CG4002: Computer Engineering Capstone Project

Internal communications: Body area network over Bluetooth Low Energy

JITHIN VACHERY

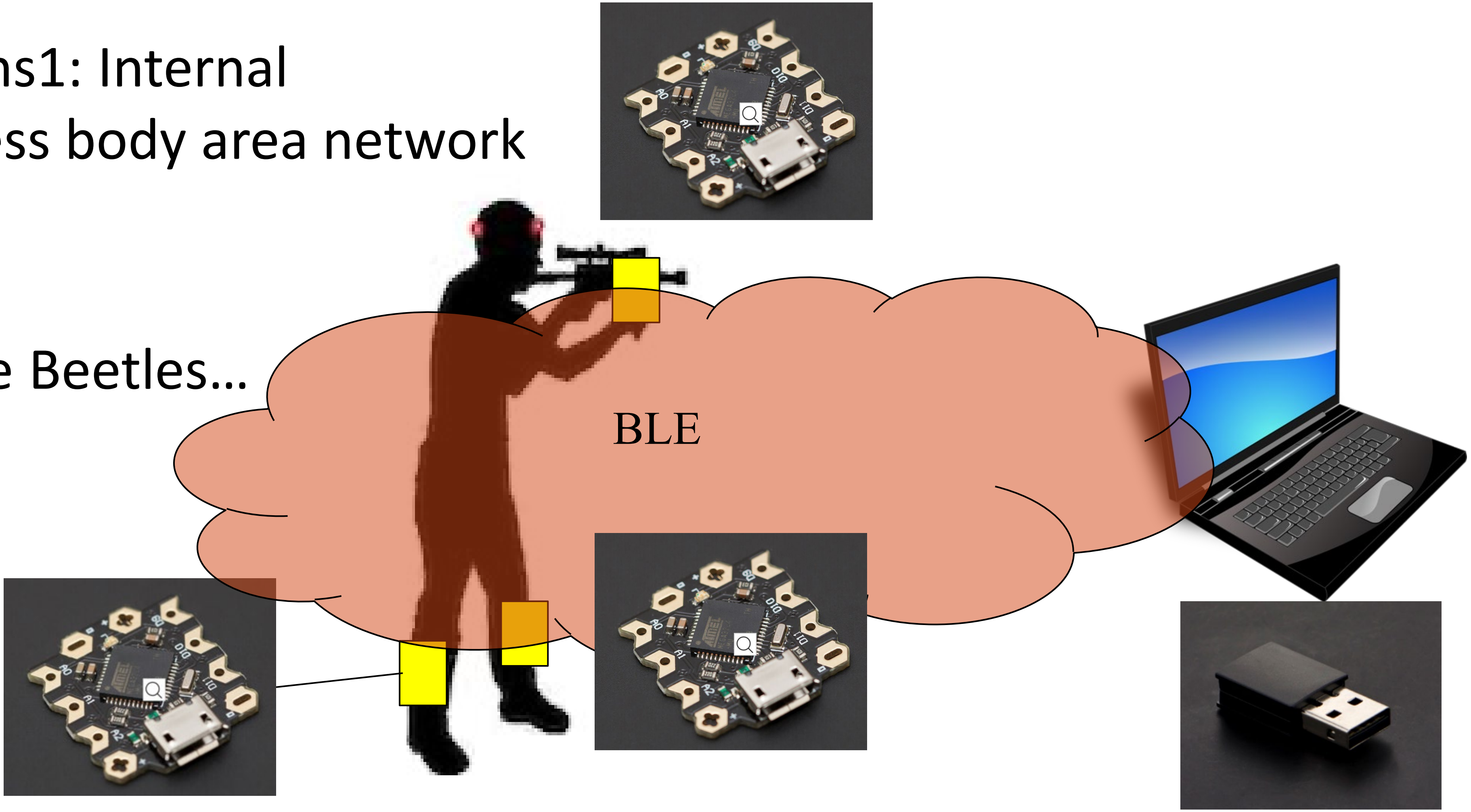
jithin@comp.nus.edg.sg

Slides adopted from
Prof. Peh Li Shiuan

Comms Internal:

Comms1: Internal
wireless body area network

...multiple Beetles...



Building a protocol

“Sending Data Reliably is Much Harder Than You Think. The Intricacy Involved in Ensuring Reliability Will Make Your Head Explode

Reliable Transfer over Unreliable Channel

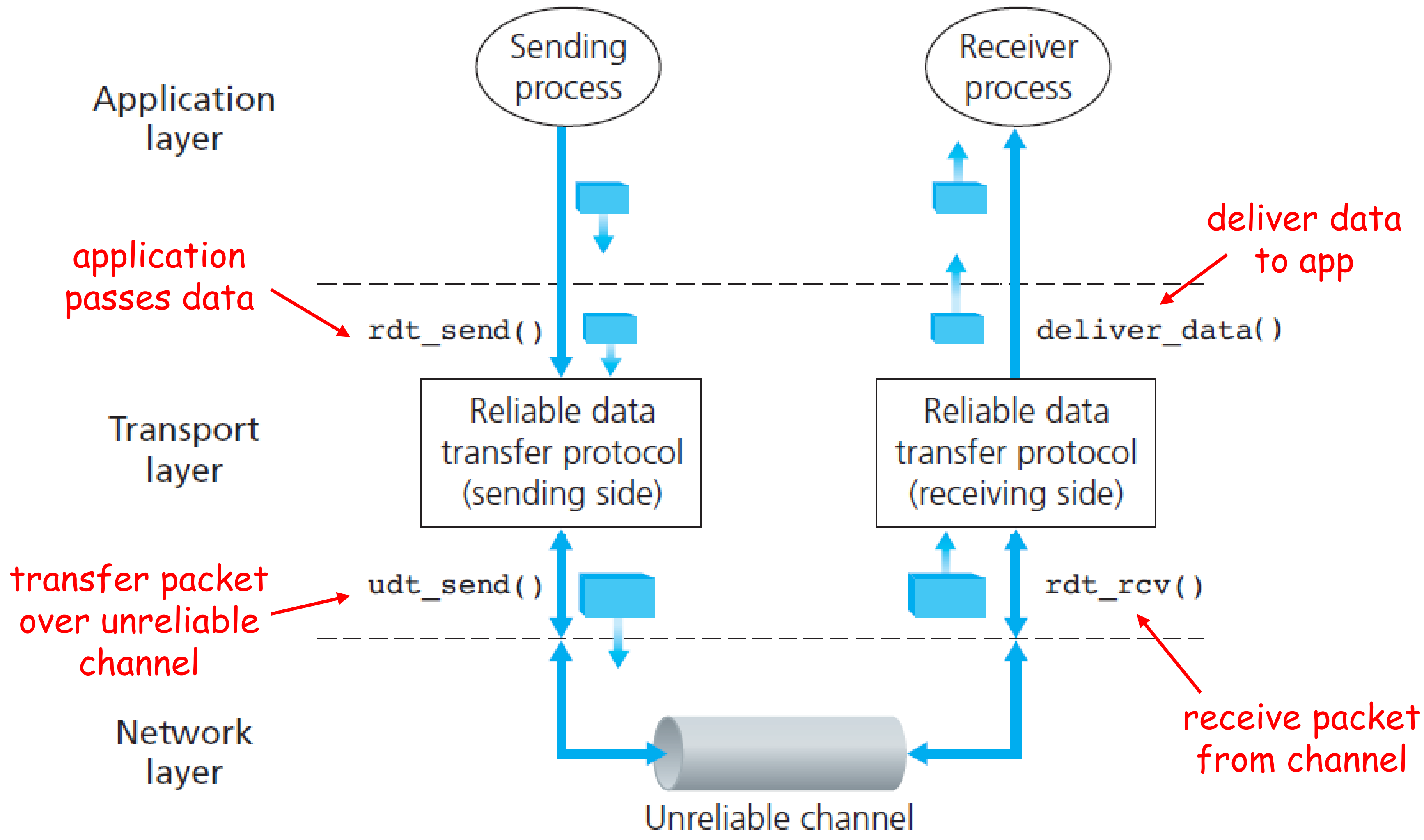
Underlying network may

- corrupt packets
- drop packets
- re-order packets (not considered in this lecture)
- deliver packets after an arbitrarily long delay

End-to-end reliable transport service should

- guarantee packets delivery and correctness
- deliver packets (to receiver application) in the same order they are sent

Reliable Data Transfer: Service Model



Reliable Data Transfer Protocols

- Characteristics of unreliable channel will determine the complexity of reliable data transfer protocols (rdt).

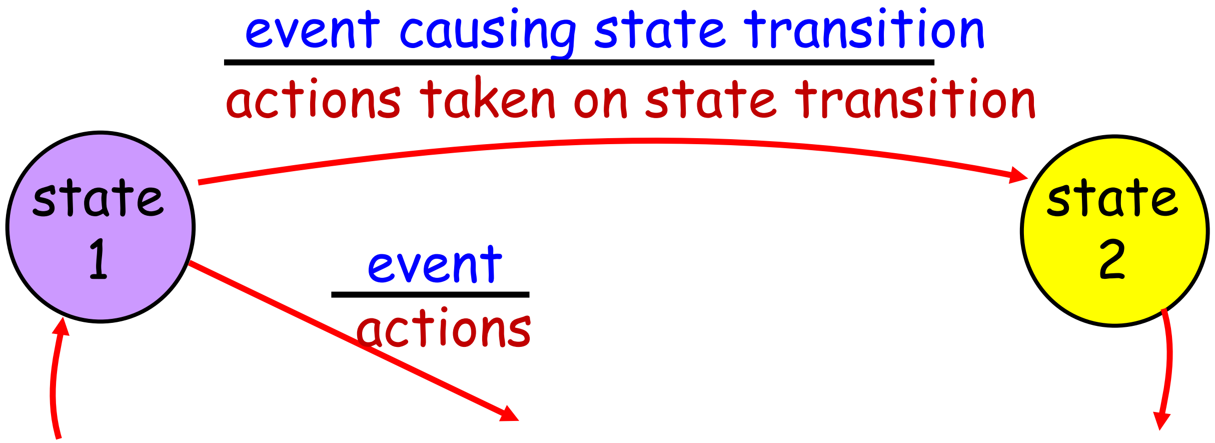
- We will incrementally develop sender & receiver sides of rdt protocols, considering increasingly complex models of unreliable channel.

- We consider only unidirectional data transfer
  - but control info may flow in reverse direction!

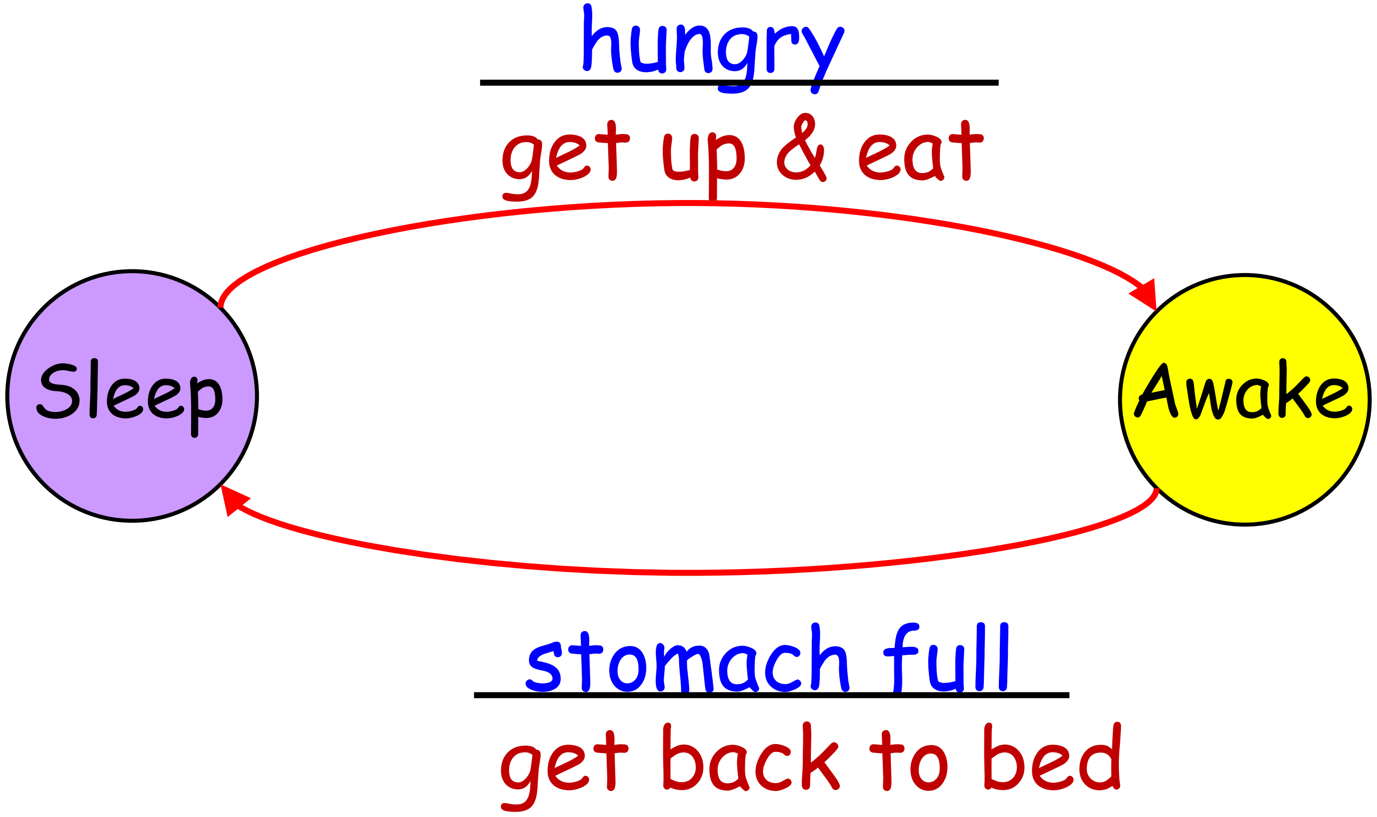
Finite State Machine (FSM)

We will use finite state machines (FSM) to describe sender and receiver of a protocol.

- We will learn a protocol by examples, but FSM provides you the complete picture to refer to as necessary.


event causing state transition
actions taken on state transition
state 1
event
actions
state 2

Example FSM

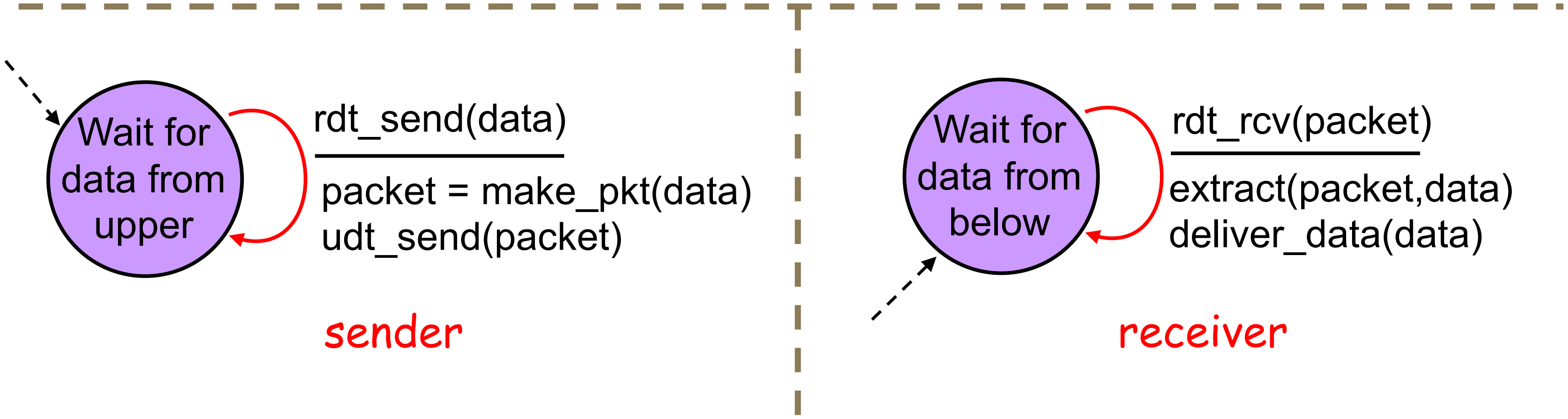


rdt 1.0: Perfectly Reliable Channel

Assume underlying channel is perfectly reliable.

Separate FSMs for sender, receiver:

- Sender sends data into underlying (perfect) channel
- Receiver reads data from underlying (perfect) channel



rdt 2.0: Channel with Bit Errors

Assumption:

- underlying channel may flip bits in packets
- other than that, the channel is perfect

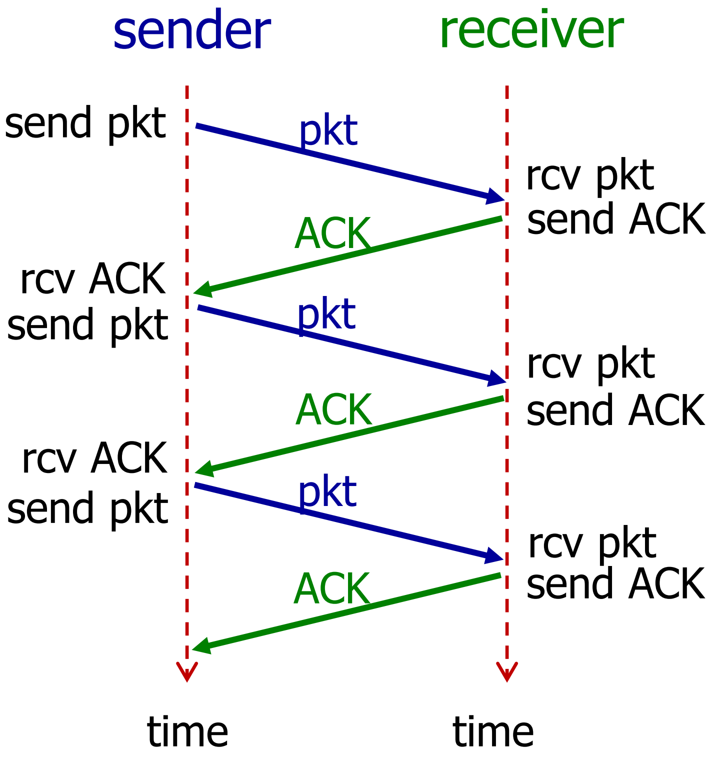
Q1: how to detect bit errors?

- Receiver may use ***checksum*** to detect bit errors.

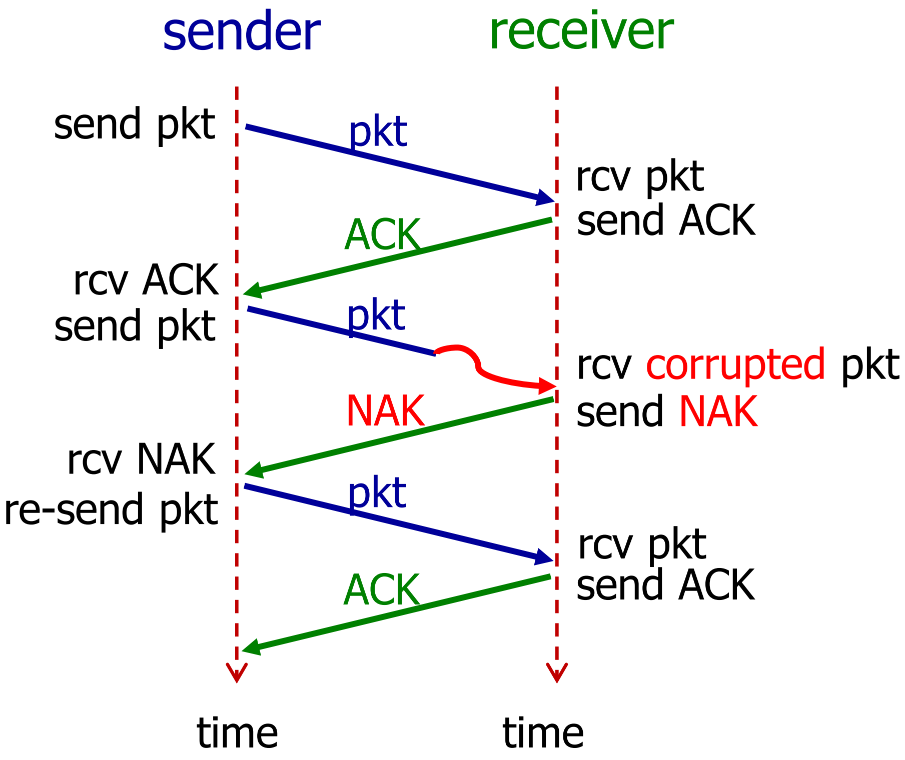
Q2: how to recover from bit errors?

- *Acknowledgements (ACKs):* receiver explicitly tells sender that packet received is OK.
- *Negative acknowledgements (NAKs):* receiver explicitly tells sender that packet has errors.
  - Sender retransmits packet on receipt of NAK.

rdt 2.0 In Action



(a) no bit error

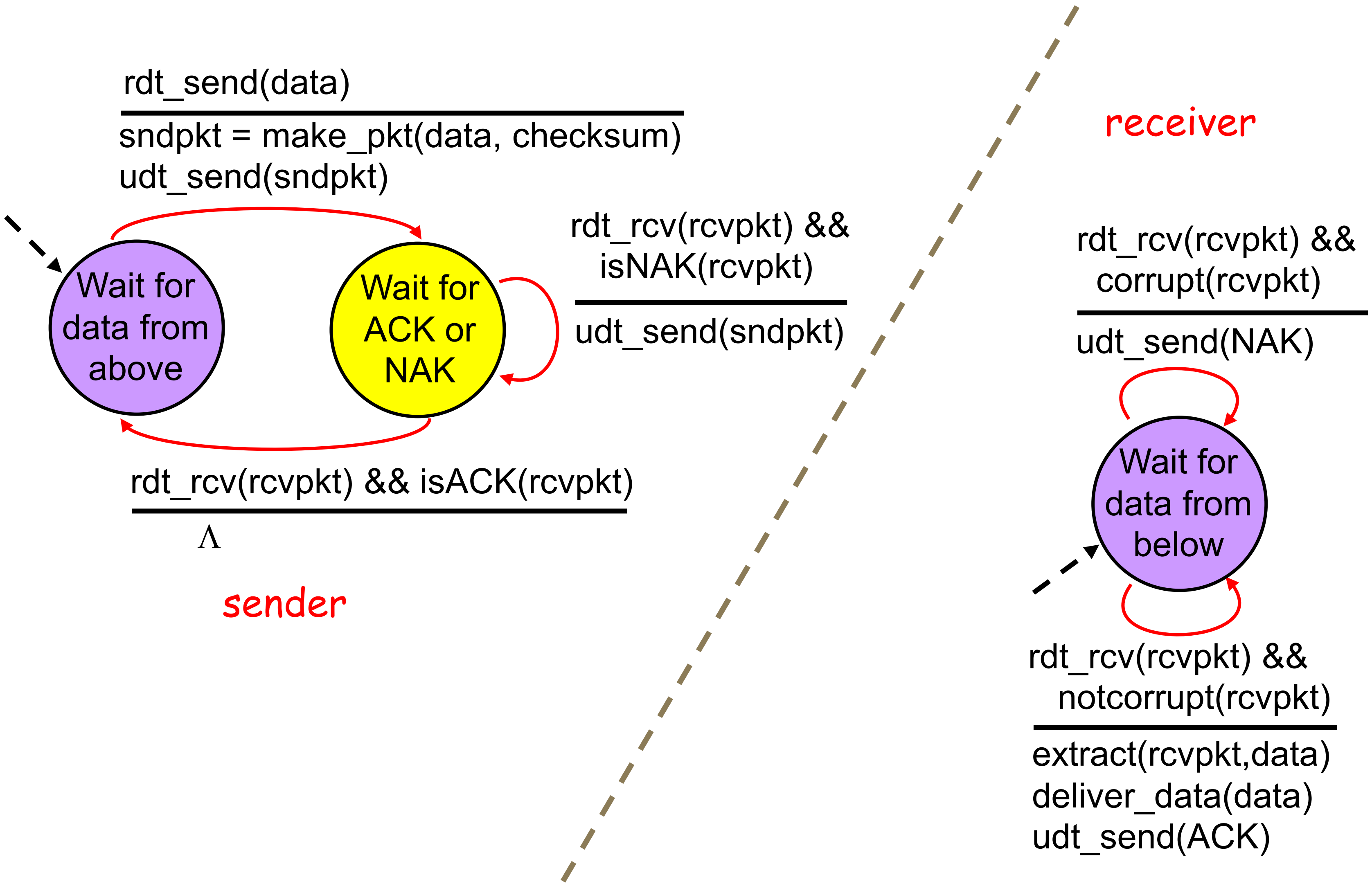


(b) with bit error

stop and wait protocol

Sender sends one packet at a time, then waits for receiver response

rdt 2.0: FSM



rdt_send(data)

sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for data from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

receiver

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

Wait for data from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

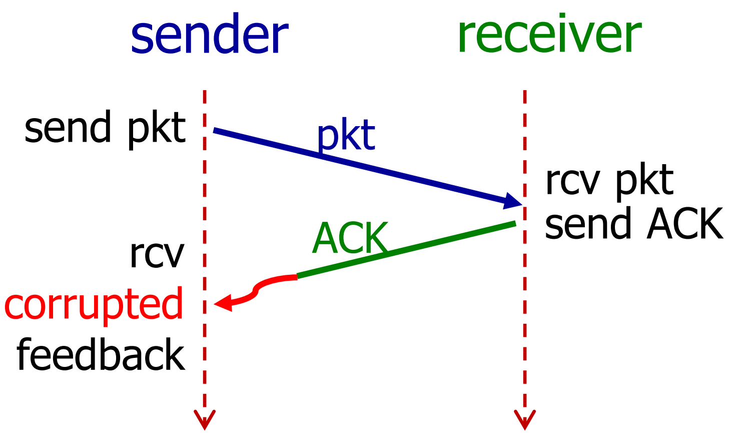
rdt 2.0 has a Fatal Flaw!

What happens if ACK/NAK is corrupted?

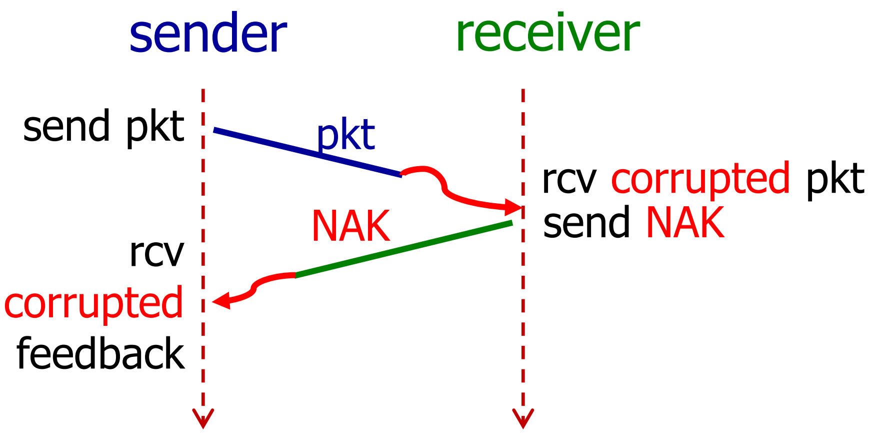
- Sender doesn't know what happened at receiver!

So what should the sender do?

- Sender just retransmits when receives garbled ACK or NAK.
- Questions: does this work?



a) corrupted ACK

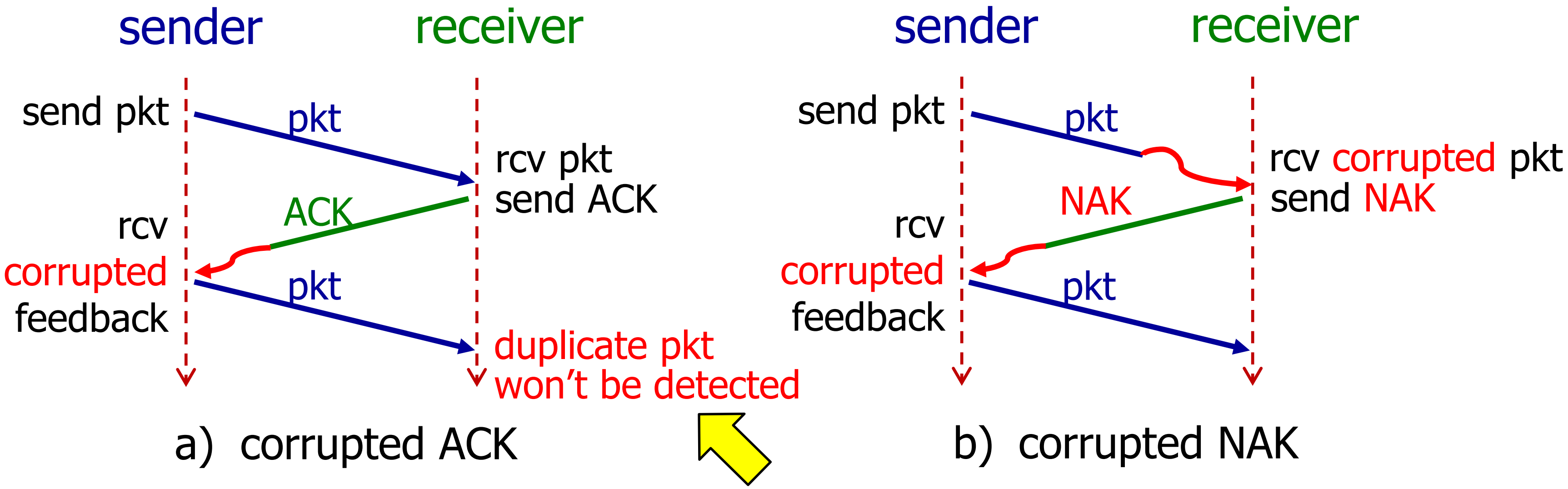


b) corrupted NAK

rdt 2.0 has a Fatal Flaw!

Sender just retransmits when it receives garbled feedback.

- This may cause retransmission of correctly received packet!
- Question: how can receiver identify duplicate packet?



a) corrupted ACK

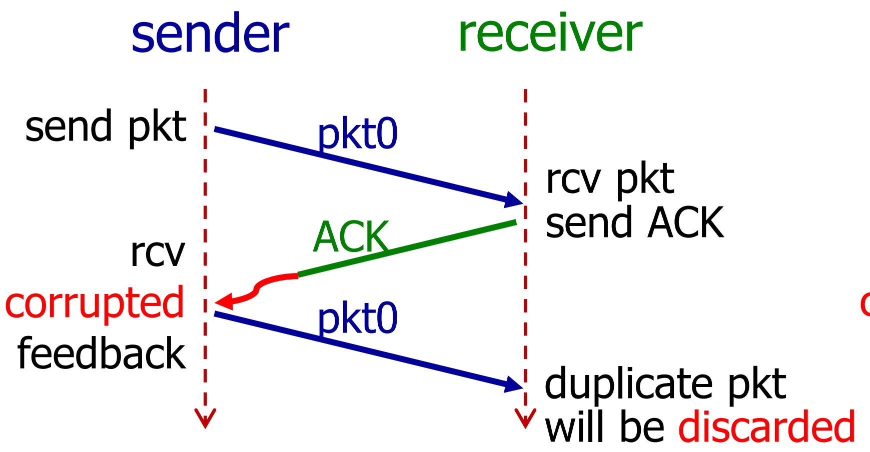
b) corrupted NAK

rdt 2.1: rdt 2.0 + Packet Seq. #

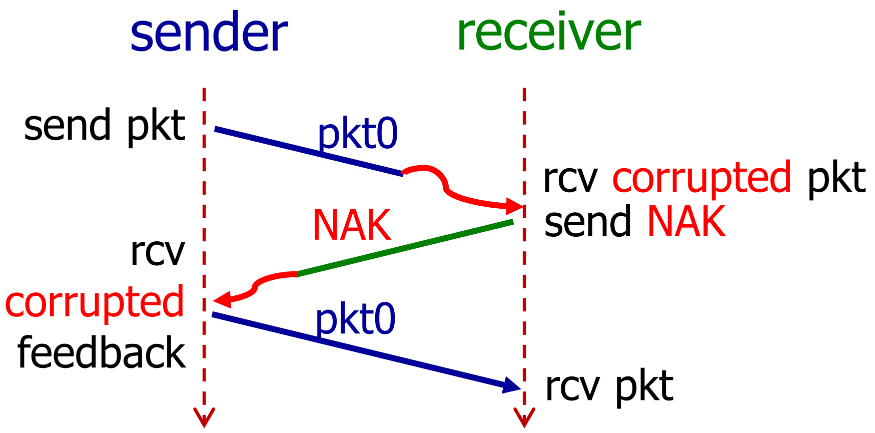
To handle duplicates:

- Sender retransmits current packet if ACK/NAK is garbled.
- Sender adds *sequence number* to each packet.
- Receiver discards (doesn't deliver up) duplicate packet.

This gives rise to protocol rdt 2.1.

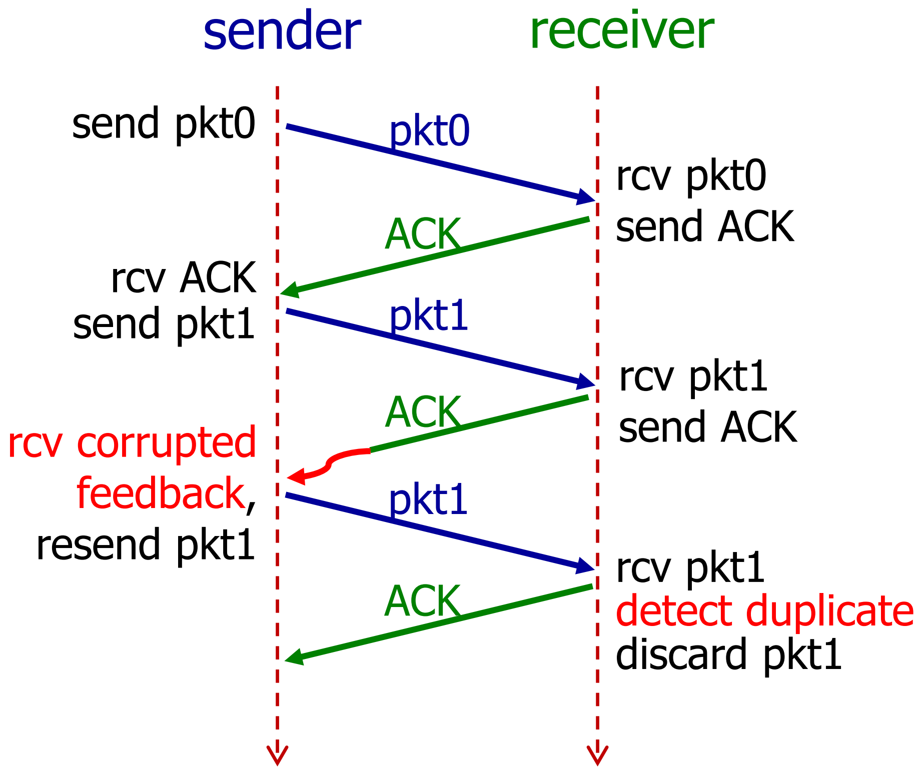


a) corrupted ACK

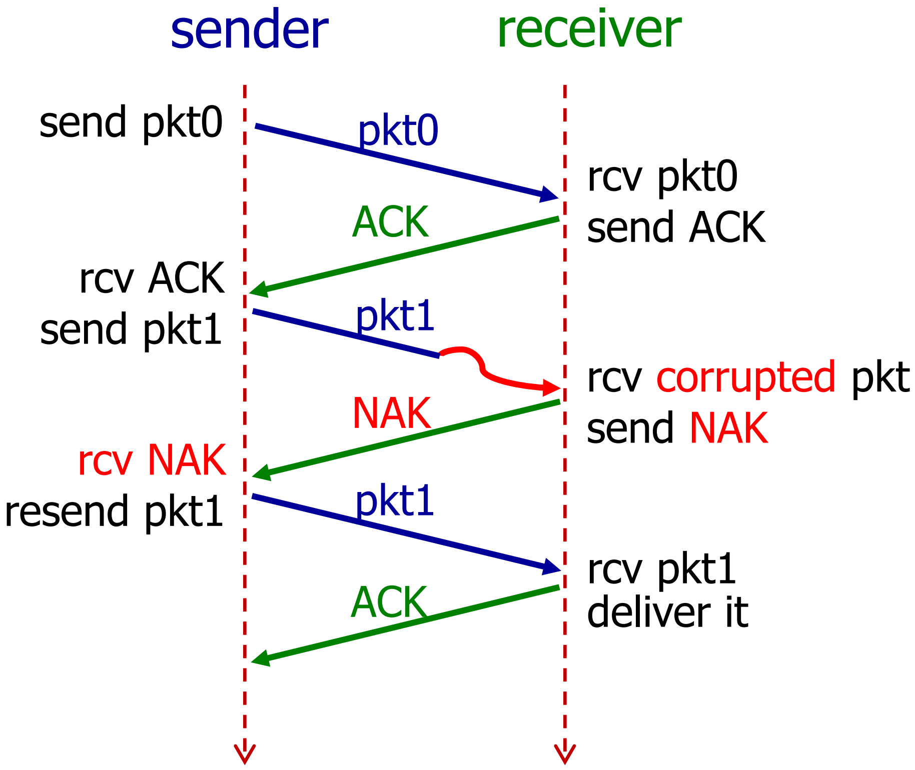


b) corrupted NAK

rdt 2.1 In Action

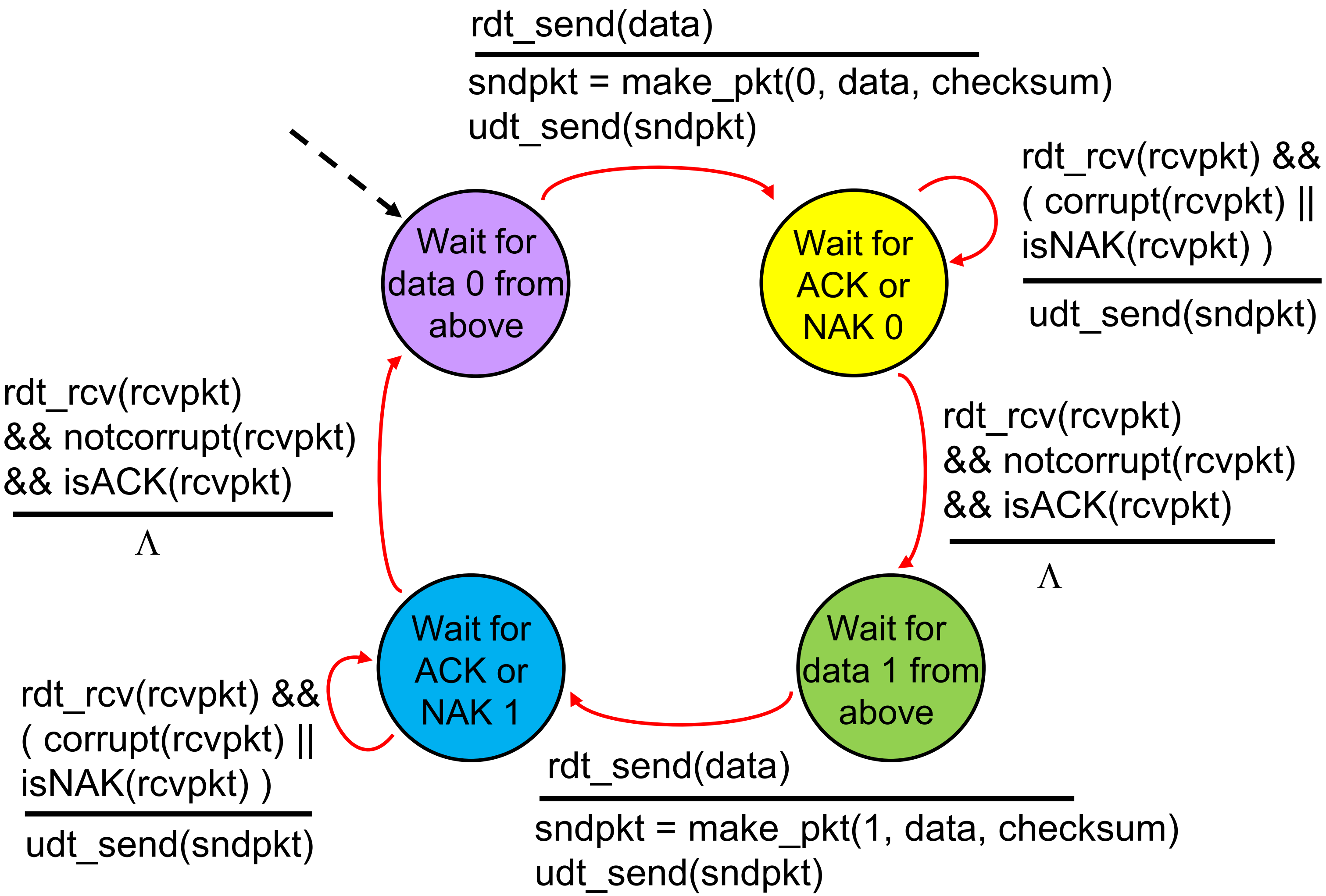


a) resend due to corrupted ACK



b) resend due to corrupted packet

rdt 2.1 Sender FSM



rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

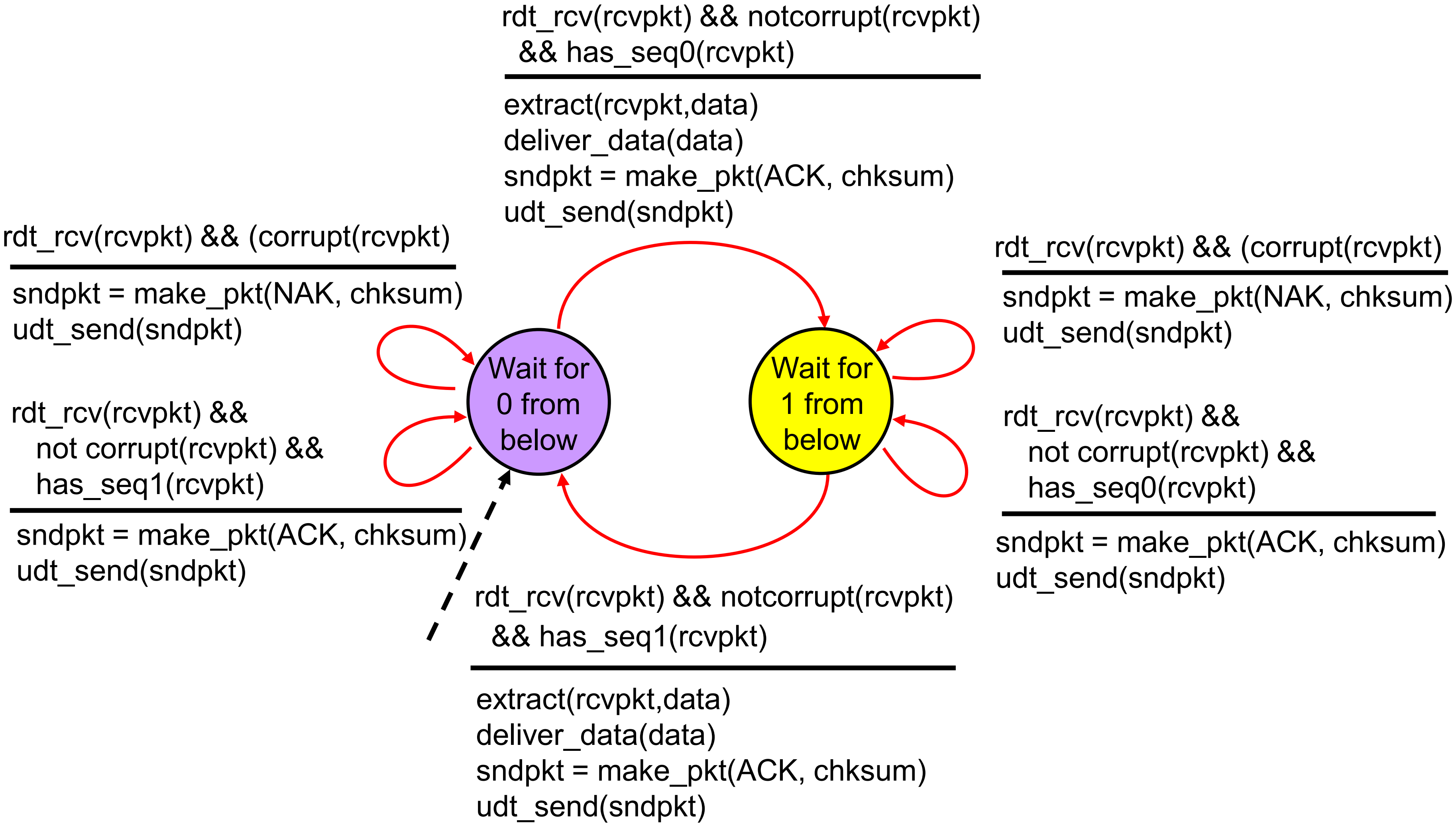
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt 2.1 Receiver FSM



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt 2.2: a NAK-free Protocol

- Same assumption and functionality as rdt 2.1, but use ACKs only.

- Instead of sending NAK, receiver sends ACK for the last packet received OK.
    - Now receiver must *explicitly* include seq. # of the packet being ACKed.

- Duplicate ACKs at sender results in same action as NAK: *retransmit current pkt*.

rdt 3.0: Channel with Errors and Loss

Assumption: underlying channel

- may flip bits in packets
- may lose packets
- may incur arbitrarily long packet delay
- but won’t re-order packets

Question: how to detect packet loss?

- checksum, ACKs, seq. #, retransmissions will be of help… but not enough

rdt 3.0: Channel with Errors and Loss

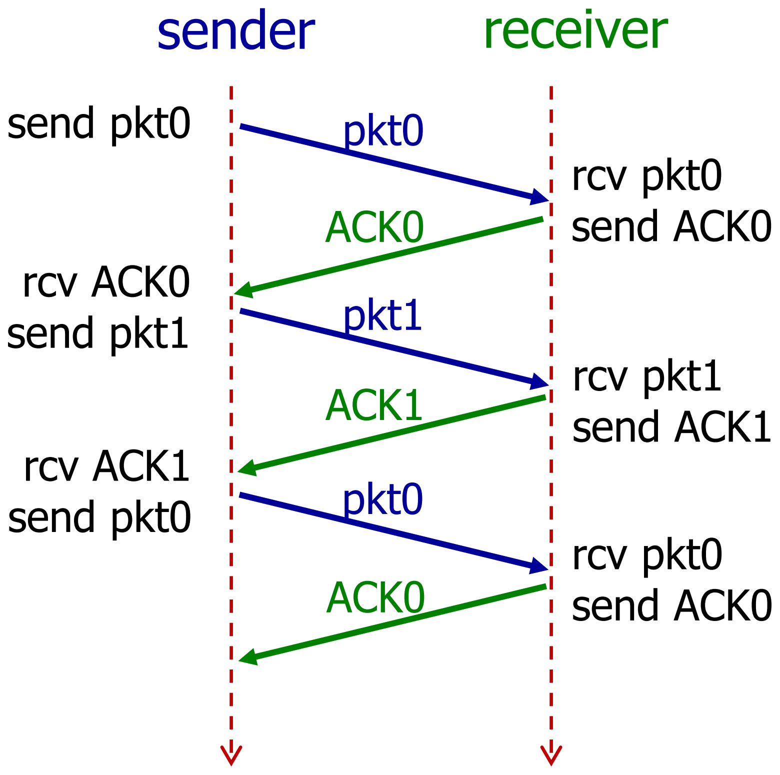
To handle packet loss:

- Sender waits “reasonable” amount of time for ACK.
- Sender retransmits if no ACK is received till *timeout*.

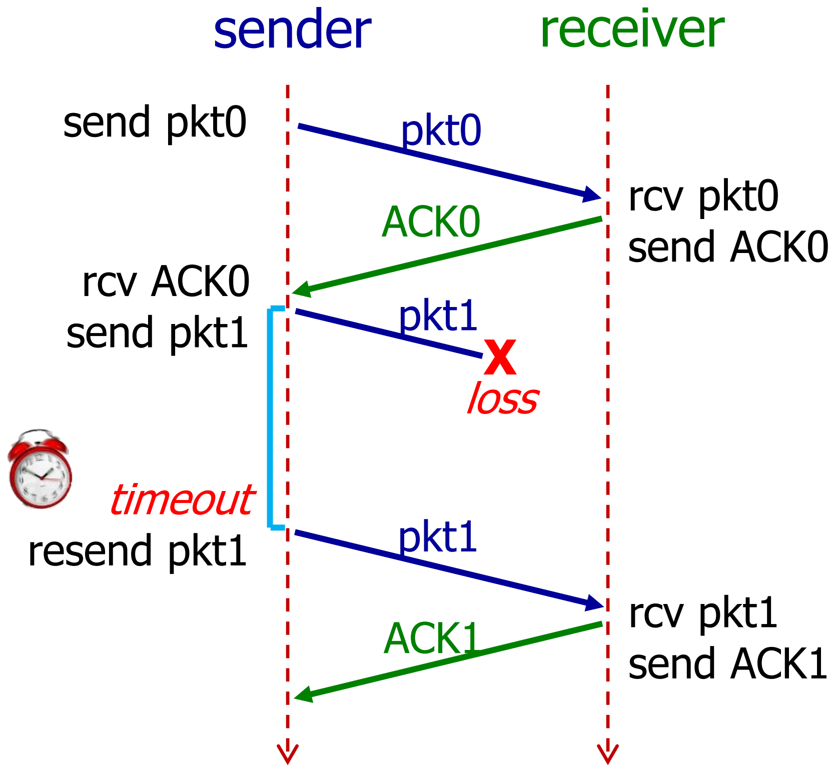
Question: what if packet (or ACK) is just delayed, but not lost?

- Timeout will trigger retransmission.
- Retransmission will generate duplicates in this case, but receiver may use seq. # to detect it.
- Receiver must specify seq. # of the packet being ACKed (check scenario (d) two pages later).

rdt 3.0 In Action

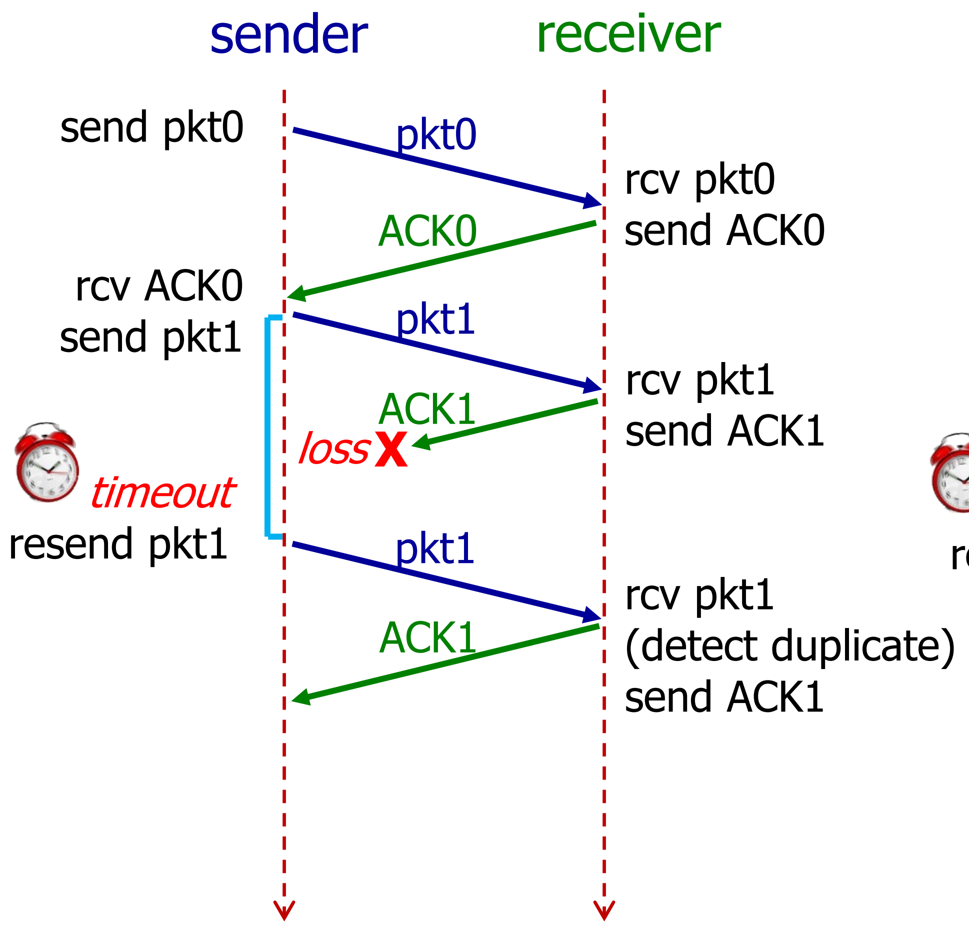


a) no packet loss

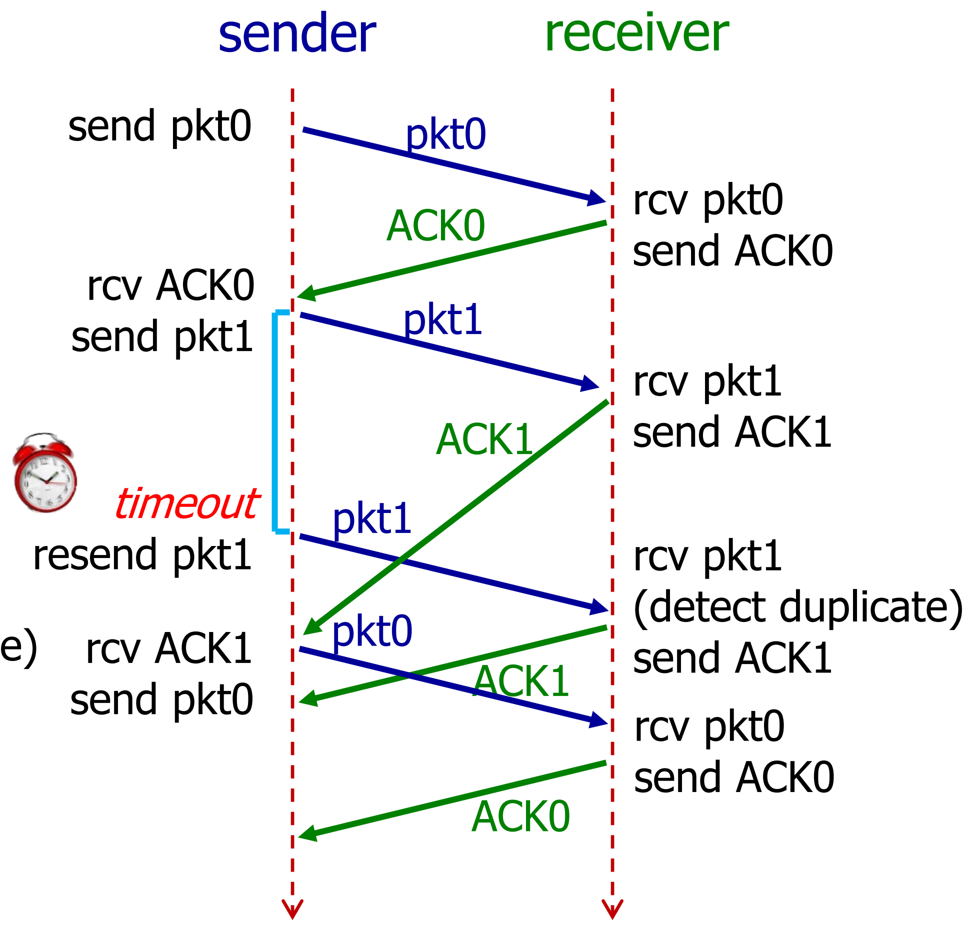


b) packet loss

rdt 3.0 In Action

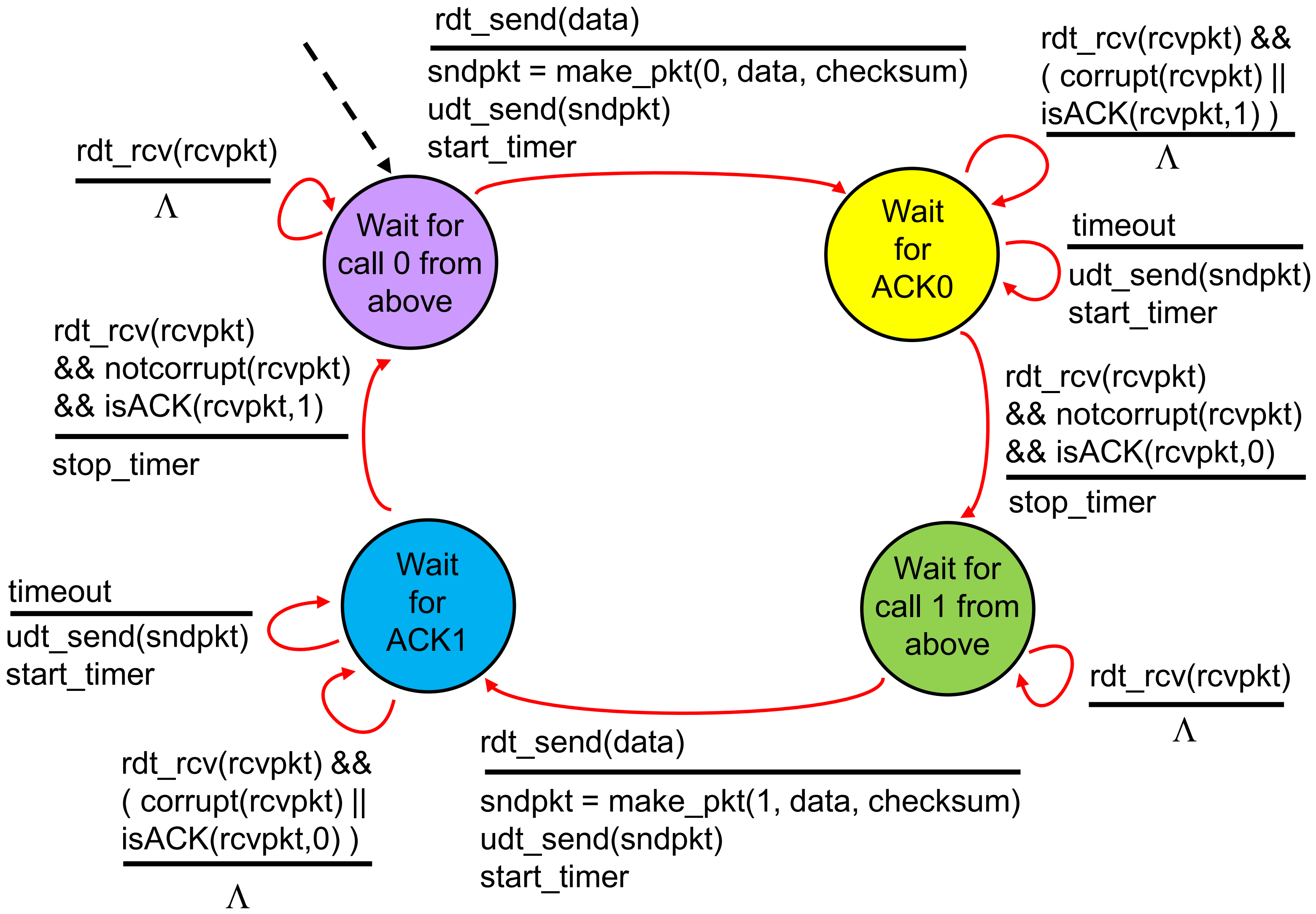


c) lost ACK



d) premature timeout / delayed ACK

rdt 3.0 Sender FSM



rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ

timeout
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer

rdt_rcv(rcvpkt)
Λ

rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ

timeout
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer

rdt_rcv(rcvpkt)
Λ

RDT Summary

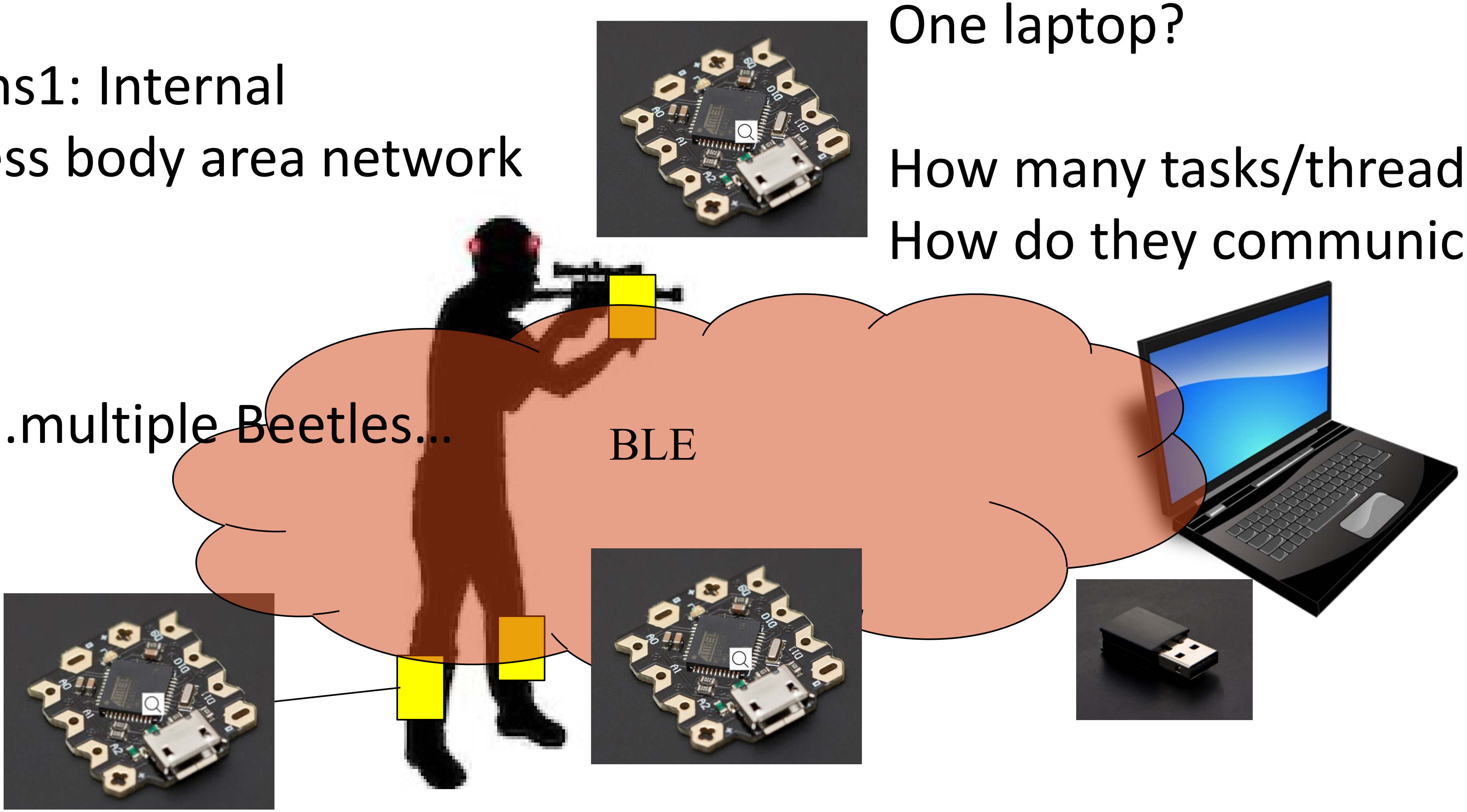
rdt Version	Scenario	Features Used
1.0	no error	nothing
2.0	data Bit Error	checksum, ACK/NAK
2.1	data Bit Error ACK/NAK Bit Error	checksum, ACK/NAK, sequence Number
2.2	Same as 2.1	NAK free
3.0	data Bit Error ACK/NAK Bit Error packet Loss	checksum, ACK, sequence Number, timeout/re-transmission

Designing your own protocol over BLE

- Handshaking: What do you send? Who starts handshaking?
- Packet format: What data do you send, in what format?
  - BLE: Max message size? **What if data is fragmented across multiple messages?**
  - Baud rate?
- Reliability?
- Concurrency?
- Security?

Comms Internal:

Comms1: Internal
wireless body area network



...multiple Beetles...

BLE

How many Beetles?

How many sensors per Beetle?

One laptop?

How many tasks/threads/processes?

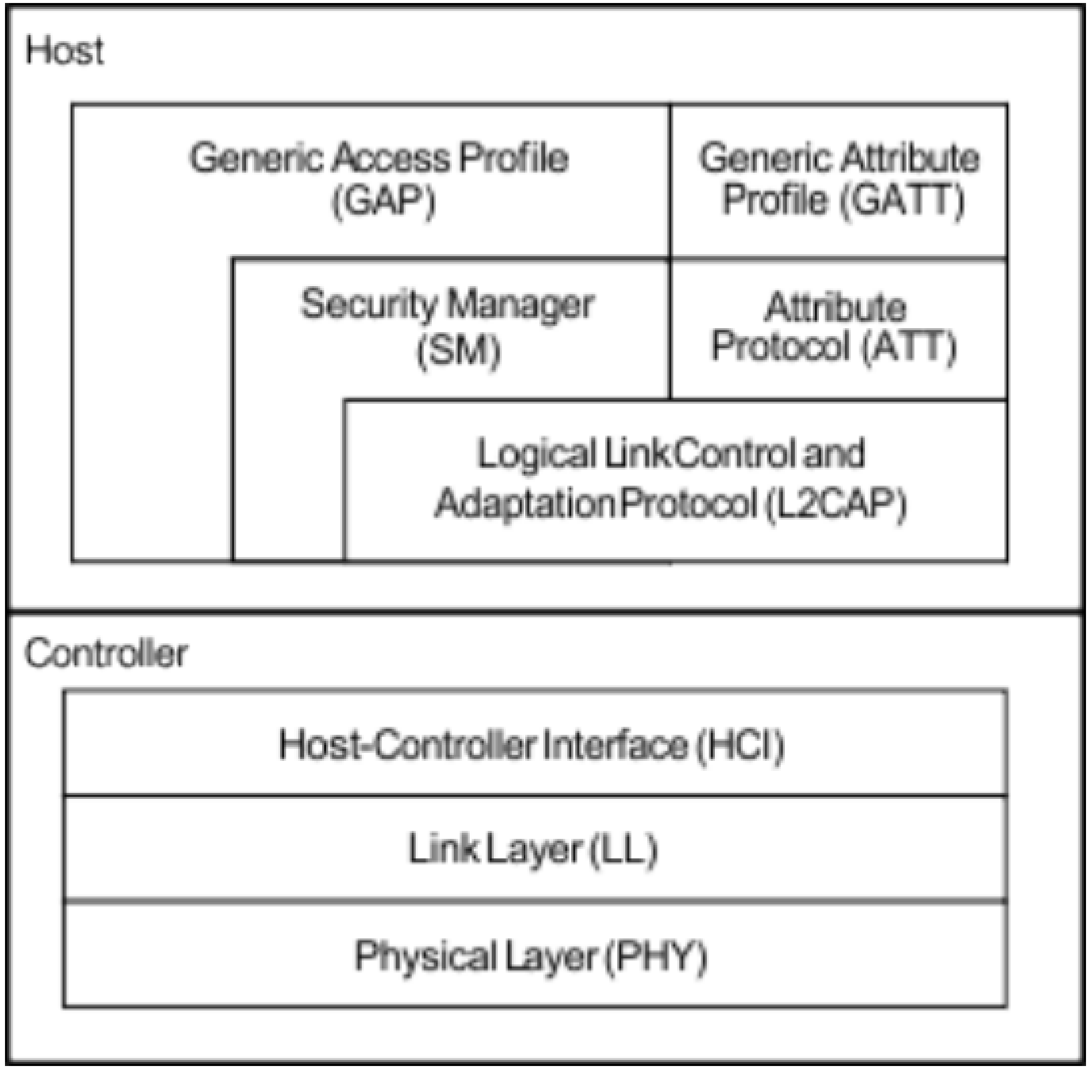
How do they communicate?

ARDUINO Beetle-Laptop SERIAL Communications

Bluetooth Low Energy (BLE)

- Targeted for low power devices, IoT, wearables, mobiles
- Widely adopted
- Small data size, low duty cycle
- Range

BLE Host and Controller



Host

Generic Access Profile (GAP)

Generic Attribute Profile (GATT)

Security Manager (SM)

Attribute Protocol (ATT)

Logical Link Control and Adaptation Protocol (L2CAP)

Controller

Host-Controller Interface (HCI)

Link Layer (LL)

Physical Layer (PHY)

[TI CC2540 software developer’s guide]

Setting up BLE host and controller on Ubuntu Linux

`hciconfig`

- print information about Bluetooth devices installed in the system.

`/dev/wilc_bt:`

- `echo BT_POWER_UP > /dev/wilc_bt`
- `echo BT_DOWNLOAD_FW > /dev/wilc_bt`
- `echo BT_FW_CHIP_WAKEUP > /dev/wilc_bt`

`hciattach /dev/ttyPS1 -t 10 any 115200 noflow nosleep`

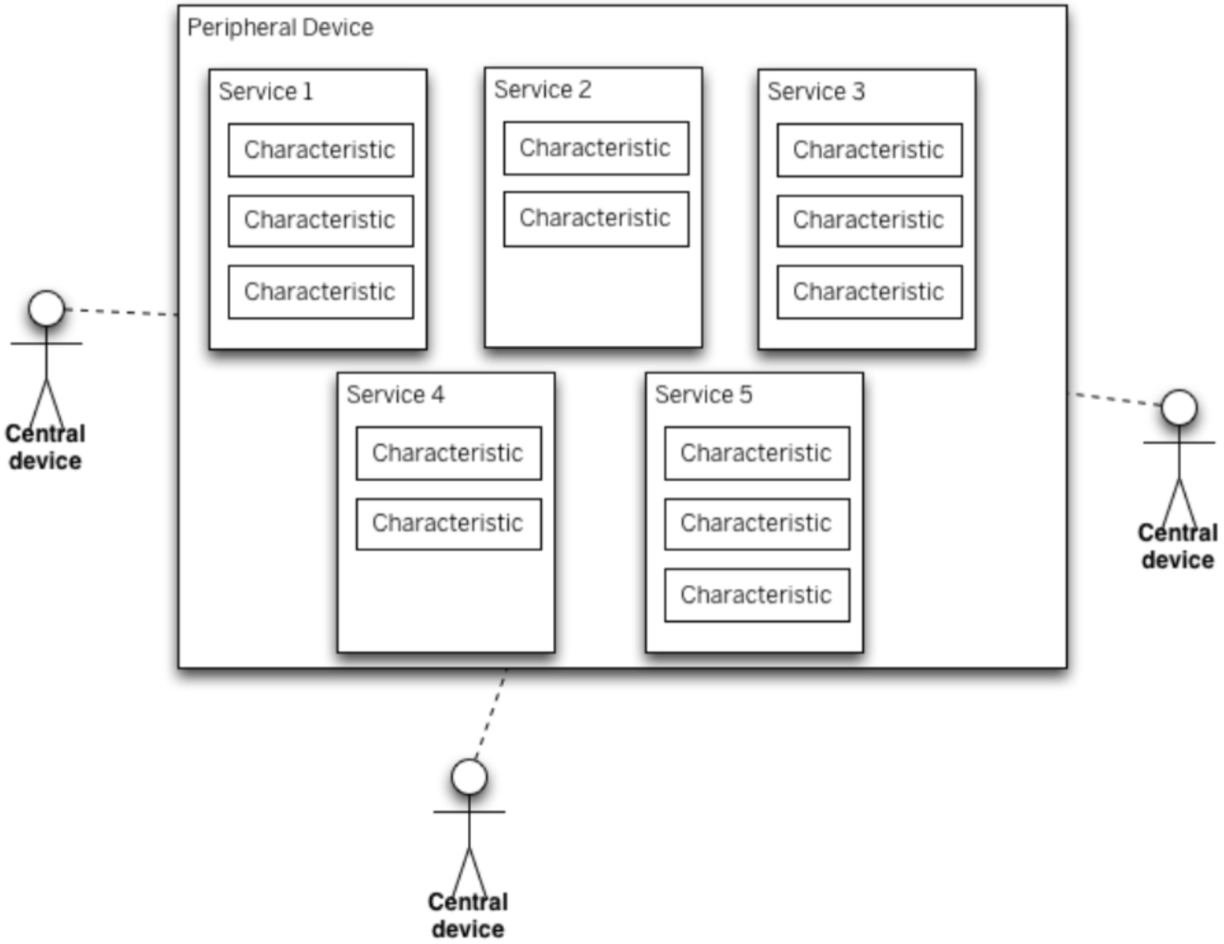
- attach serial UART to bluetooth stack as HCI transport interface

Configure conn_min_interval and conn_max_interval settings

`bluetoothctl`

- commands: list, show, connect
- Get UUID

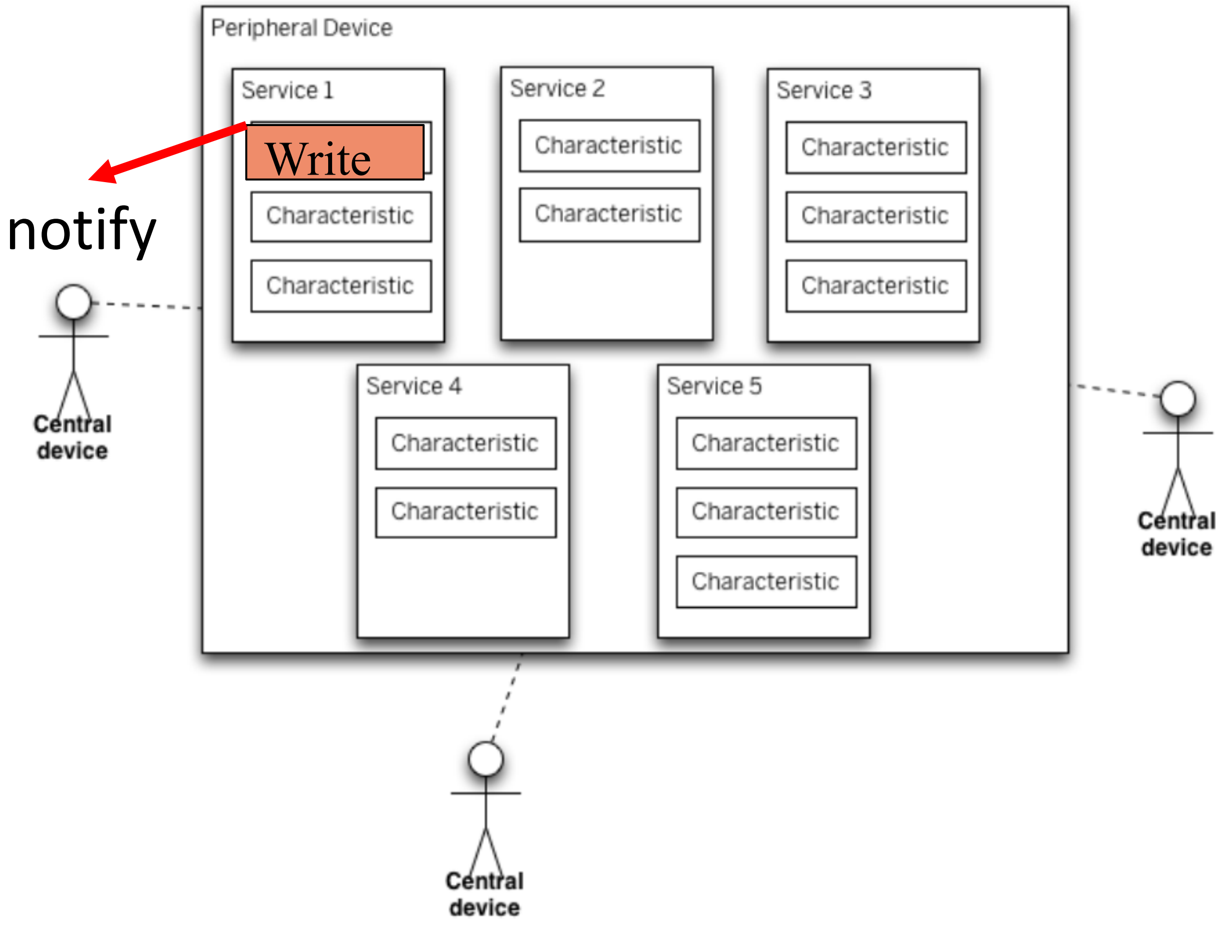
BLE basics



- Peripherals vs Central devices
  - Peripherals publish/write data
  - Central subscribes/reads data

- Service and characteristics

BLE basics: notifications



How to establish BLE connections?

- Connection = Peripheral – Central can communicate

- Discovery and advertising
  - Central device can scan and look for new devices
  - Do you need it?

- Handshaking
  - Need to make sure both devices are awake so you can establish connection
  - How will you handshake?

BLE on Beetle: Serial Programming

```
Serial.begin

Serial.available

Serial.read

Serial.print
```

BLE on Ubuntu: bluepy? pySerial? bluez?

To do serial programming using Python you can use the bluepy package.

◦ `github.com/IanHarvey/bluepy`

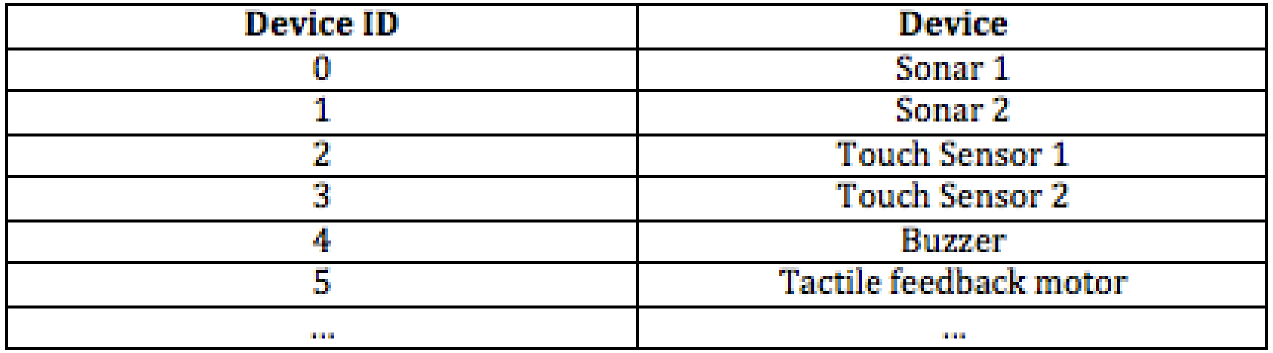
Sample code skeleton

```
from bluepy import btle

dev = btle.Peripheral("B0:B4:48:BF:C9:83")
```

Assign an ID to each device

You need to be able to identify sensors (actuators) to read from (send data to).

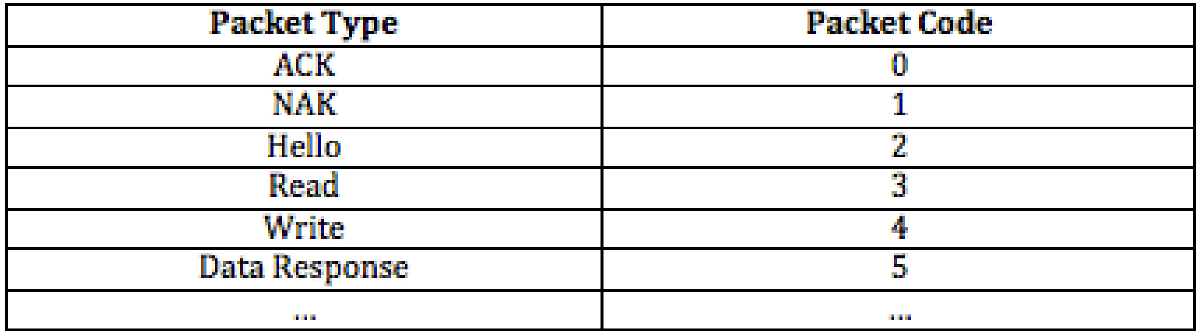


Device ID	Device
0	Sonar 1
1	Sonar 2
2	Touch Sensor 1
3	Touch Sensor 2
4	Buzzer
5	Tactile feedback motor
...	...

Do you have more than one sensor connected to a Beetle?

Create Packet Types

So both sides know what sort of packets are being sent (and the appropriate response)

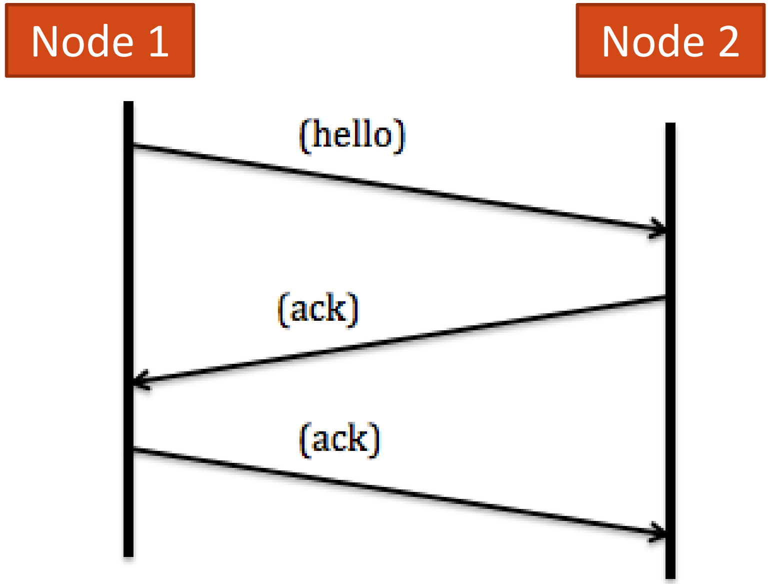


Packet Type	Packet Code
ACK	0
NAK	1
Hello	2
Read	3
Write	4
Data Response	5
...	...

Bootup 3-way Handshake

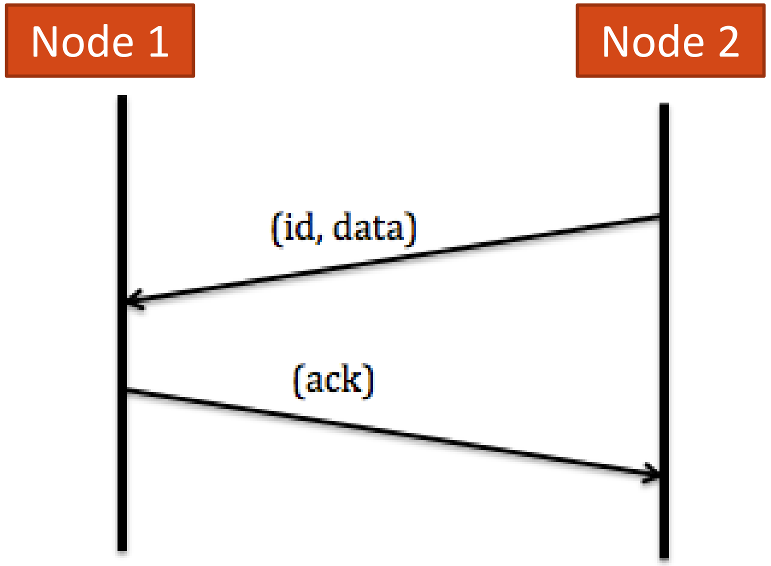
Objective:

- So both beetles and laptop know that each is ready to communicate.



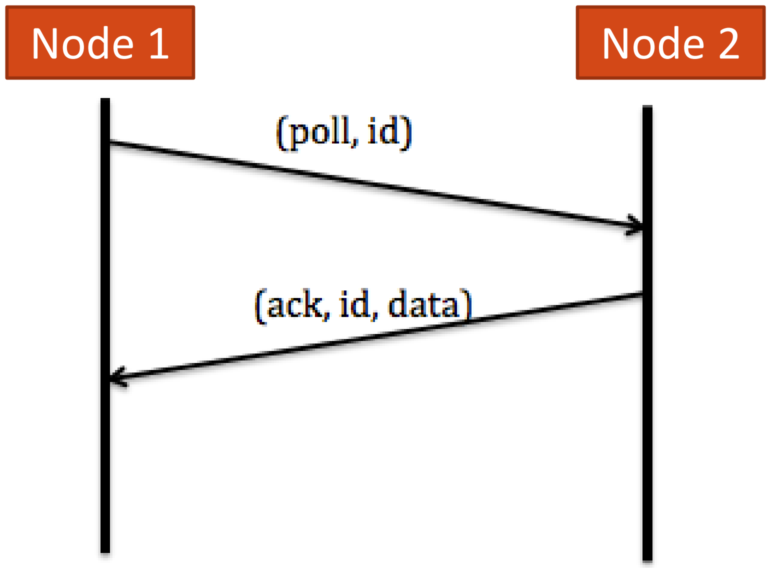
- Do this at the very start of your programs on both sides

Periodic Push By Arduino?



- Arduino sends data whenever it is available.
- Laptop monitors and buffers data as it comes in.
  - + Arduino sends data whenever it is available.
  - - Laptop needs to buffer incoming data.

Periodic Poll by Laptop



- Arduino waits for poll packets from laptop
- Laptop requests data when it needs it.

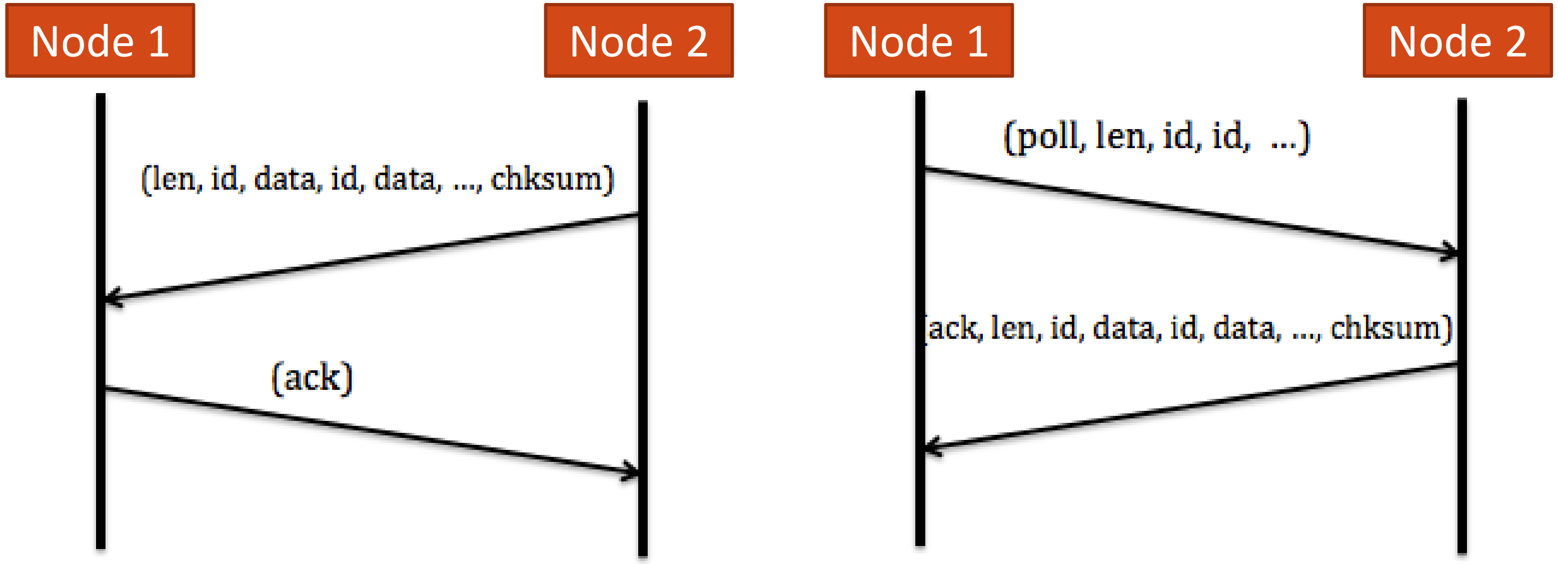
  + Laptop decides when it needs the data and sends poll packet.

  - If laptop doesn't poll often enough, may lose data on Arduino (Arduino has small memory).

Sending Raw or Processed Data?

Polling/Pushing individual sensor data can be expensive.

Might be better to send processed?



Node 1

Node 2

(len, id, data, id, data, ..., chksum)

(ack)

Node 1

Node 2

(poll, len, id, id, ...)

(ack, len, id, data, id, data, ..., chksum)

Reliability: Checksums, Reconnections, Fragmentation

- Checksums are used to check that data is received correctly.
  - Does BLE specs support checksum?
- Disconnections and reconnections
- Packet fragmentation

Concurrency: Tasks and processes in our project

Arduino: RTOS?

- What are the tasks?
- Priorities among the tasks?

Laptop:

- What are the processes? Or threads?
- Synchronization/communication between the threads/processes?

Goal: Send sensor data from Beetles to laptop reliably

Burning questions…

- ***Beetle:***
  - How to connect wirelessly?
  - How to handshake?
  - How to send?
  - Real-time OS?
- ***Laptop***:
  - How to discover the beetles?
  - How to handshake?
  - How to receive from multiple beetles?
  - How to ensure reliable communication?

Individual subcomponent test

Comms Internal

- Walkthrough protocol for BLE communications
  - Handshaking
  - Packet format
- Dummy sensor data
- Demonstrate concurrent BLE connections from 3 Beetles to laptop lasting at least a minute
- Demonstrate connection loss recovery