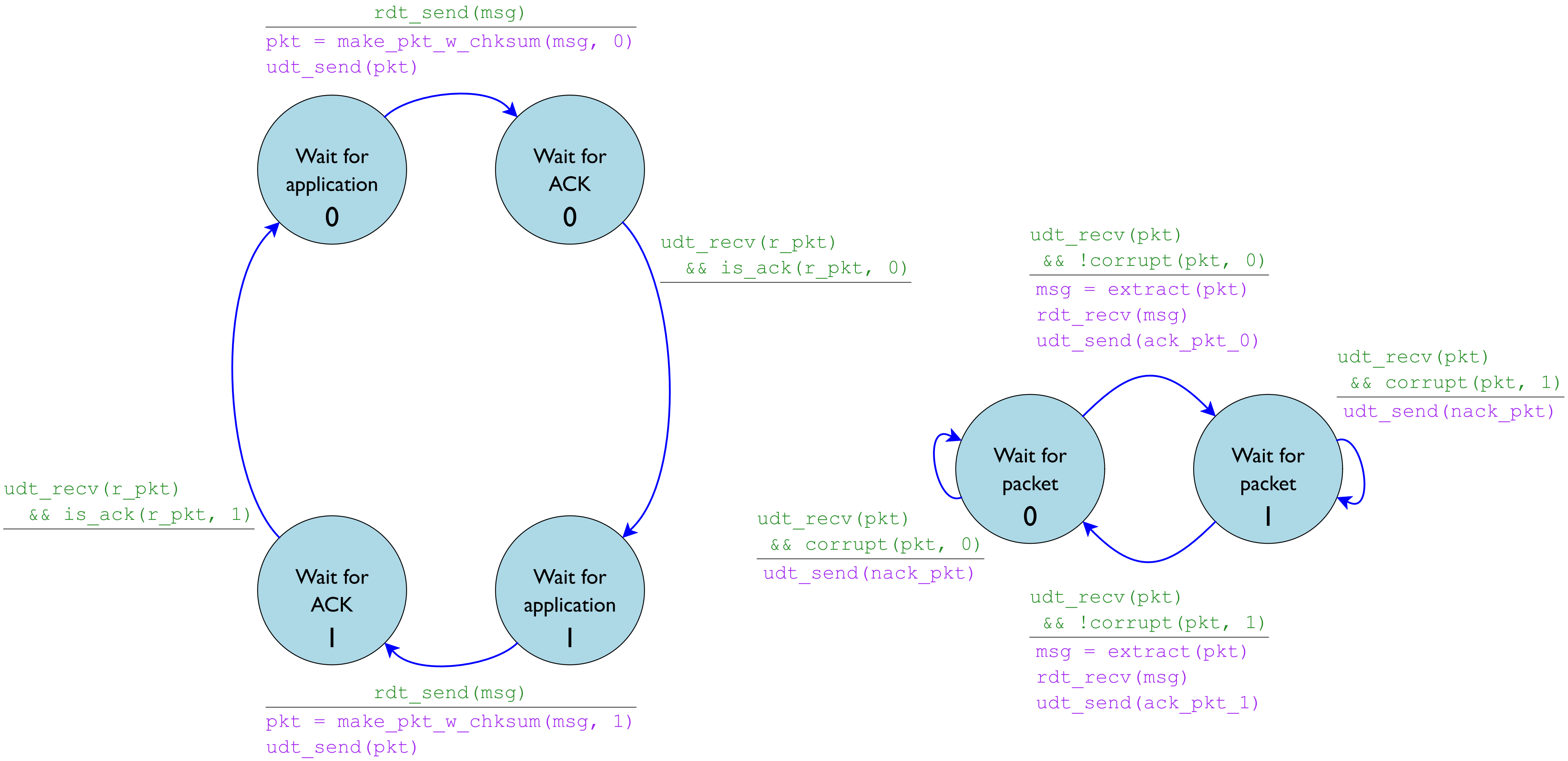
```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 1)
udt_send(pkt)
```



```
udt_recv(pkt)
 && !corrupt(pkt, 0)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_0)
```

```
udt_recv(pkt)
 && !corrupt(pkt, 1)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_1)
```

Handling ACK Corruption



```
rdt_send(msg)
---
pkt = make_pkt_w_chksum(msg, 0)
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 0)
---
```

```
rdt_send(msg)
---
pkt = make_pkt_w_chksum(msg, 1)
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 1)
---
```

```
udt_recv(pkt)
 && !corrupt(pkt, 0)
---
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_0)
```

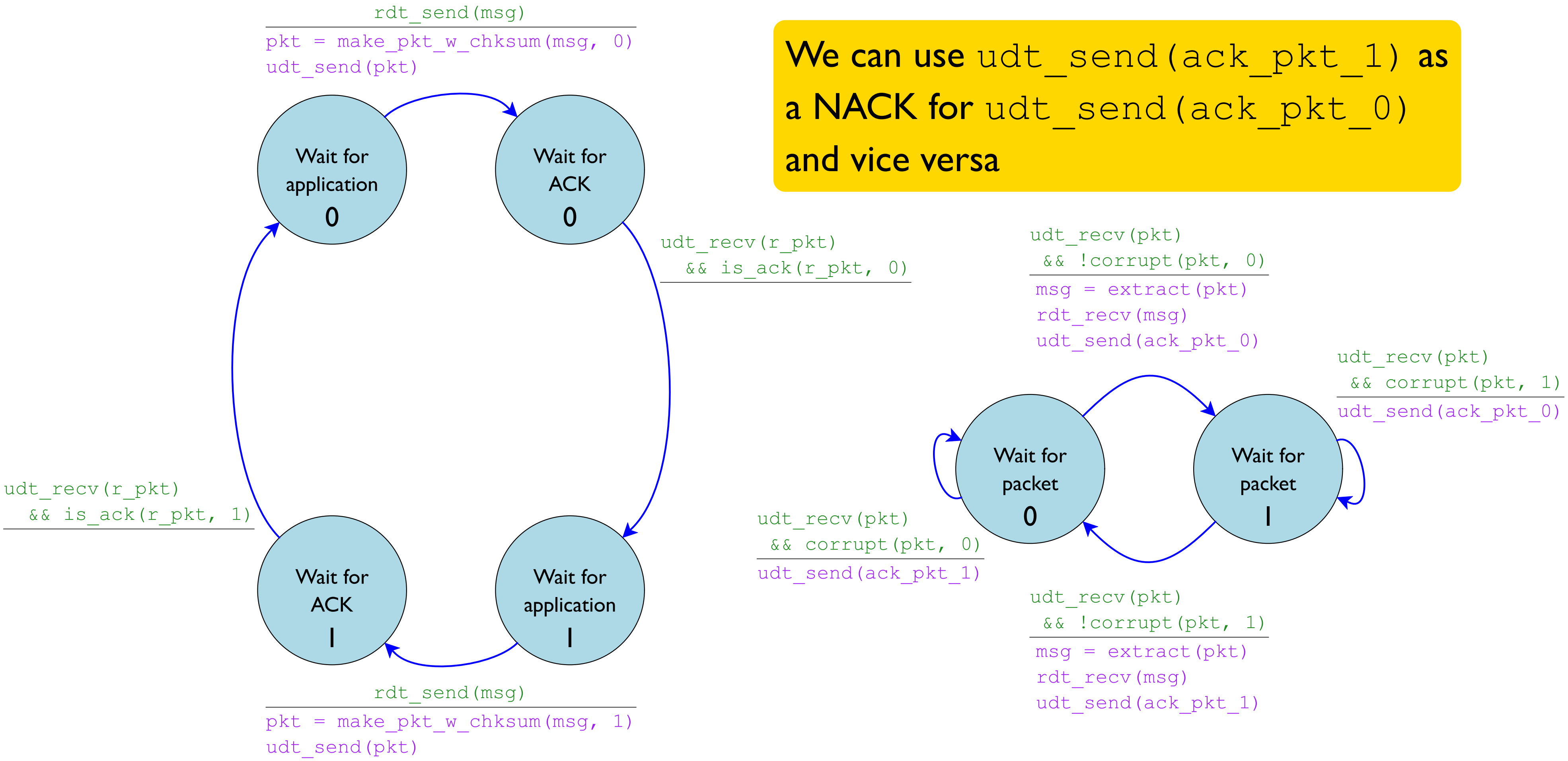
```
udt_recv(pkt)
 && corrupt(pkt, 1)
---
udt_send(nack_pkt)
```

```
udt_recv(pkt)
 && corrupt(pkt, 0)
---
udt_send(nack_pkt)
```

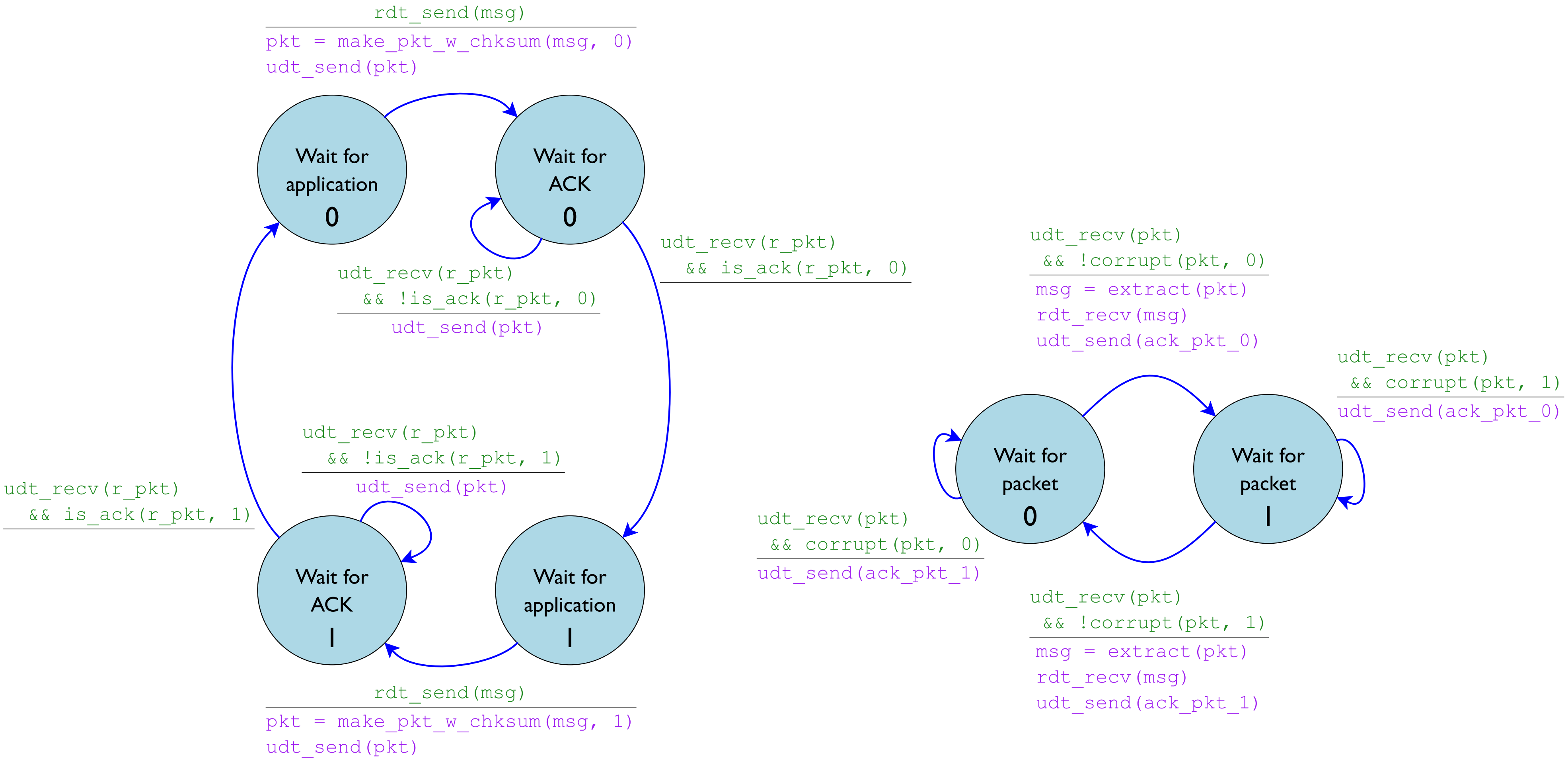
```
udt_recv(pkt)
 && !corrupt(pkt, 1)
---
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_1)
```

Handling ACK Corruption

We can use `udt_send(ack_pkt_1)` as a NACK for `udt_send(ack_pkt_0)` and vice versa



Handling ACK Corruption



Wait for application 0

Wait for ACK 0

Wait for ACK 1

Wait for application 1

rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 0)
udt_send(pkt)

udt_recv(r_pkt)
&& !is_ack(r_pkt, 0)
udt_send(pkt)

udt_recv(r_pkt)
&& is_ack(r_pkt, 0)

udt_recv(r_pkt)
&& !is_ack(r_pkt, 1)
udt_send(pkt)

udt_recv(r_pkt)
&& is_ack(r_pkt, 1)

rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 1)
udt_send(pkt)

Wait for packet 0

Wait for packet 1

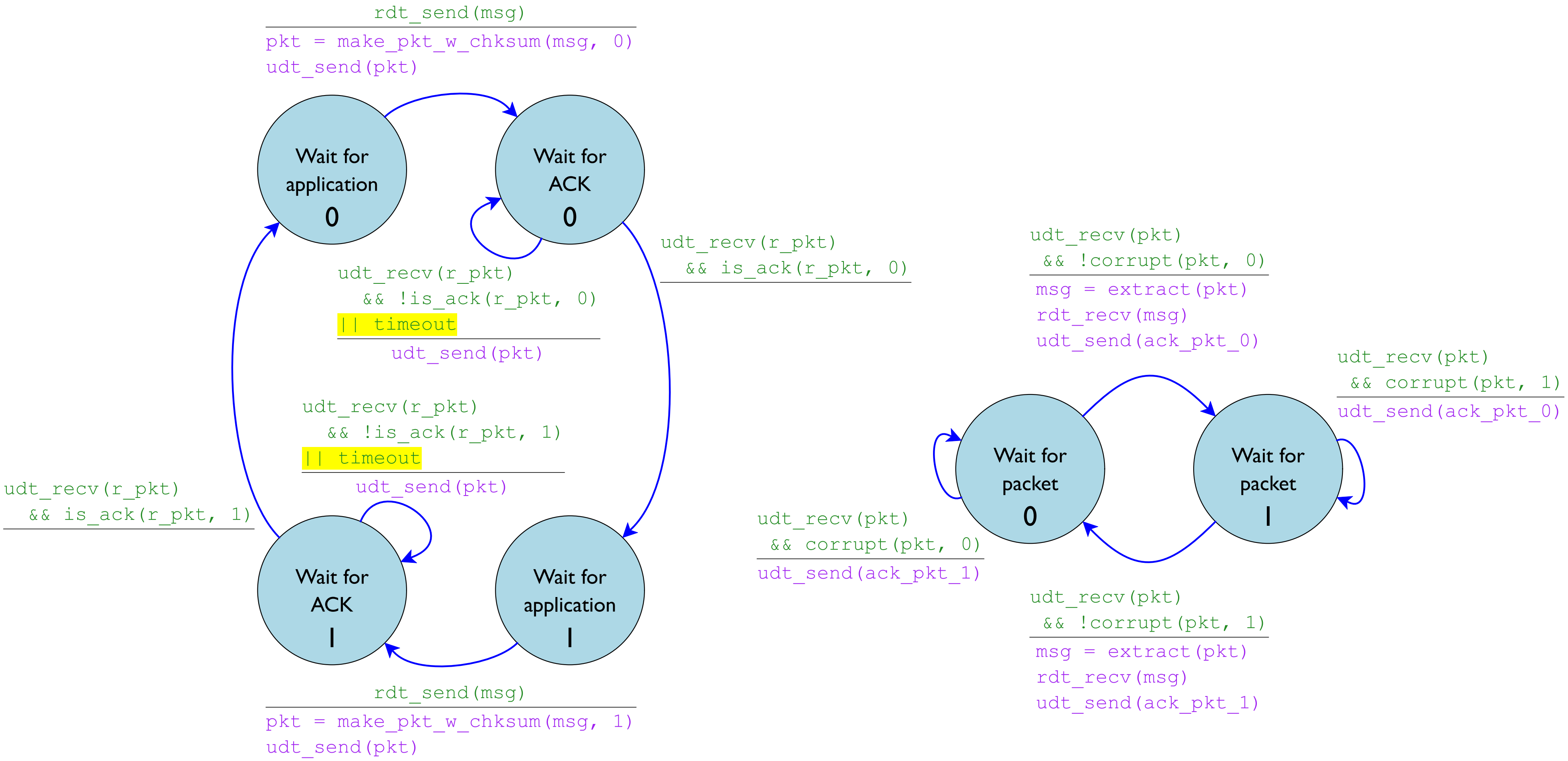
udt_recv(pkt)
&& !corrupt(pkt, 0)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_0)

udt_recv(pkt)
&& corrupt(pkt, 1)
udt_send(ack_pkt_0)

udt_recv(pkt)
&& corrupt(pkt, 0)
udt_send(ack_pkt_1)

udt_recv(pkt)
&& !corrupt(pkt, 1)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_1)

Handling Corrupt and Lost Packets



```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 0)
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && !is_ack(r_pkt, 0)
|| timeout
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 0)
```

```
udt_recv(r_pkt)
  && !is_ack(r_pkt, 1)
|| timeout
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 1)
```

```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 1)
udt_send(pkt)
```

```
udt_recv(pkt)
  && !corrupt(pkt, 0)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_0)
```

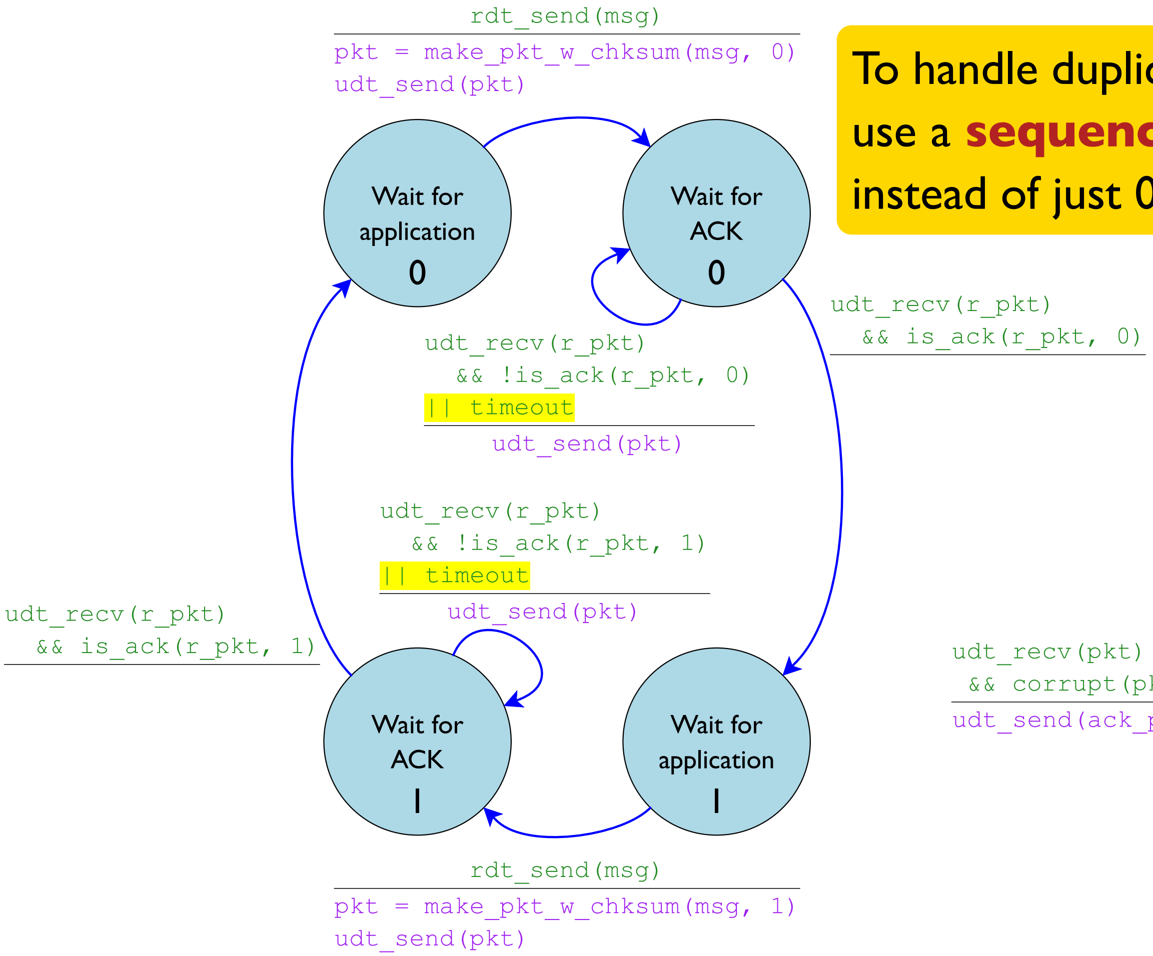
```
udt_recv(pkt)
  && corrupt(pkt, 1)
udt_send(ack_pkt_0)
```

```
udt_recv(pkt)
  && corrupt(pkt, 0)
udt_send(ack_pkt_1)
```

```
udt_recv(pkt)
  && !corrupt(pkt, 1)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_1)
```

Handling Corrupt and Lost Packets

To handle duplicated and reordered packets, use a **sequence number** that always counts up instead of just 0 and 1



```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 0)
udt_send(pkt)
```

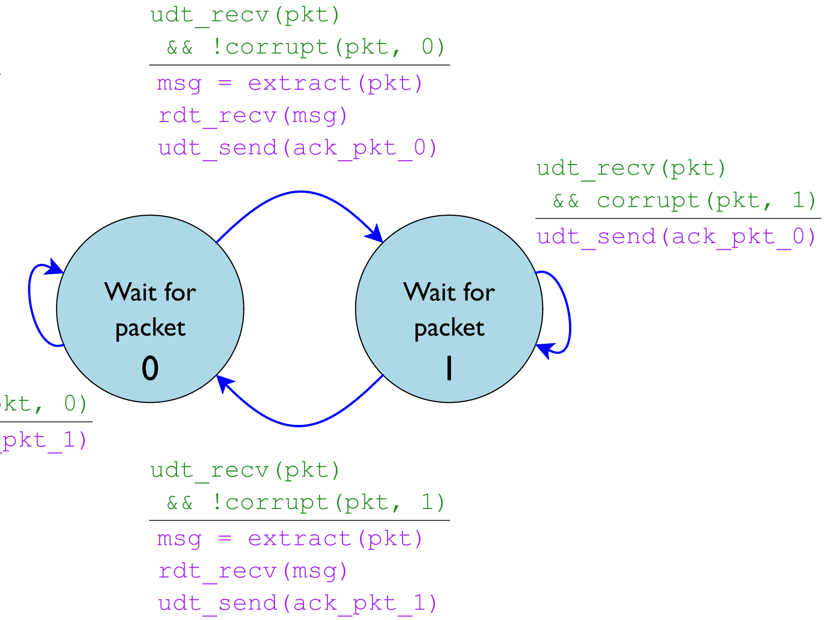
```
udt_recv(r_pkt)
  && !is_ack(r_pkt, 0)
|| timeout
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 0)
```

```
udt_recv(r_pkt)
  && !is_ack(r_pkt, 1)
|| timeout
udt_send(pkt)
```

```
udt_recv(r_pkt)
  && is_ack(r_pkt, 1)
```

```
rdt_send(msg)
pkt = make_pkt_w_chksum(msg, 1)
udt_send(pkt)
```



```
udt_recv(pkt)
  && !corrupt(pkt, 0)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_0)
```

```
udt_recv(pkt)
  && corrupt(pkt, 1)
udt_send(ack_pkt_0)
```

```
udt_recv(pkt)
  && corrupt(pkt, 0)
udt_send(ack_pkt_1)
```

```
udt_recv(pkt)
  && !corrupt(pkt, 1)
msg = extract(pkt)
rdt_recv(msg)
udt_send(ack_pkt_1)
```

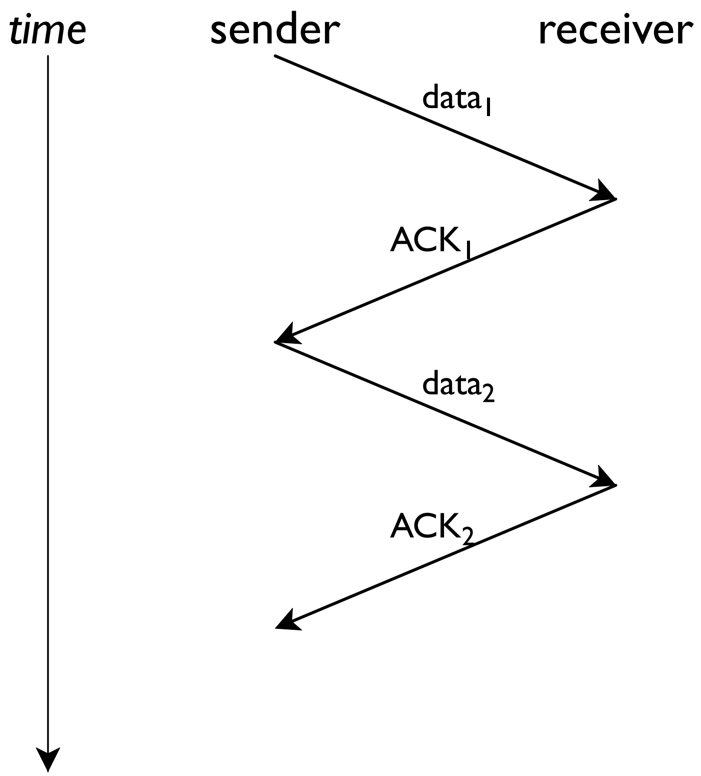
Choosing a Timeout

RTT is minimum useful timeout

- too small ⇒ resend data and ACKs unnecessarily
- too large ⇒ sender waits too long to resend

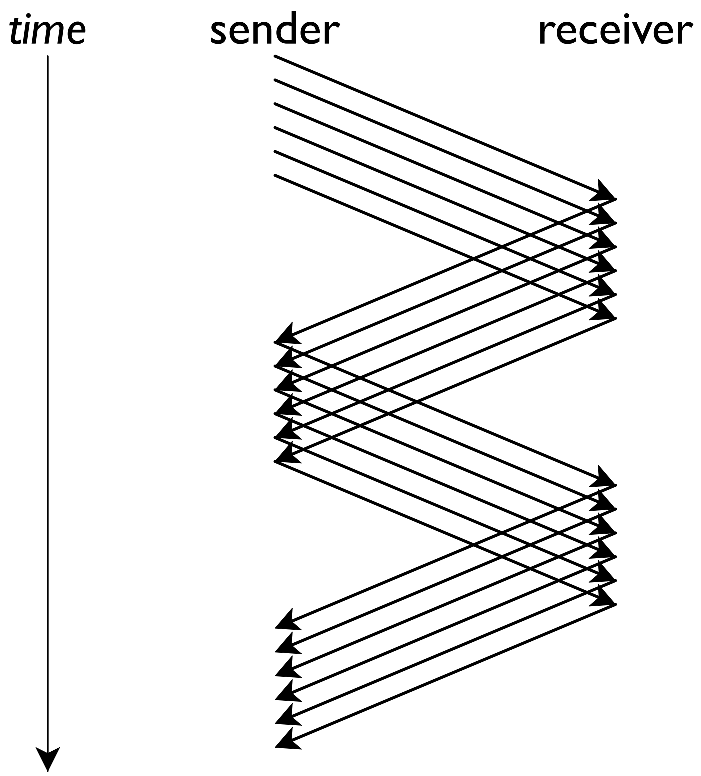
scale × avg(RTT) + stddev(RTT) is a good approach

Sequential Messages



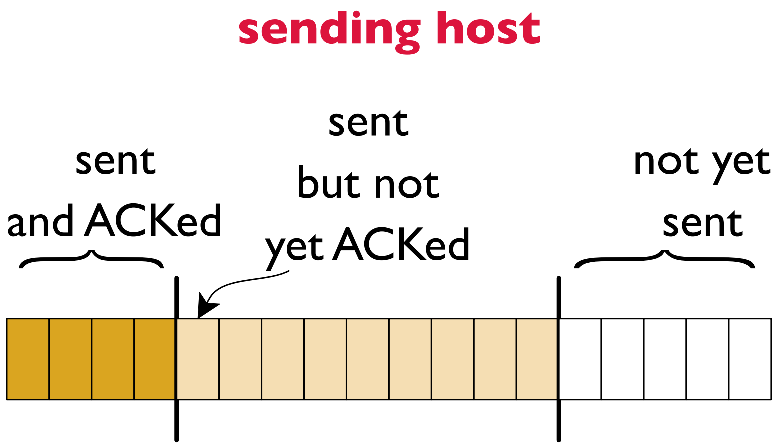
Throughput is limited by latency

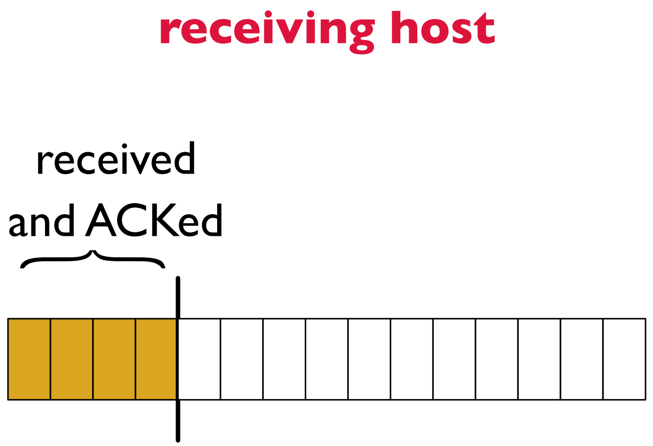
Pipelined Messages



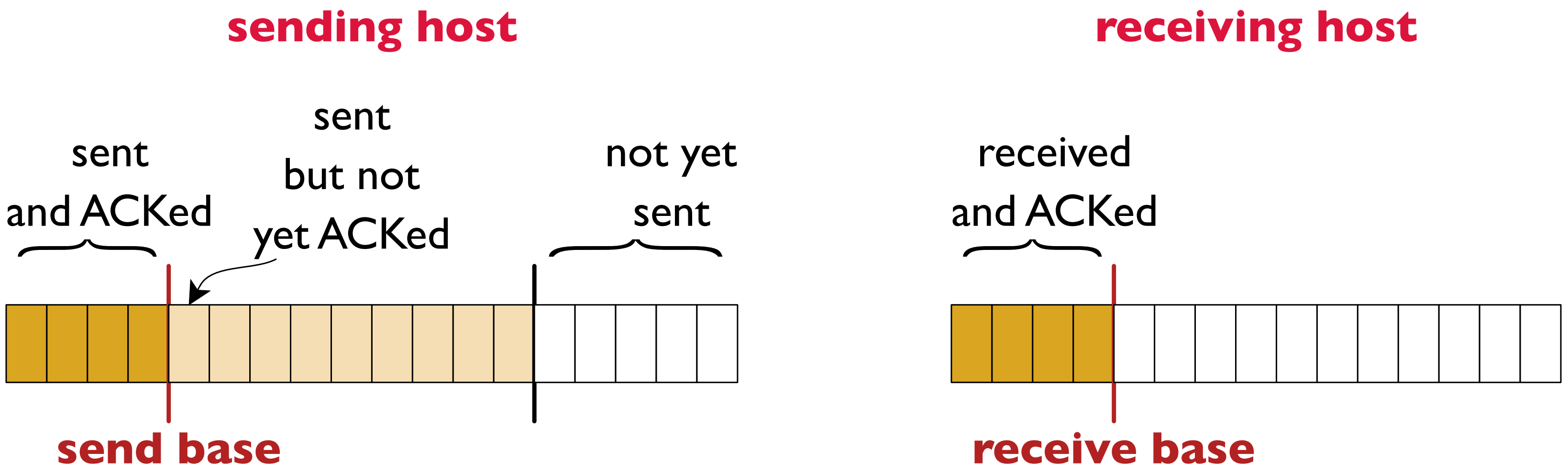
Need a way to track multiple packets in flight

Buffers

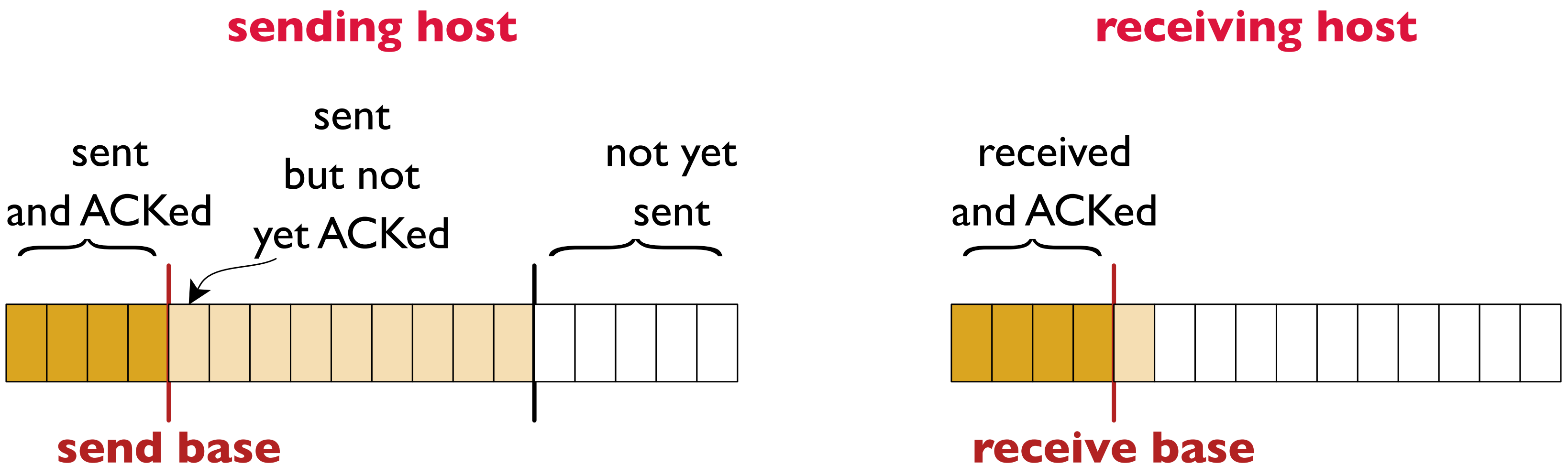

sending host
sent
and ACKed
sent
but not
yet ACKed
not yet
sent


receiving host
received
and ACKed

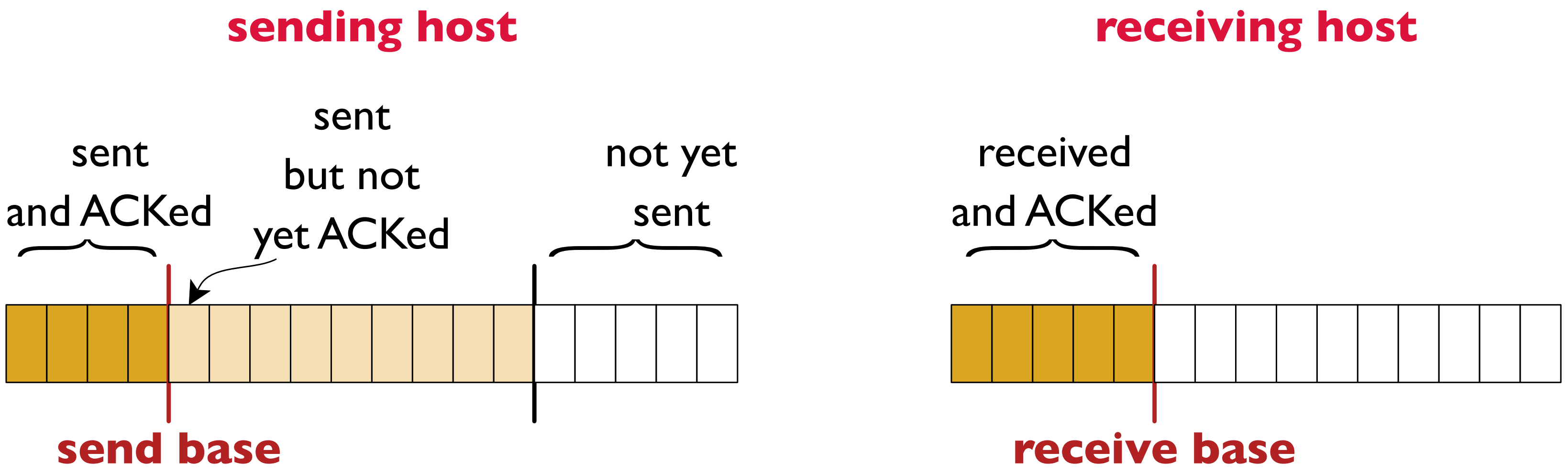
Buffers


sending host
sent
and ACKed
sent
but not
yet ACKed
not yet
sent
send base
receiving host
received
and ACKed
receive base

Buffers


sending host
sent
and ACKed
sent
but not
yet ACKed
not yet
sent
send base
receiving host
received
and ACKed
receive base

Buffers



sending host

sent and ACKed

sent but not yet ACKed

not yet sent

send base

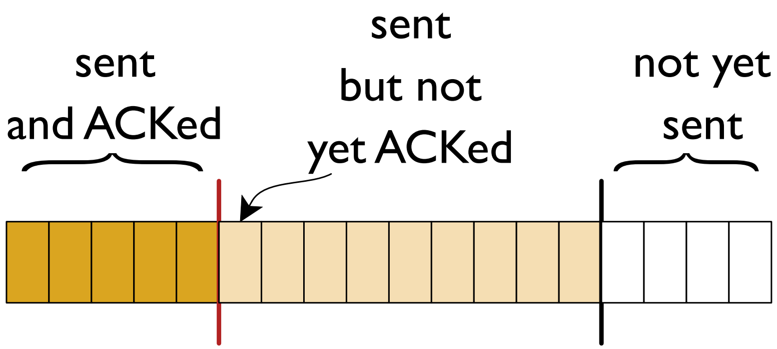
receiving host

received and ACKed

receive base

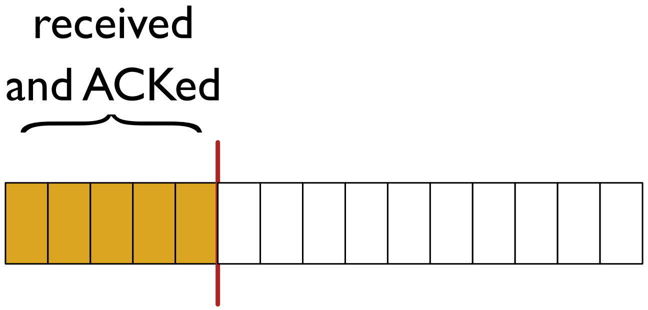
Buffers

sending host



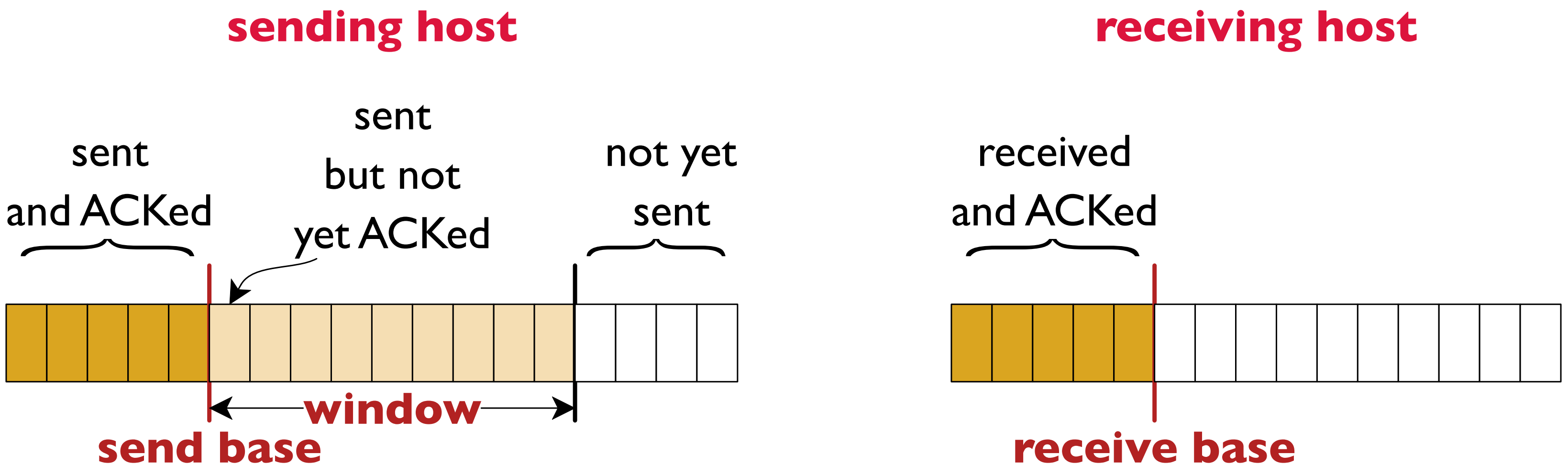
send base

receiving host

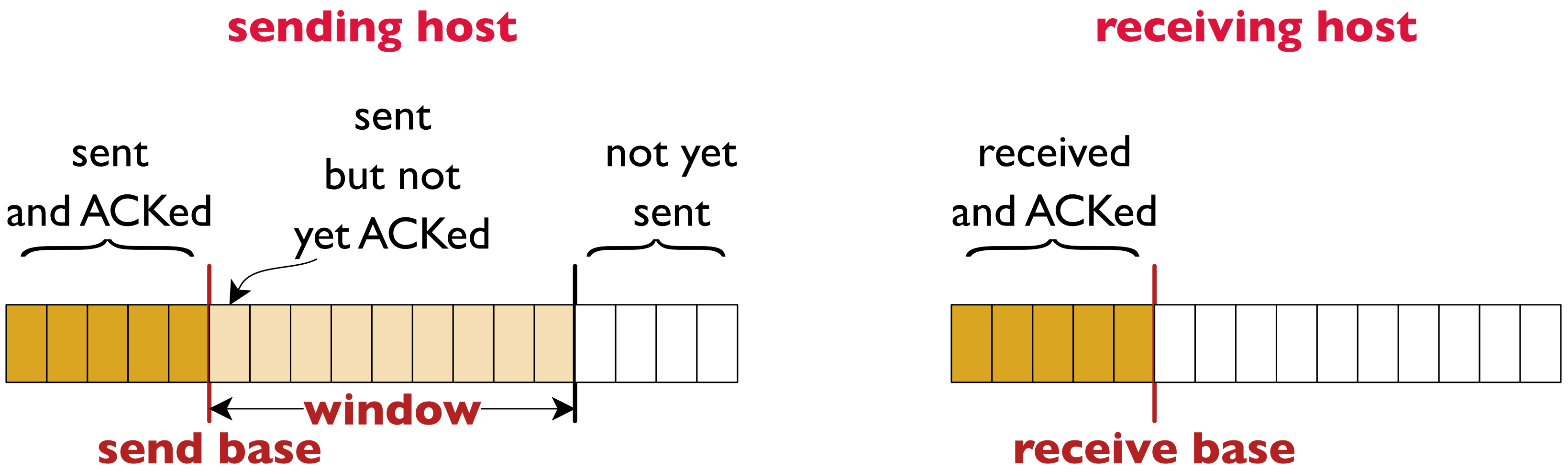


receive base

Buffers

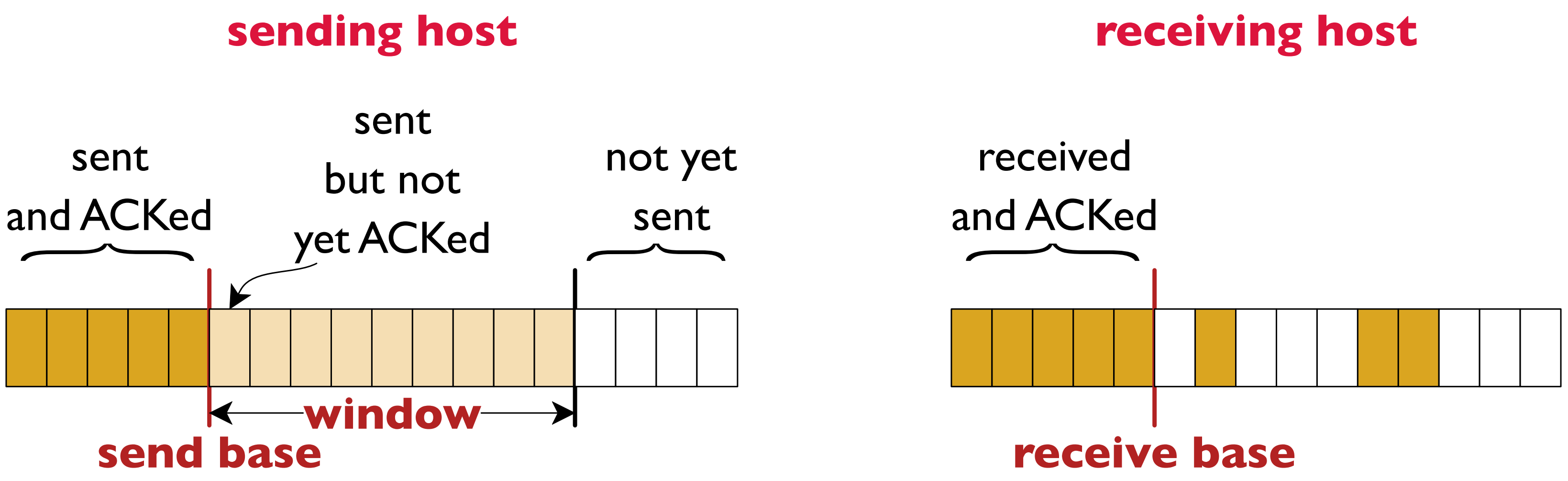

sending host
sent
and ACKed
sent
but not
yet ACKed
not yet
sent
window
send base
receiving host
received
and ACKed
receive base

Buffers



Like a timeout, the window size needs to be chosen well

Buffers



sending host

sent
and ACKed

sent
but not
yet ACKed

not yet
sent

window

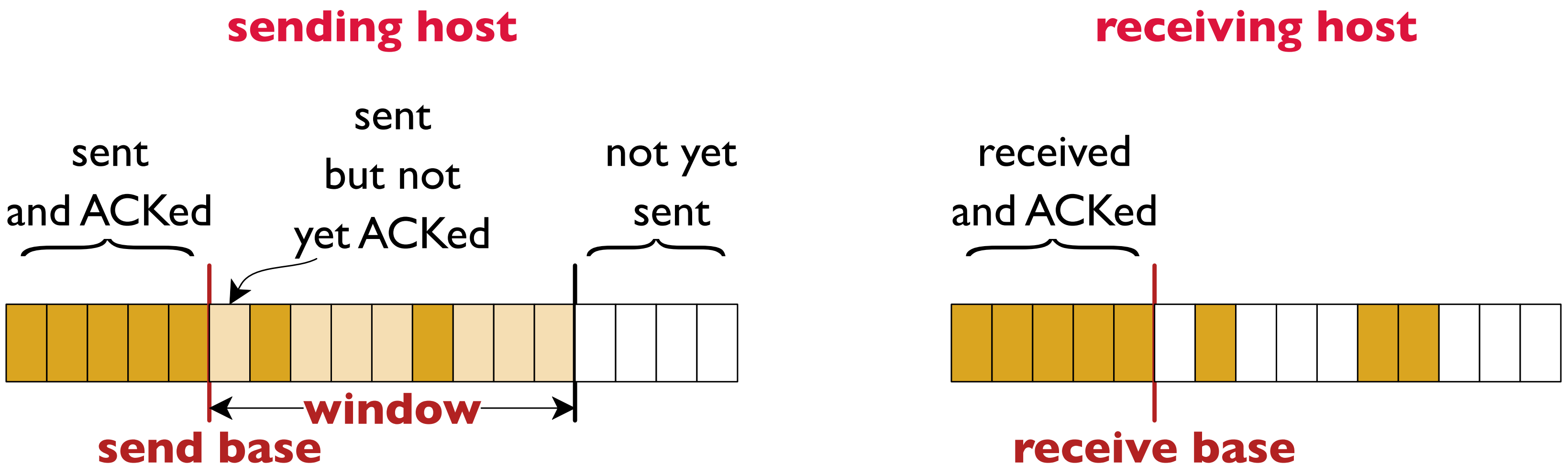
send base

receiving host

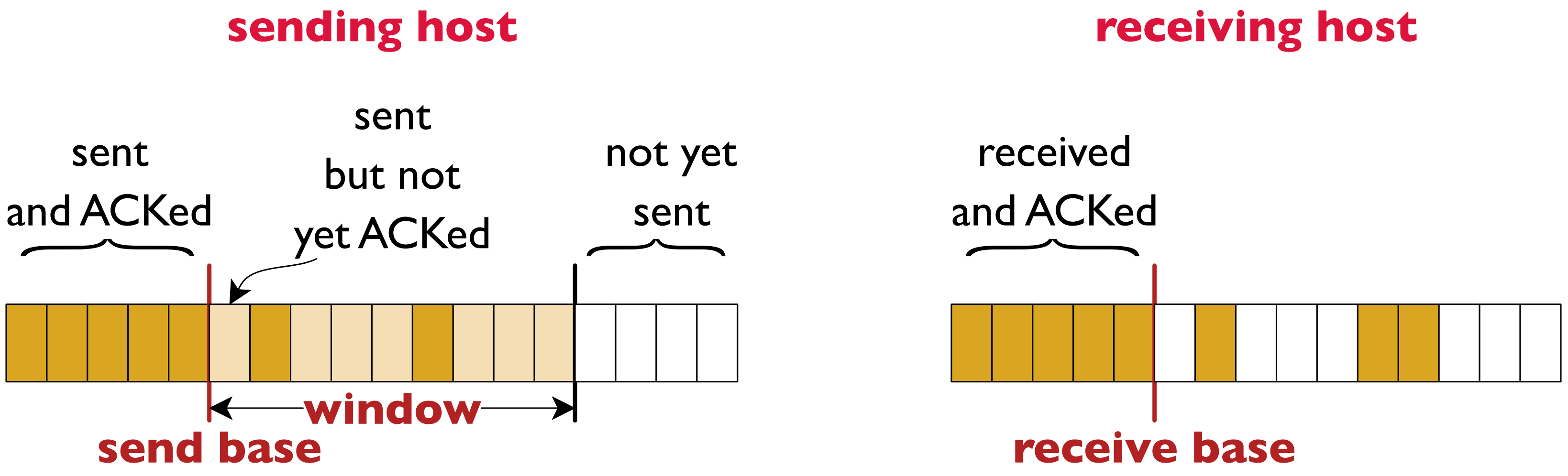
received
and ACKed

receive base

Buffers

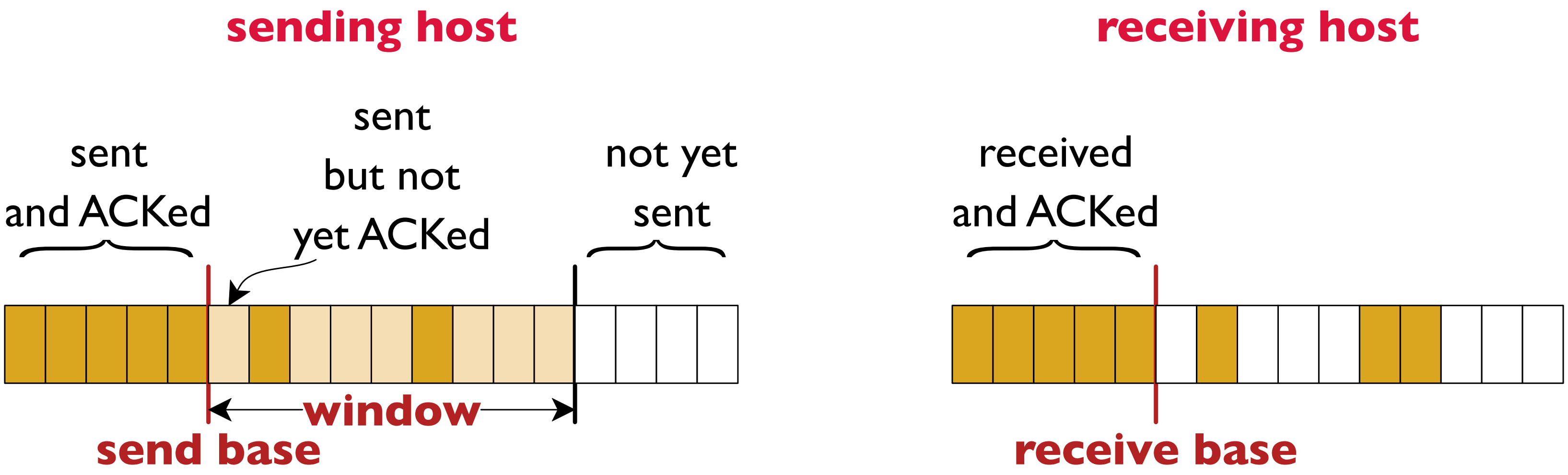

sending host
sent
and ACKed
sent
but not
yet ACKed
not yet
sent
window
send base
receiving host
received
and ACKed
receive base

Buffers



Selective repeat: on timeout, re-send unACKed

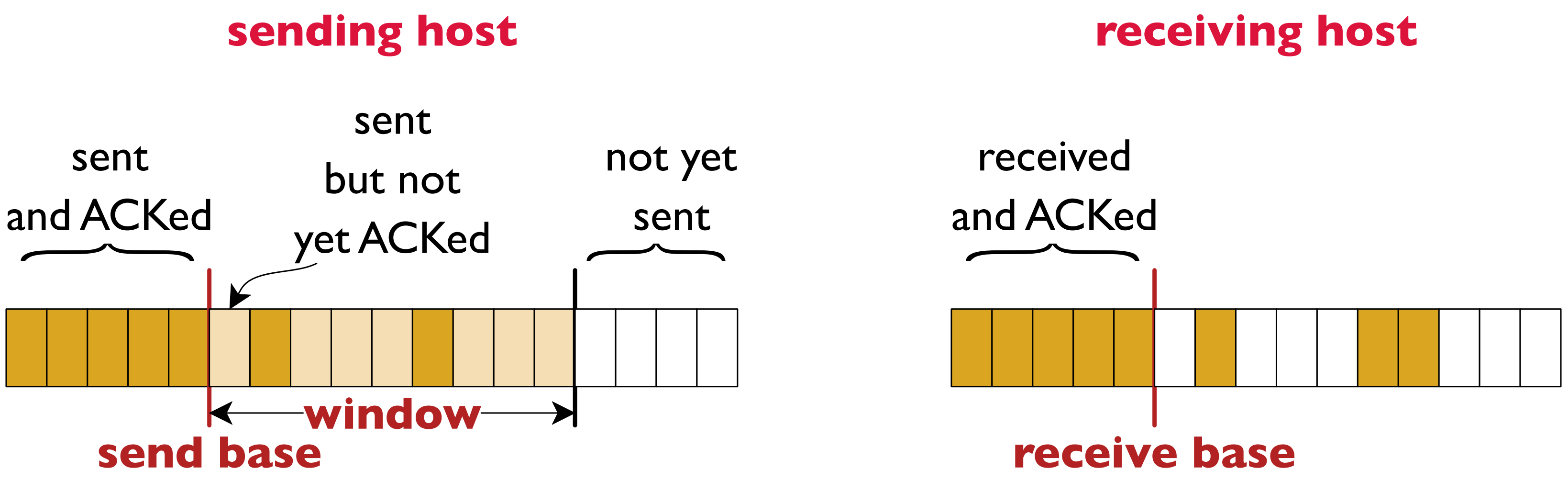
Buffers



Selective repeat: on timeout, re-send unACKed

Each packet must be specifically ACKed

Buffers



sending host

sent
and ACKed

sent
but not
yet ACKed

not yet
sent

window

send base

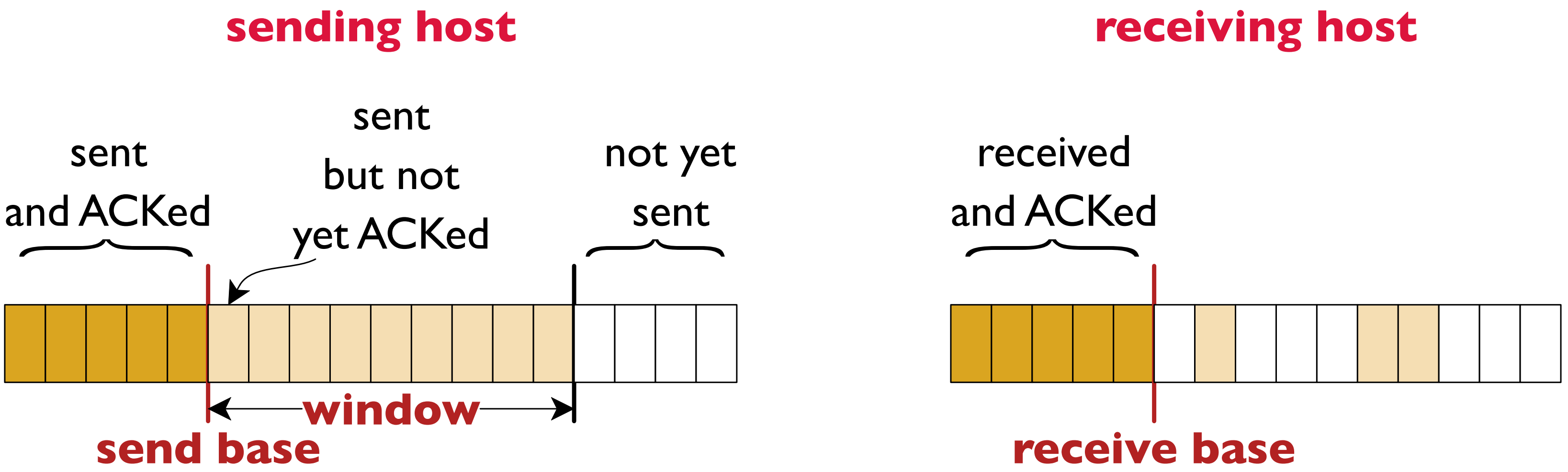
receiving host

received
and ACKed

receive base

Go-Back-N: on timeout, re-send in window

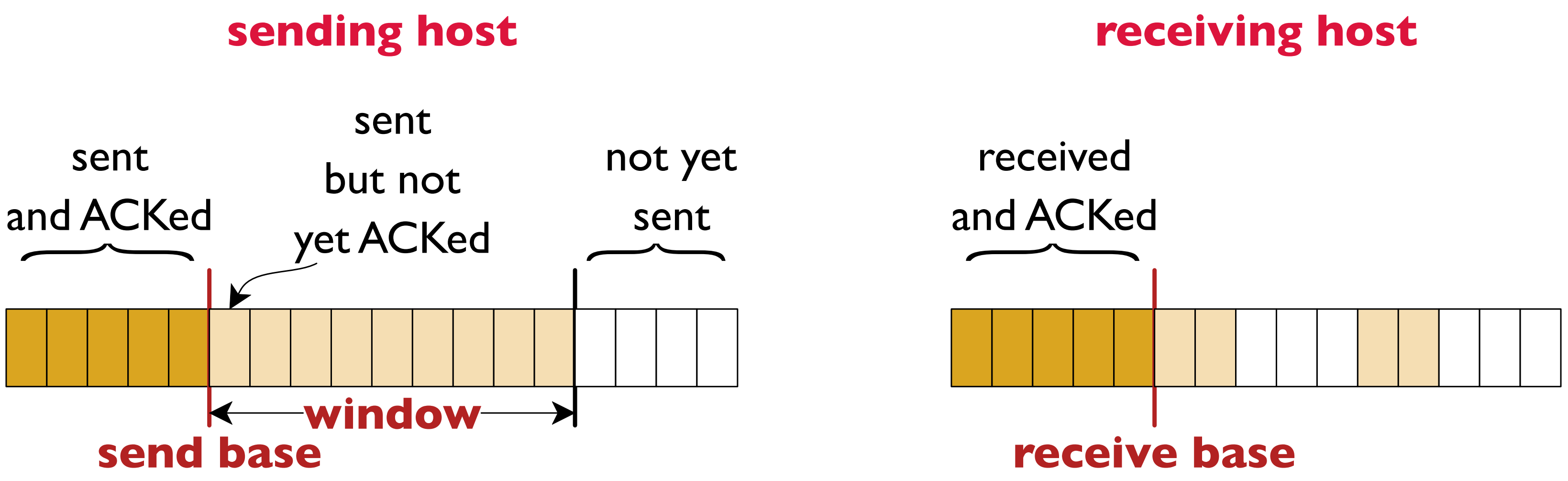
Buffers



Go-Back-N: on timeout, re-send in window

Can use a **cumulative** ACK

Buffers



sending host

sent
and ACKed

sent
but not
yet ACKed

not yet
sent

window

send base

receiving host

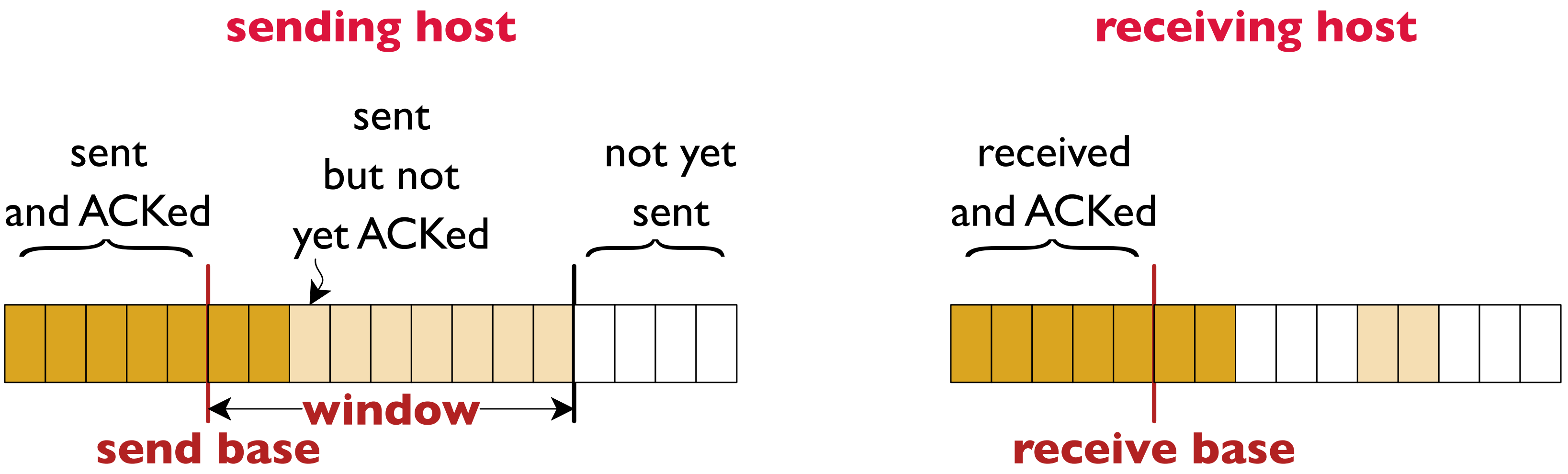
received
and ACKed

receive base

Go-Back-N: on timeout, re-send in window

Can use a **cumulative** ACK

Buffers



Go-Back-N: on timeout, re-send in window

Can use a **cumulative** ACK

Summary

Reliable data transfer can be implemented on top of an unreliable layer

State machines abstract over program details to explain just the program's states and transitions

- **ACKs** and **NACKs**
- **sequence numbers** for both ACKs and implicit NACKs
- **cumulative** ACKs versus **selective repeat**