Advanced Network Technologies

Application layer

Transport layer

Dr. Wei Bao| Lecturer
School of Computer Science

THE UNIVERSITY OF SYDNEY

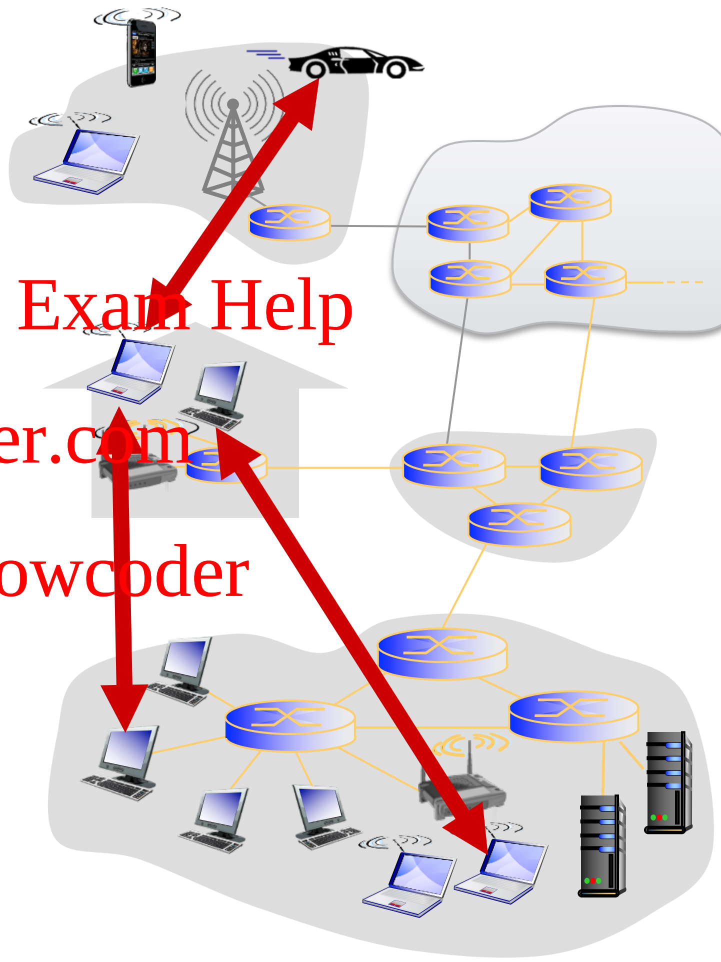
Peer-to-Peer

Pure peer-to-peer model architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

examples:

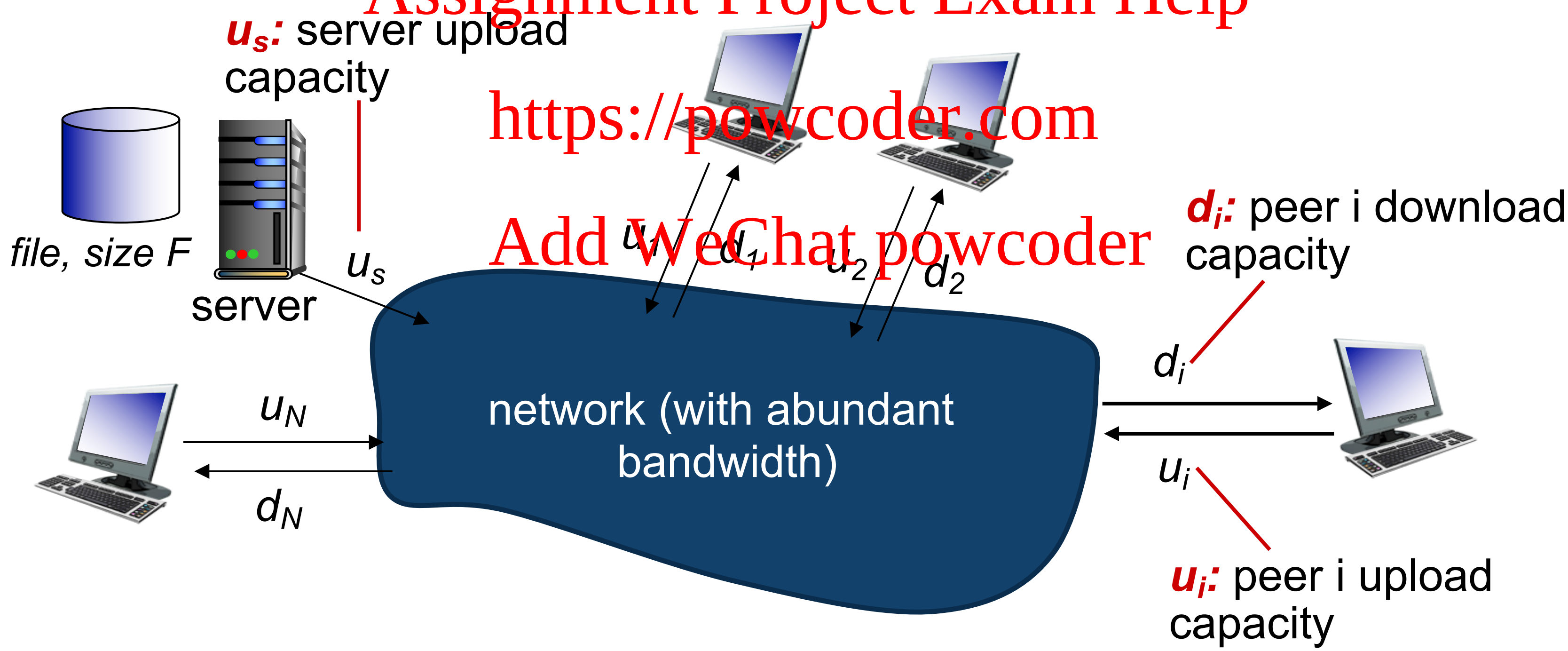
- file distribution (BitTorrent)
- Streaming (Zattoo, KanKan)
- VoIP (Skype)



File distribution: client-server vs. p2p

Question: how much time to distribute file (size *F*) from one server to *N* peers?

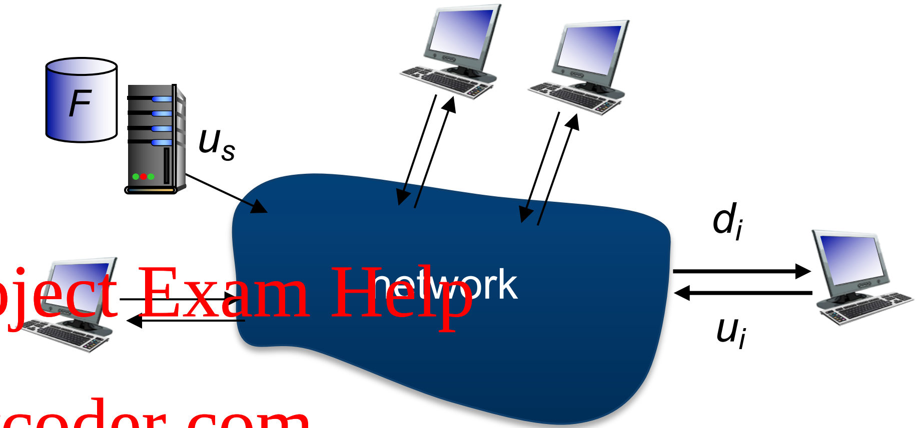
- peer upload/download capacity is limited resource



THE UNIVERSITY OF SYDNEY

File distribution time: client-server

- *server transmission:* must sequentially send (upload) N file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} = min client download rate
  - (worst case) client download time: F/d_{min}

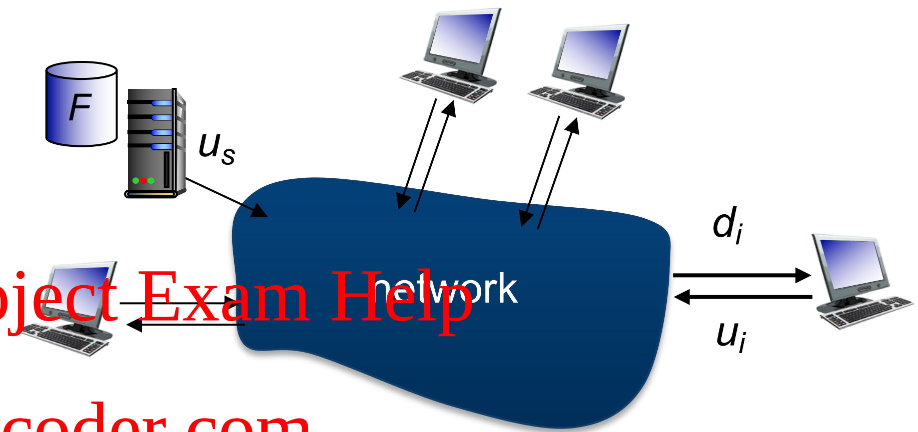


time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_{s,},F/d_{min}\}$$

increases linearly in N

File distribution time: p2p



- *server transmission:* must upload at least one copy
  - time to send one copy: F/u_s
    - *client:* each client must download file copy
      - client download time: F/d_{min}
    - *clients:* as aggregate must download *NF* bits = upload *NF* bits
      - Max upload rate $u_s + \sum u_i$
      - $NF/(u_s + \sum u_i)$

time to distribute F to N clients using P2P approach

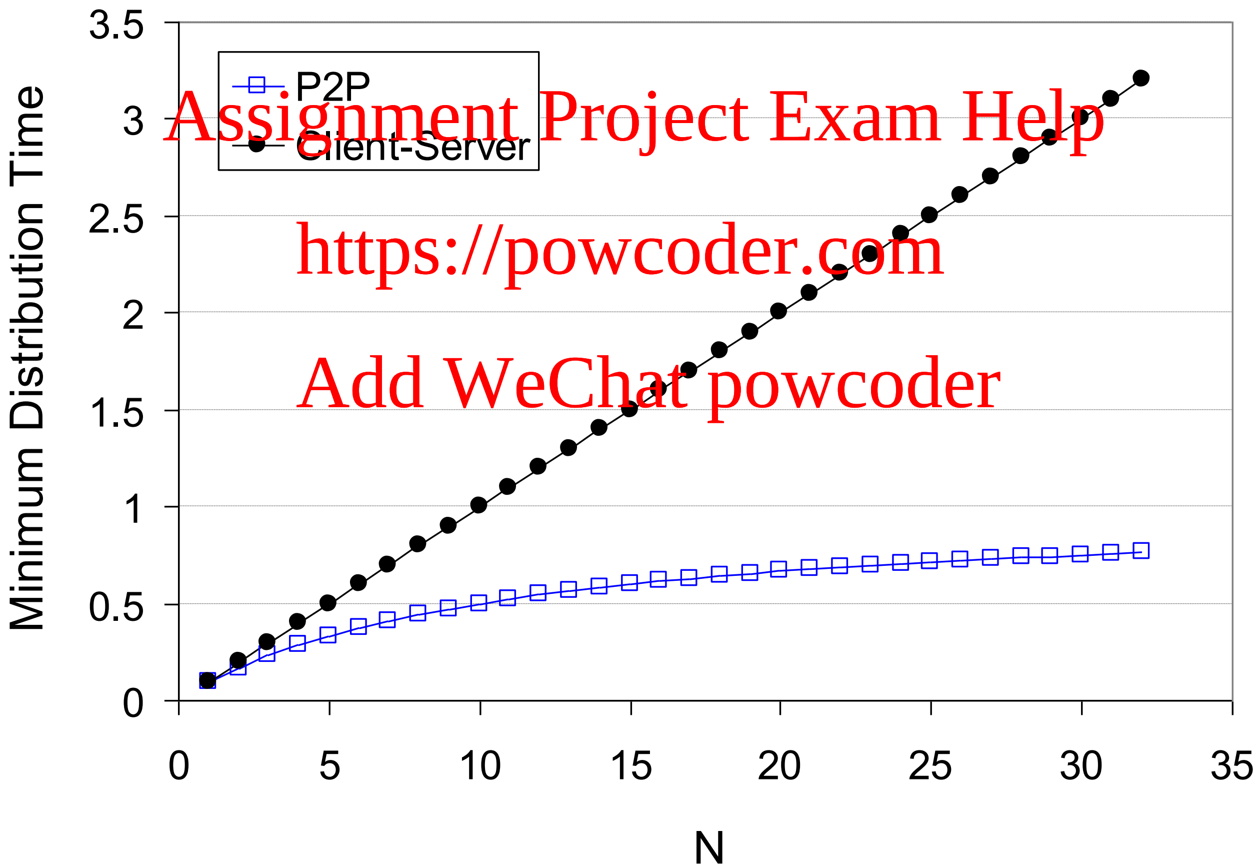
$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

Client-server vs. p2p

client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$



P2P file distribution: BitTorrent

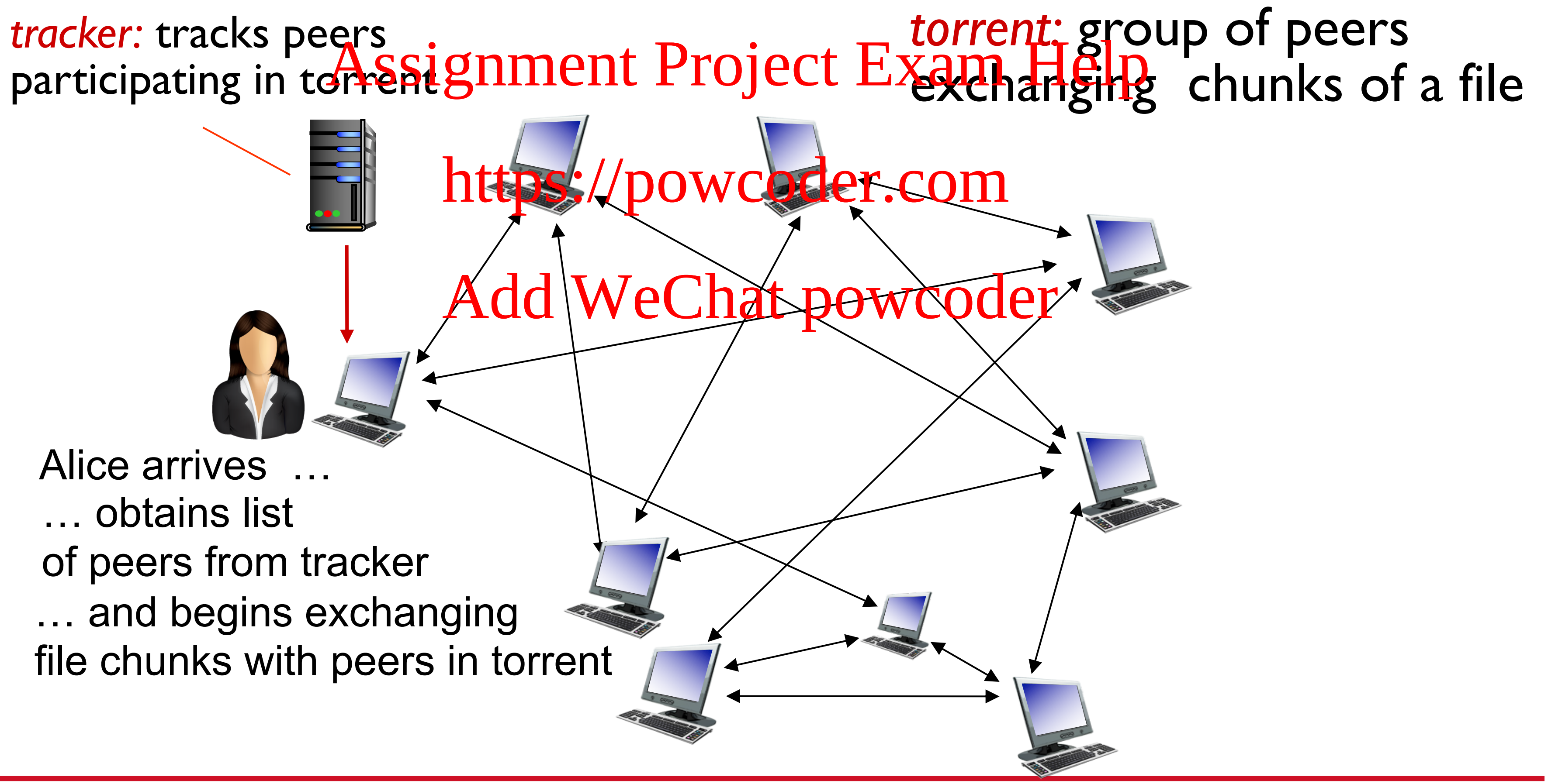
BitTorrent, a file sharing application

- 20% of European internet traffic in 2012.
- Used for Linux distribution, software patches, distributing movies
- Goal: quickly replicate large files to large number of clients

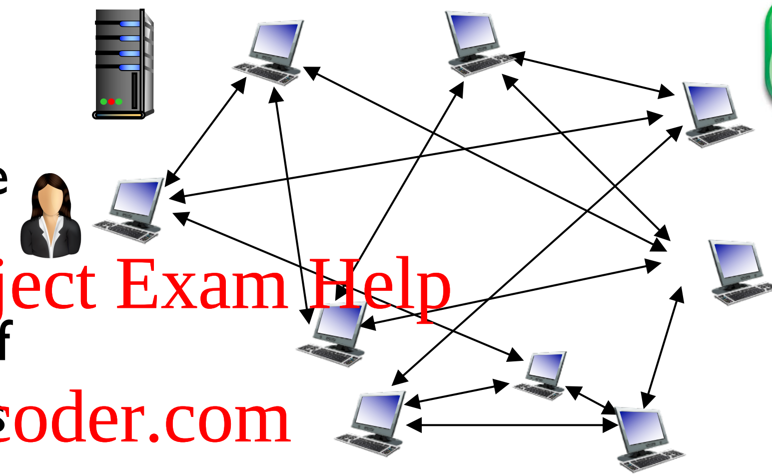
- Web server hosts a .torrent file (w/ file length, hash, tracker's URL…)
- A tracker tracks downloaders/owners of a file
- Files are divided into chunks (256kB-1MB)
- Downloaders download chunks from themselves (and owners)
- Tit-for-tat: the more one shares (server), the faster it can download (client)

P2P file distribution: BitTorrent

- file divided into 256KB chunks
- peers in torrent send/receive file chunks



P2P file distribution: BitTorrent



- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")

- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

sending chunks: tit-for-tat

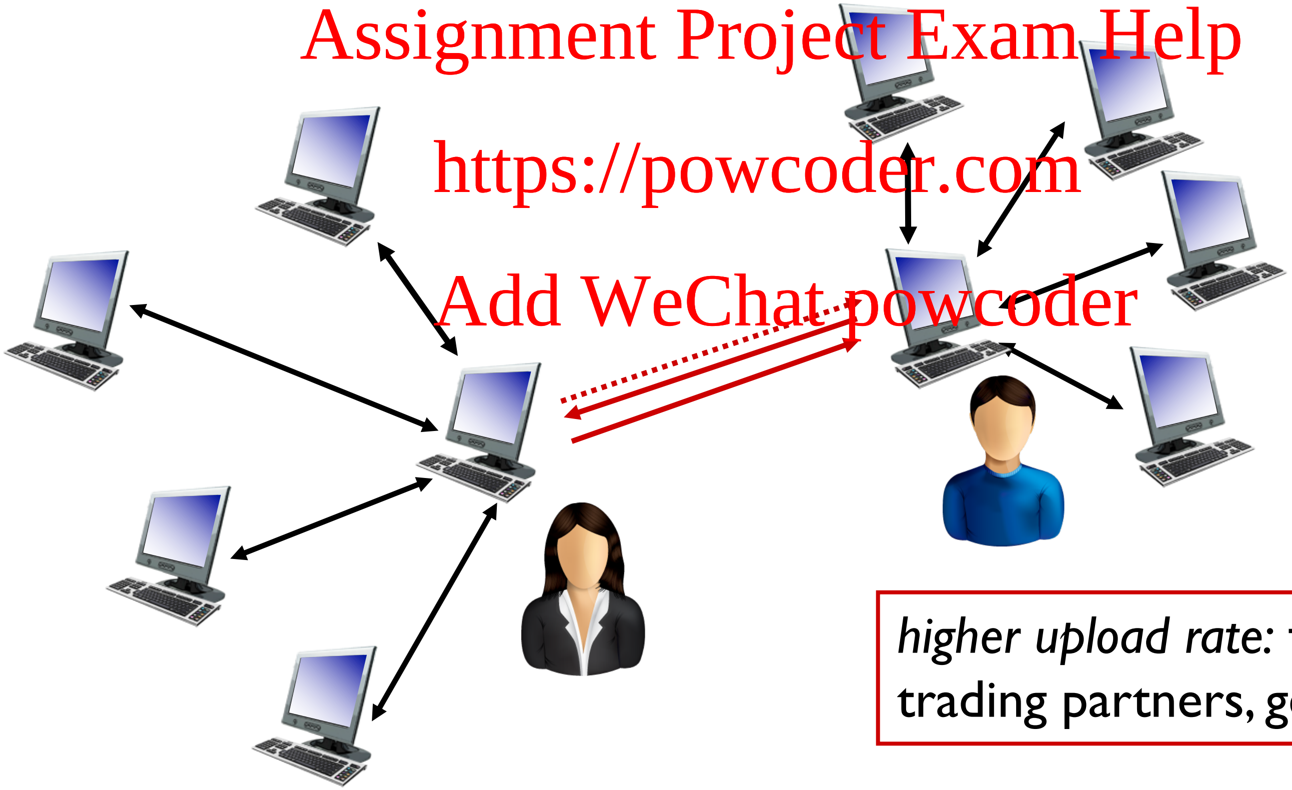
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



higher upload rate: find better trading partners, get file faster !

Distributed Hash Table

Distributed hash table (DHT)

- DHT: a *distributed P2P database*
- database has (key, value) pairs; examples:
  - key: social security number; value: human name
- distribute the (key, value) pairs over the many peers
- a peer queries DHT with key
  - DHT returns values that match the key
- peers can also insert (key, value) pairs

Distributed hash table (DHT)

- Assign the keys
- Lookup the keys

Assigning key to peer

- central issue:
  - assigning (key, value) pairs to peers.
- basic idea:
  - Key: generate an integer
  - Assign an integer ID to each peer
  - put (key,value) pair in the peer that is closest to the key

Assigning key to peer

- **distance**: assign integer identifier to each peer in range $[0,2^n-1]$ for some *n*.
  - each identifier represented by *n* bits.

- Each key to be an integer in same range $[0,2^n-1]$

- to get integer key, hash original key

  - A hash function is any function that can be used to map data of arbitrary size to data of fixed size (e.g., an integer in $[0,2^n-1]$).
  - e.g., 15 = hash("Led Zeppelin IV")
  - this is why its is referred to as a *distributed "hash" table*

Assigning key to peer

› rule: assign key to the peer that has the *closest* ID.

› Here: closest is the *immediate successor* of the key.

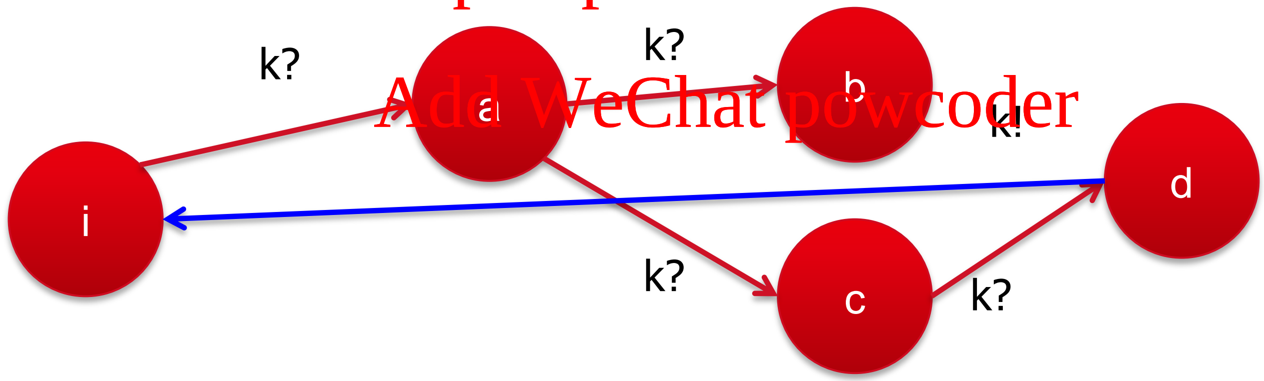
› e.g., $n = 4$; peers: 1, 3, 4, 5, 8, 10, 12, 14;

- key = 13, then successor peer = 14
- key = 15, then successor peer = 1

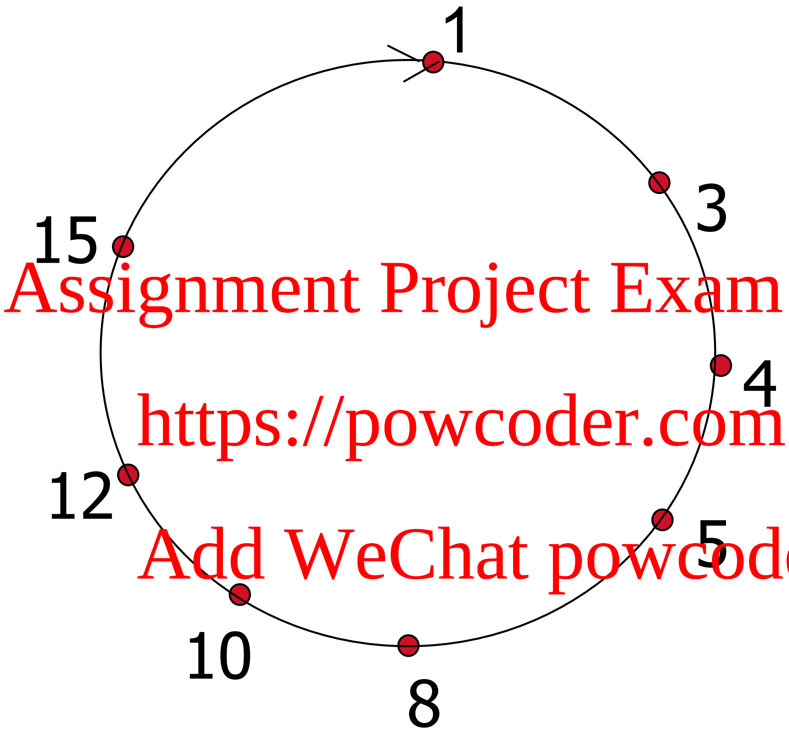
Lookup the keys

Goal: to provide a distributed lookup service returning the host that owns the key

› Given a key, find the host that owns the key



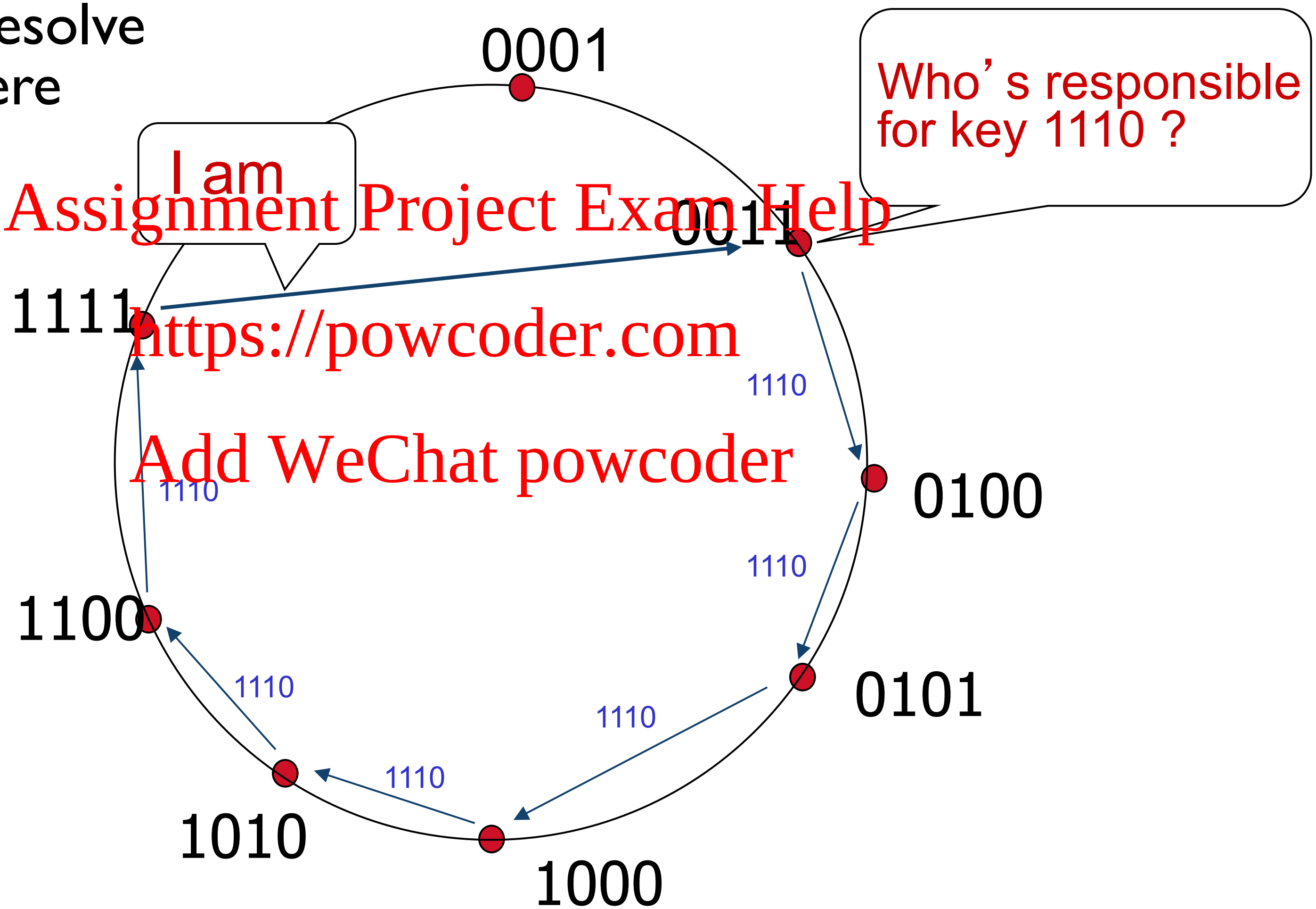
Circular DHT



- each peer *only* aware of immediate successor.

Circular DHT (con't)

O(N) messages on average to resolve query, when there are *N* peers



Define closest as closest successor

Circular DHT (con't)

Example: Chord is an example of a Distributed Hash Table (DHT)

As a node:

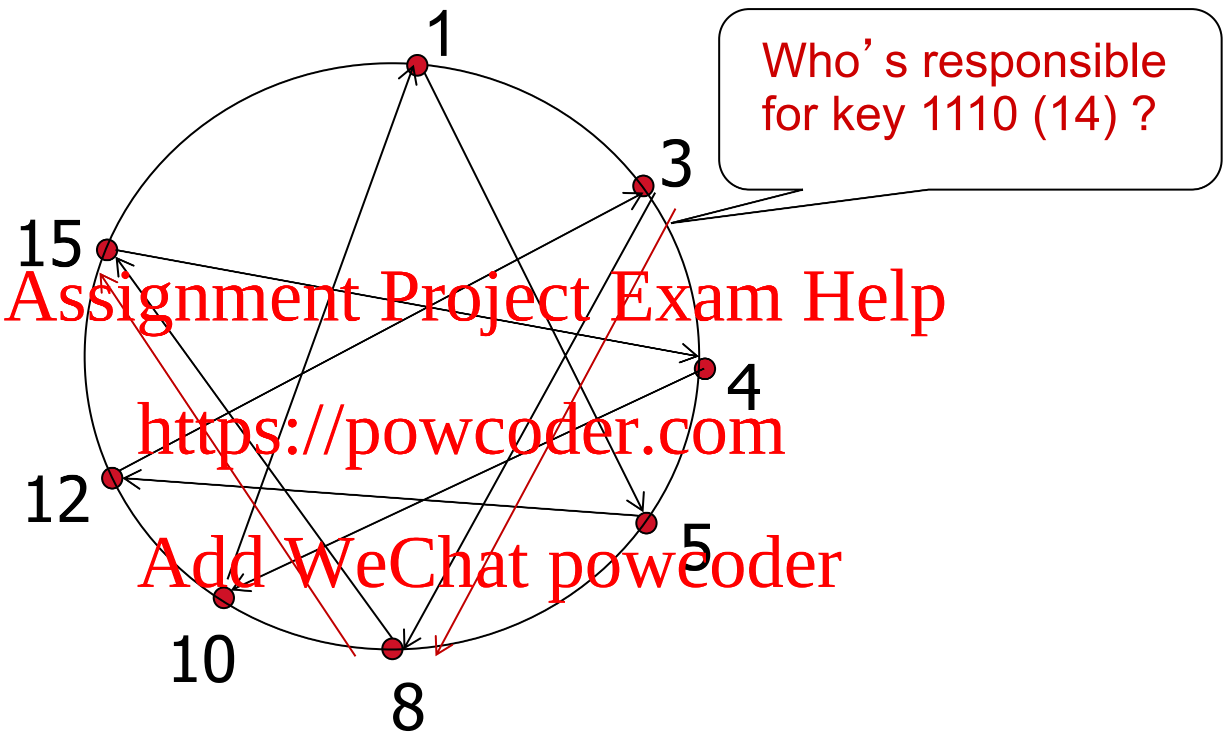
› I have a successor peer

› I have a predecessor peer

› I have some shortcuts to other nodes to speedup delivery of requests

› Chord: A scalable peer-to-peer lookup service for internet applications. Stoica et al. *SIGCOMM* 2001.

Circular DHT (con't)



- each peer keeps track of predecessor, successor, short cuts.
- reduced from 6 to 2 messages.
- possible to design shortcuts so *O(log N)* neighbors, *O(log N)* messages in query

THE UNIVERSITY OF SYDNEY

Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

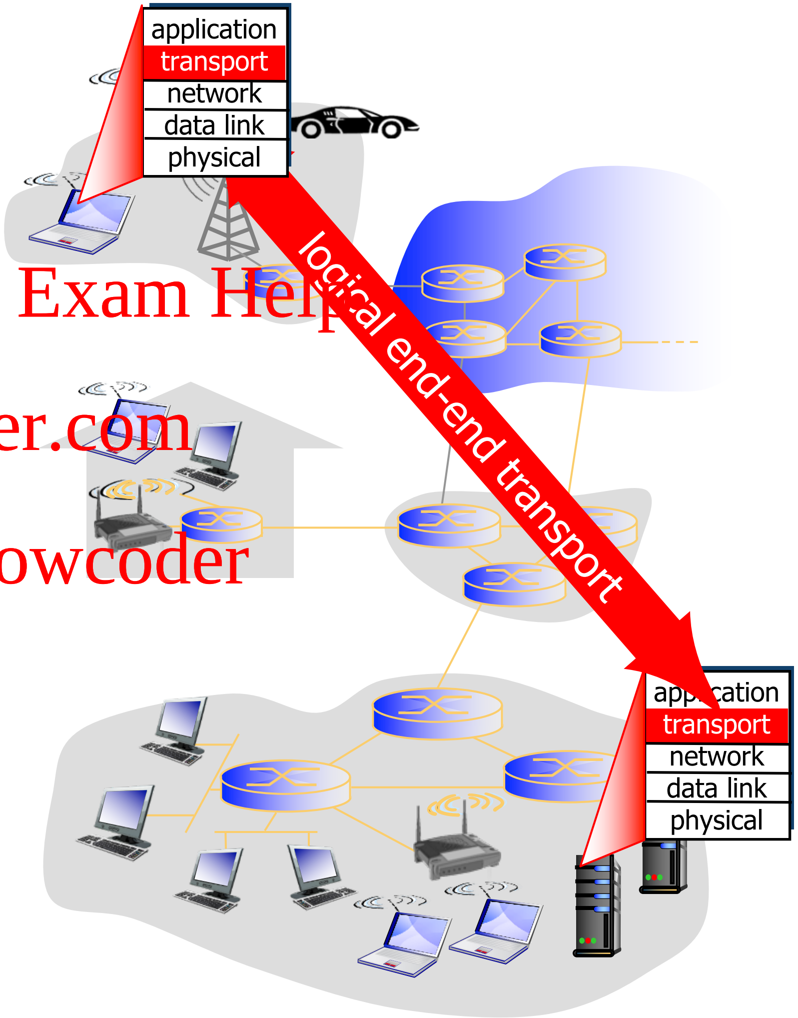
Outline

- Transport-layer services
- Multiplexing/demultiplexing
- Connectionless transport (UDP)
- Principles of reliable data transfer
- TCP protocol

Transport Services

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



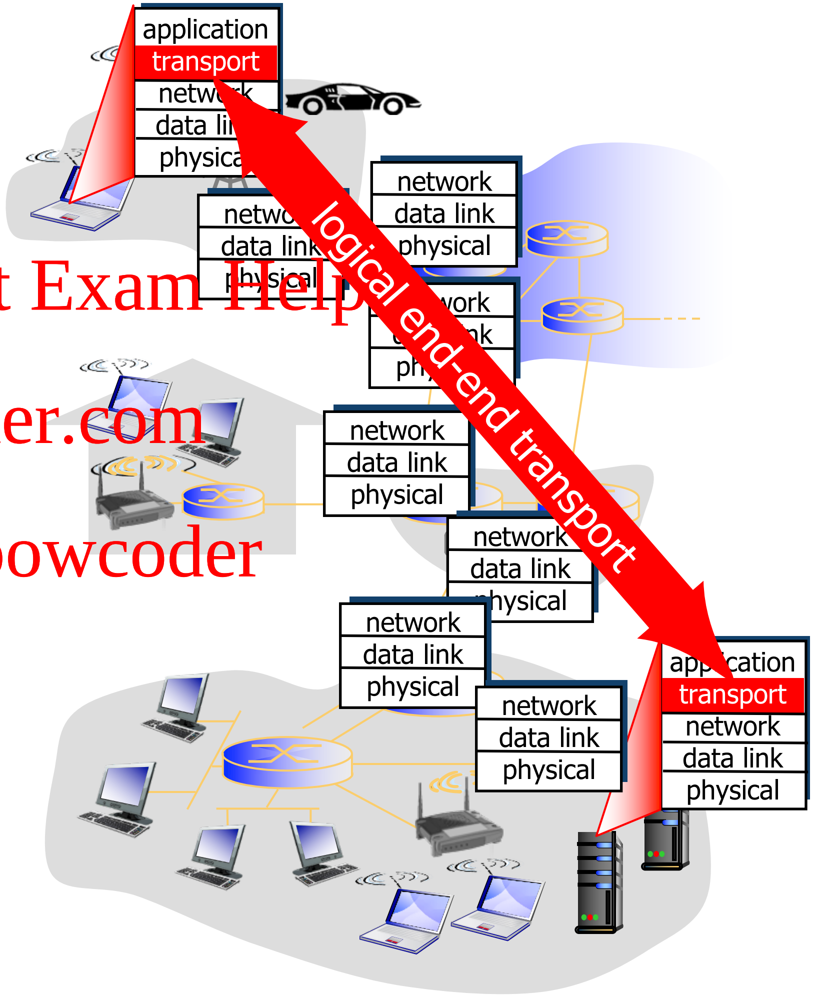
Transport vs. network layer

- *network layer:* host-to-host communication
  - best-effort, unreliable

- *transport layer:* process-to-process communication
  - relies on, enhances, network layer services

Internet transport-layer protocols

- IP: best effort service
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

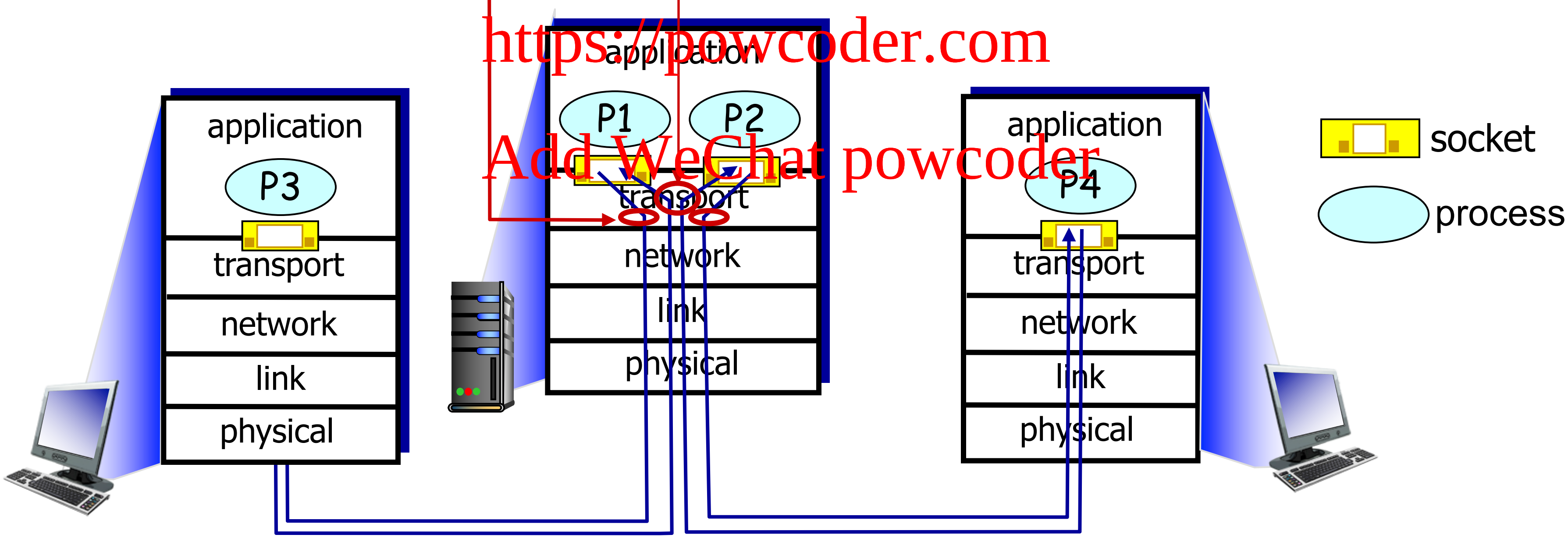


Transport Services

Multiplexing/demultiplexing

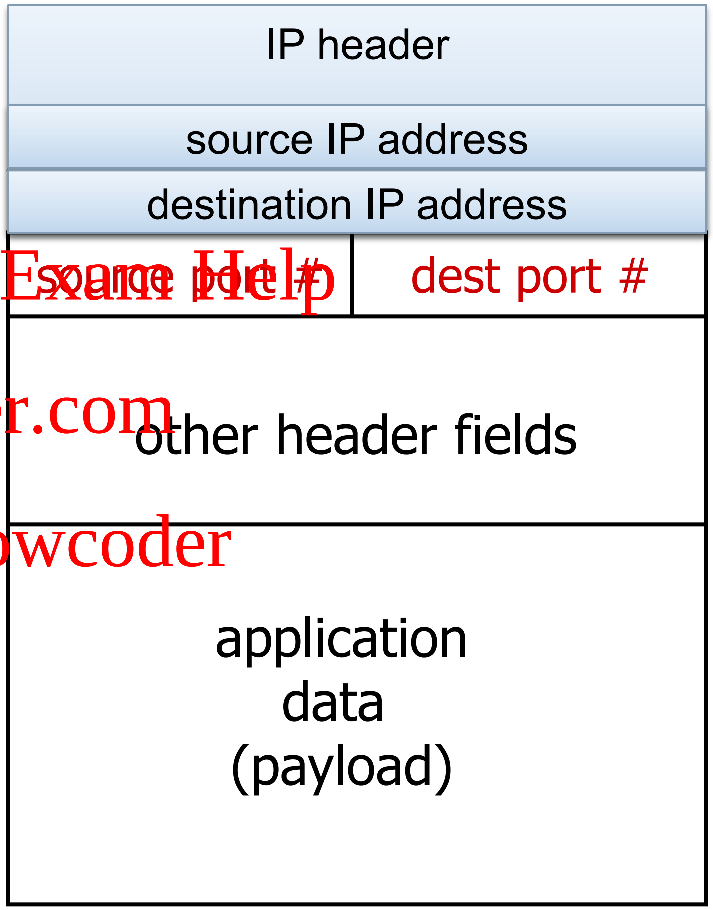
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

THE UNIVERSITY OF SYDNEY

Connectionless demultiplexing

› *Receiver*

› *recall:* created socket has host-local port #:

› when host receives UDP segment:

- Checks destination port # in segment
- directs UDP segment to socket with that port #

› *Sender*

❖ *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

clientSocket.sendto(message,(desip, des port))

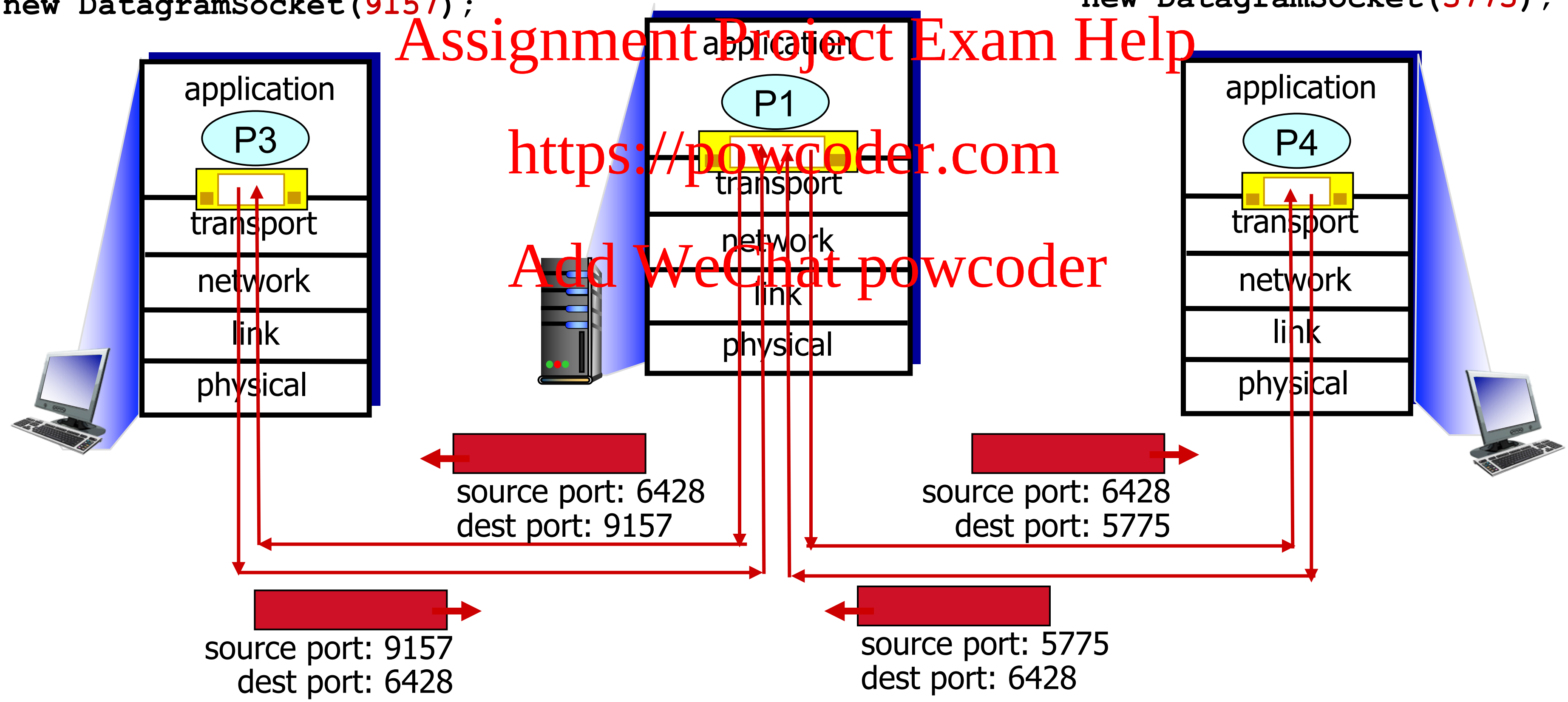
Packets with *same dest.IP address, dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
DatagramSocket serverSocket =
  new DatagramSocket(6428);
```

```
DatagramSocket mySocket2 =
new DatagramSocket(9157);
```

```
DatagramSocket mySocket1 =
  new DatagramSocket(5775);
```

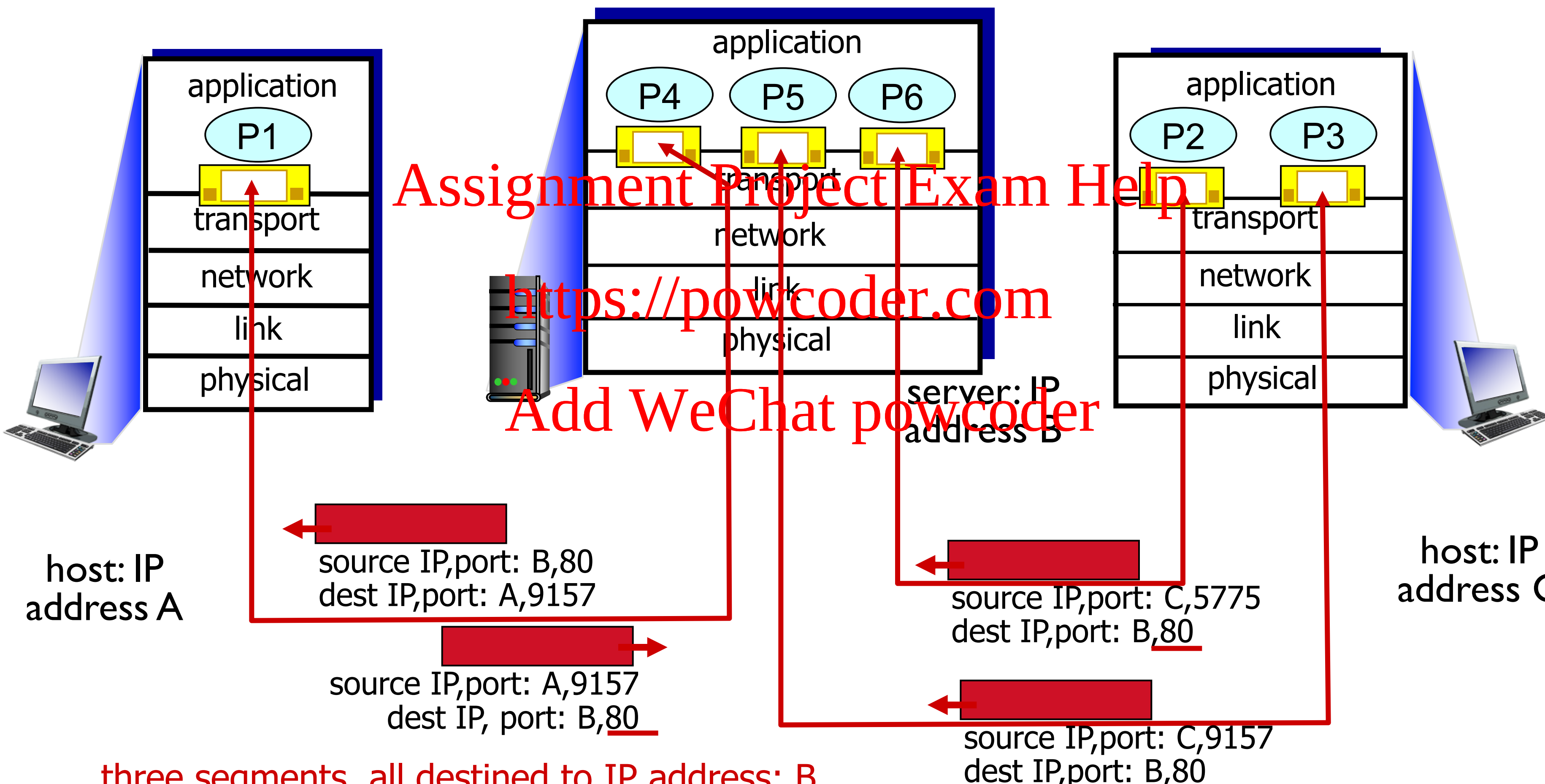



Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

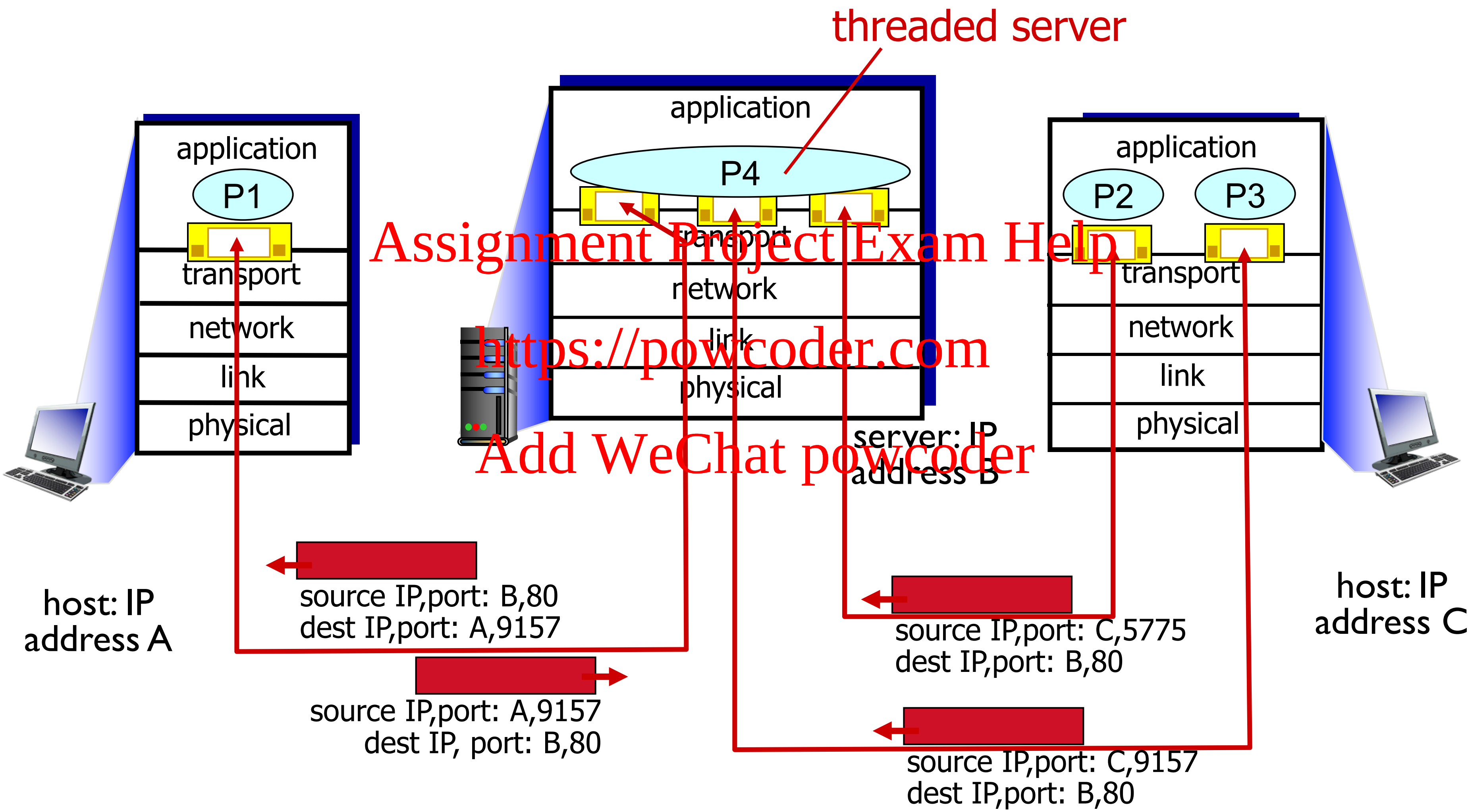
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



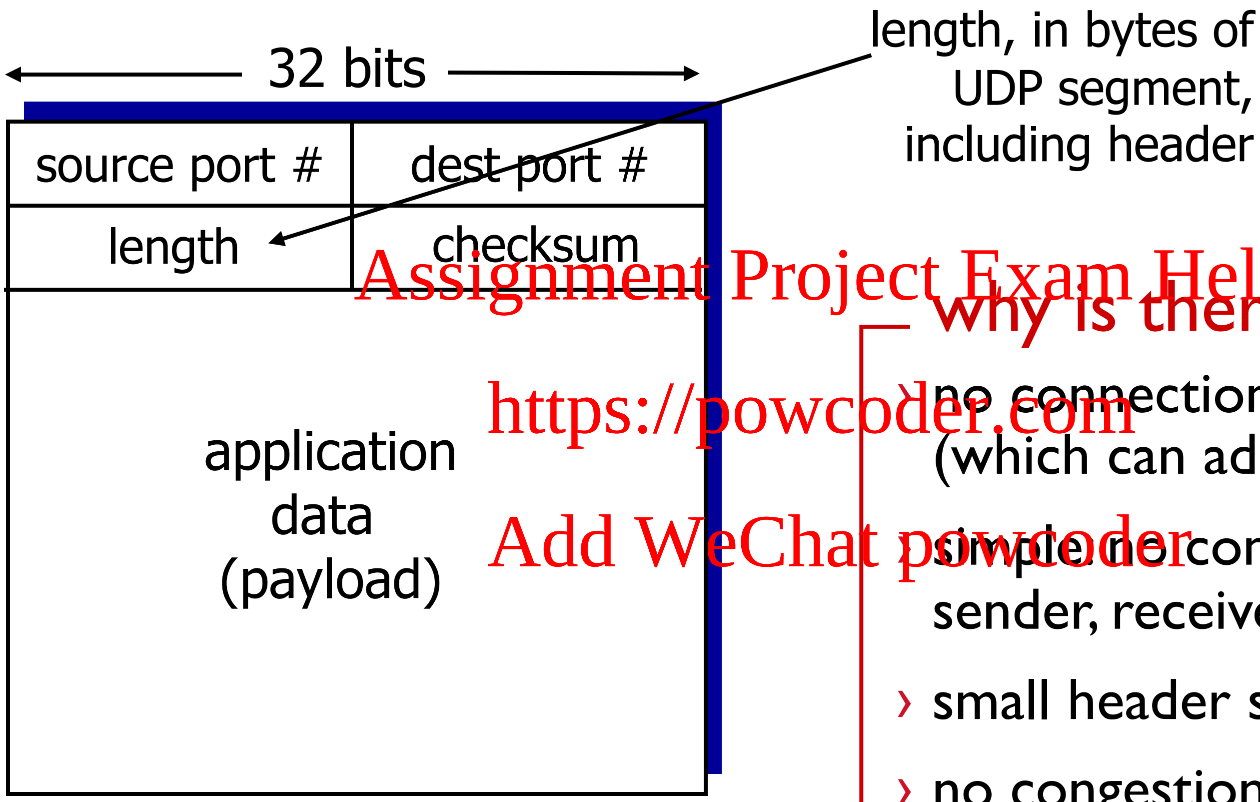
Connectionless Transport UDP

UDP: User Datagram Protocol [RFC 768]

- "no frills," Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- sum: addition (one's complement sum) of segment contents
- checksum: complement of sum
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

Internet checksum: example

example: add two 16-bit integers

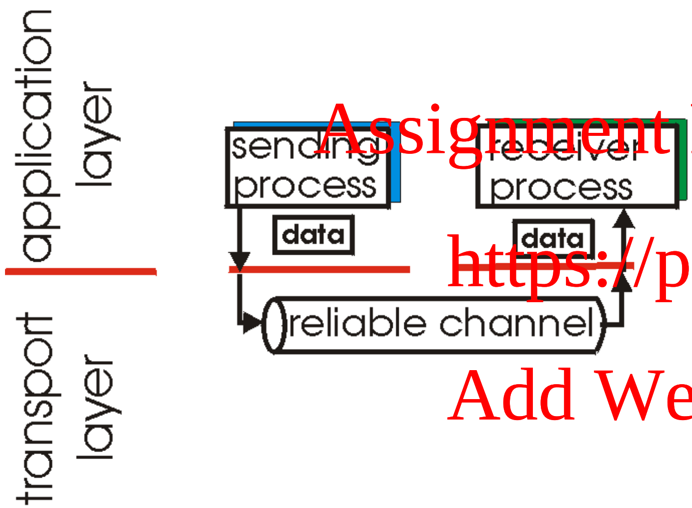
	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
carryout wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Principles of Reliable Data Transfer

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

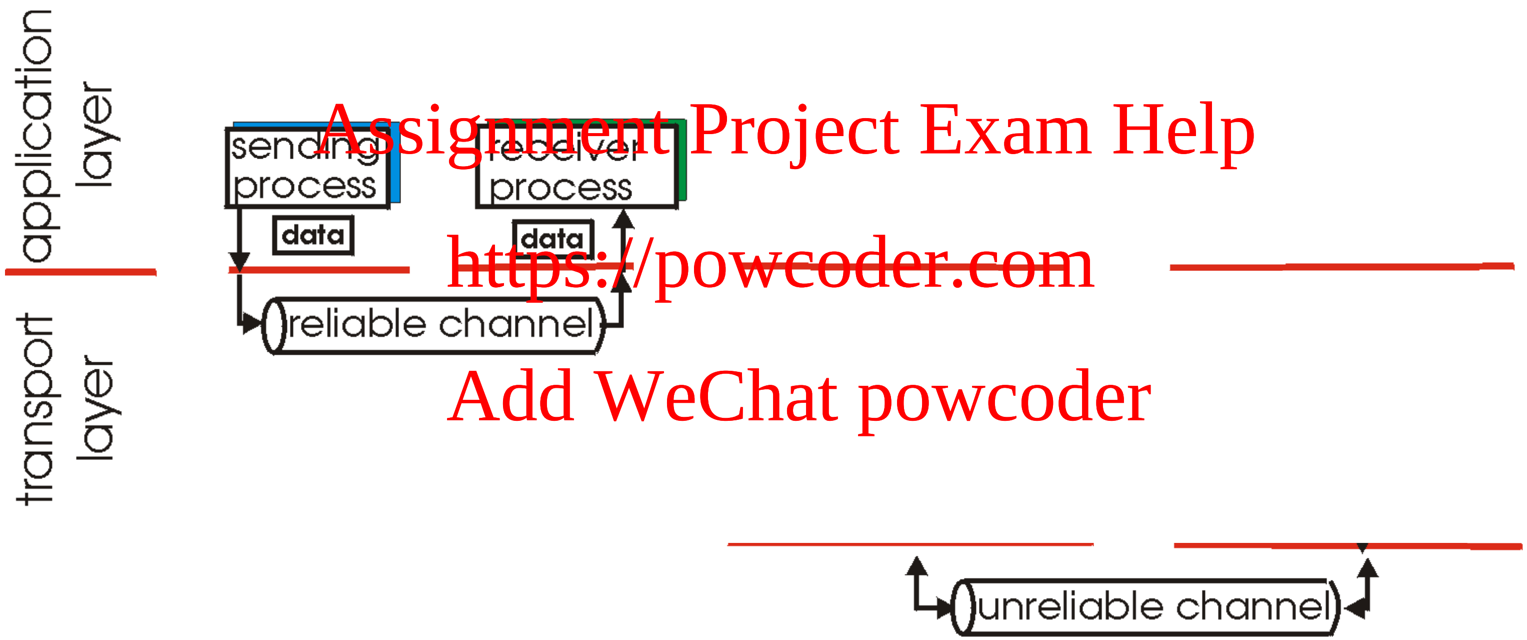


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

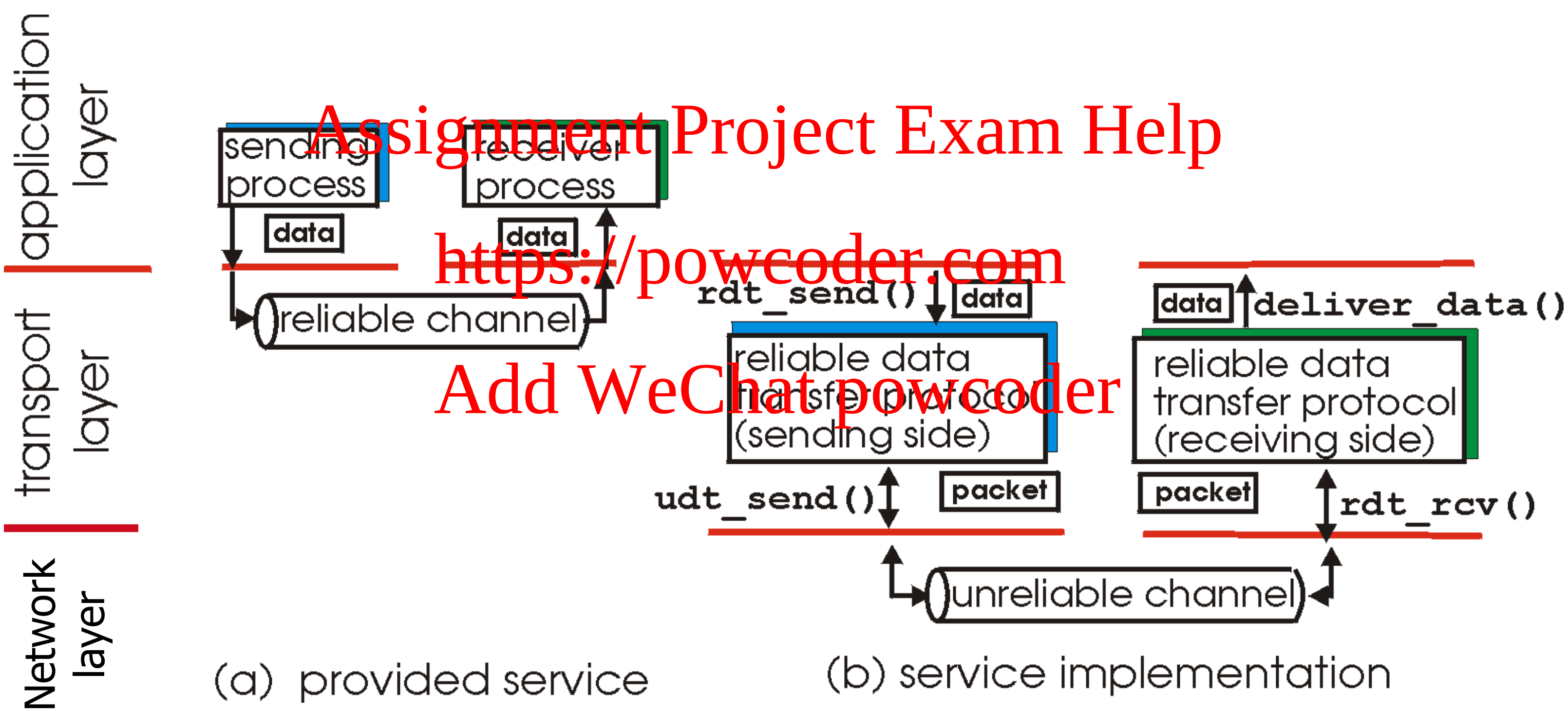


(a) provided service (b) service implementation

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

› important in application, transport, link layers
  - top-10 list of important networking topics!

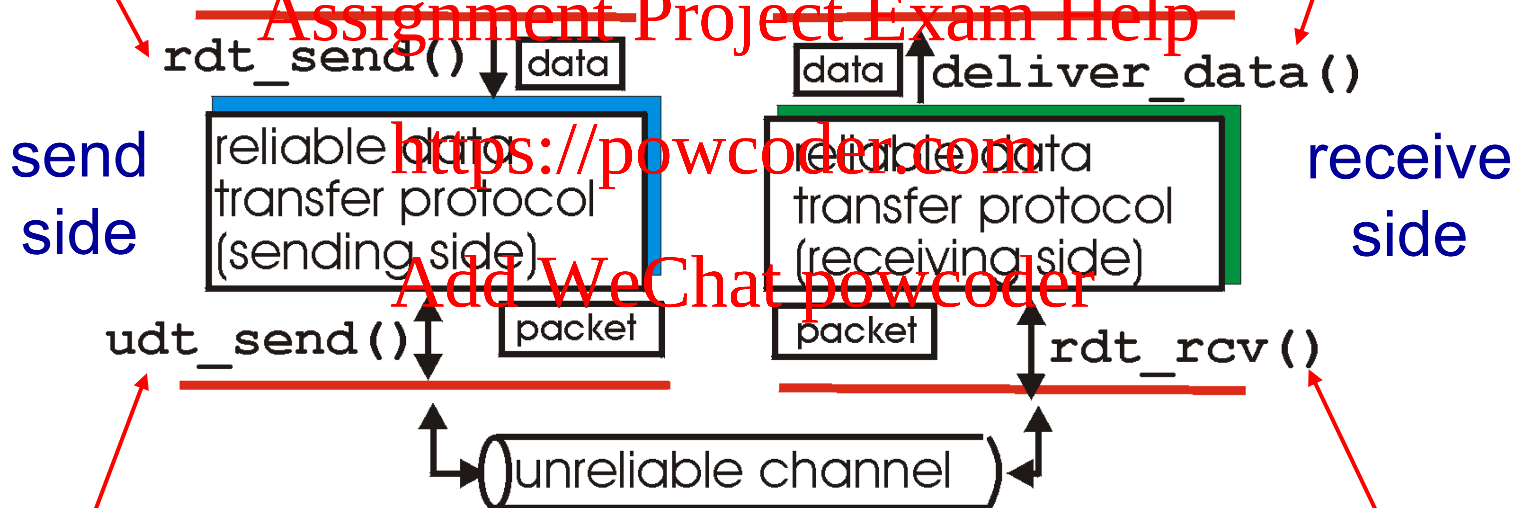


› characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Principles of reliable data transfer

We will:

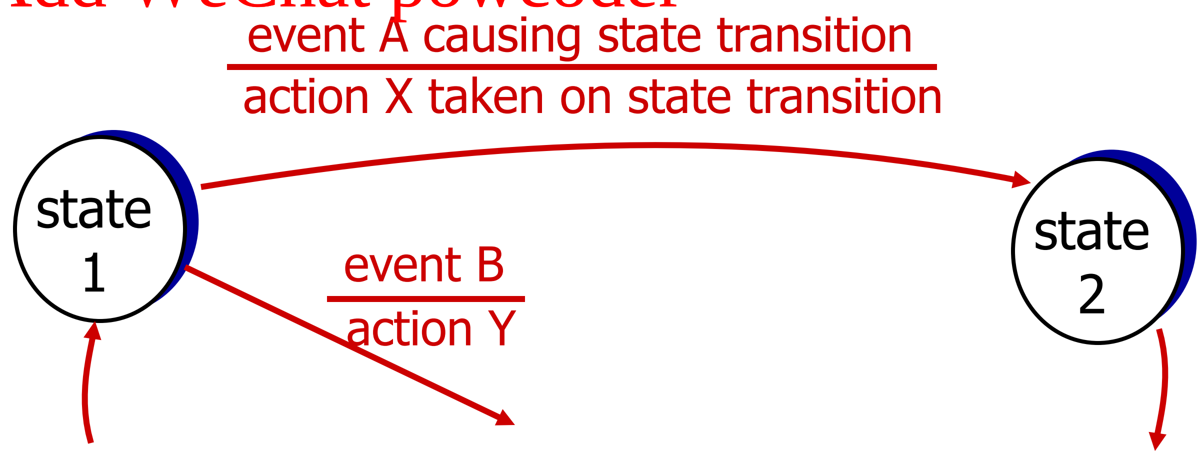
› incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)

› consider only unidirectional data transfer

- but control info will flow on both directions!

› use finite state machines (FSM) to specify sender, receiver

state: when in this "state", next state and action uniquely determined by next event

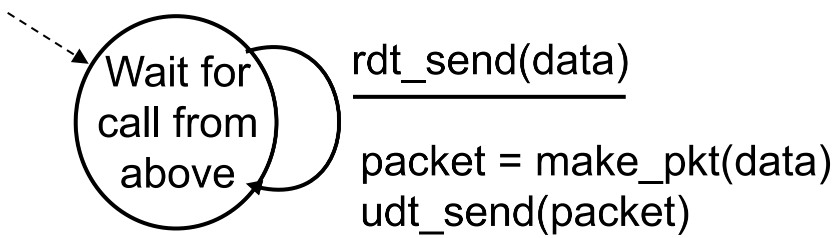


Principles of reliable data transfer

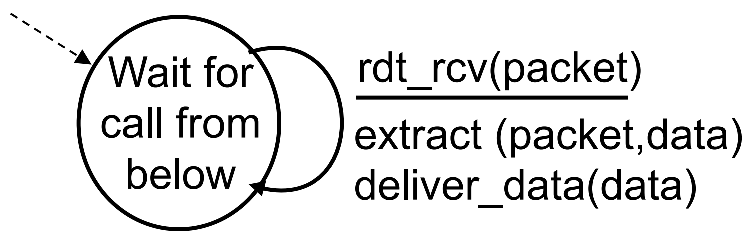
› underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

› separate FSMs for sender, receiver:

  - sender sends data into underlying channel
  - receiver reads data from underlying channel



sender



receiver

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

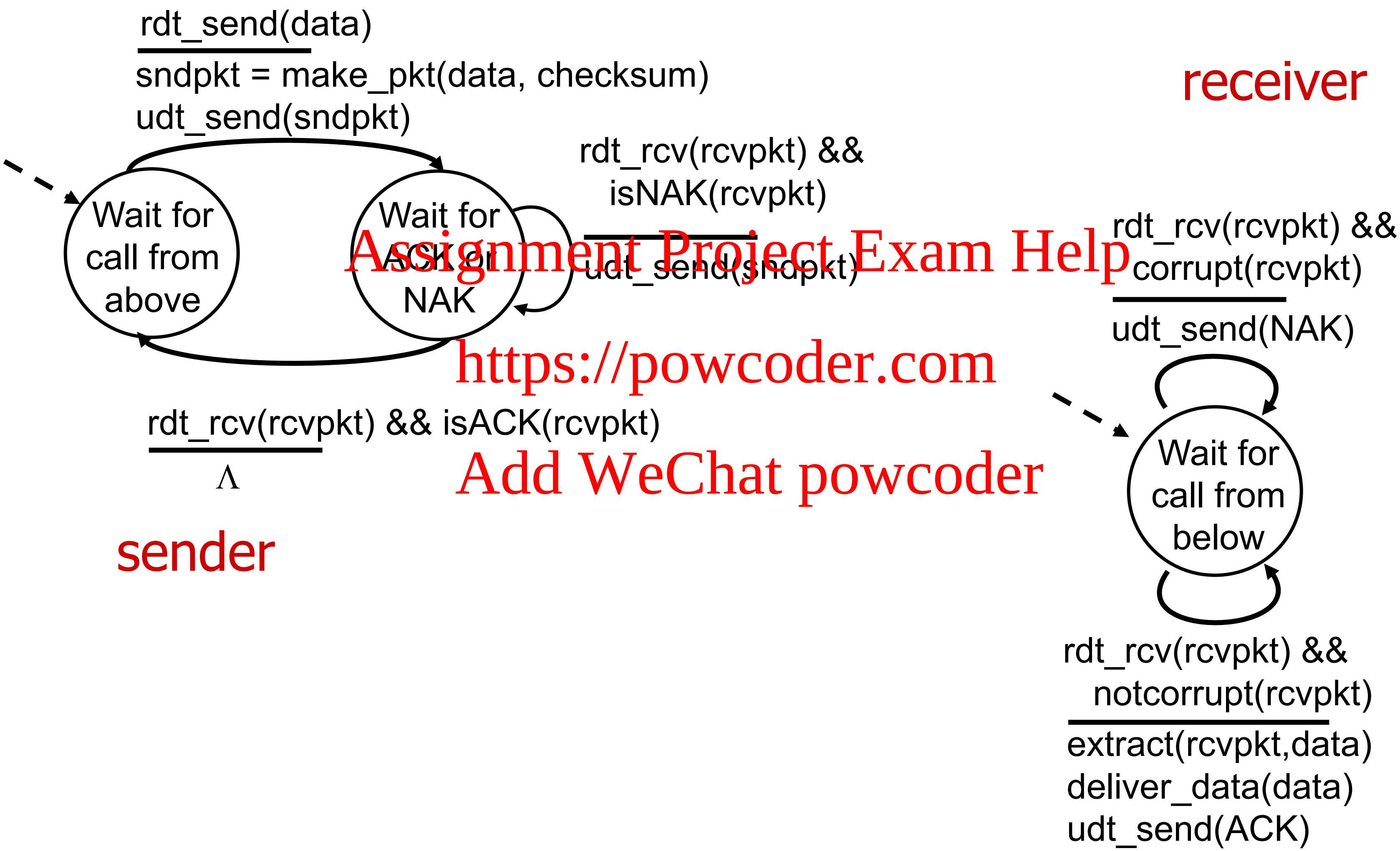
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK

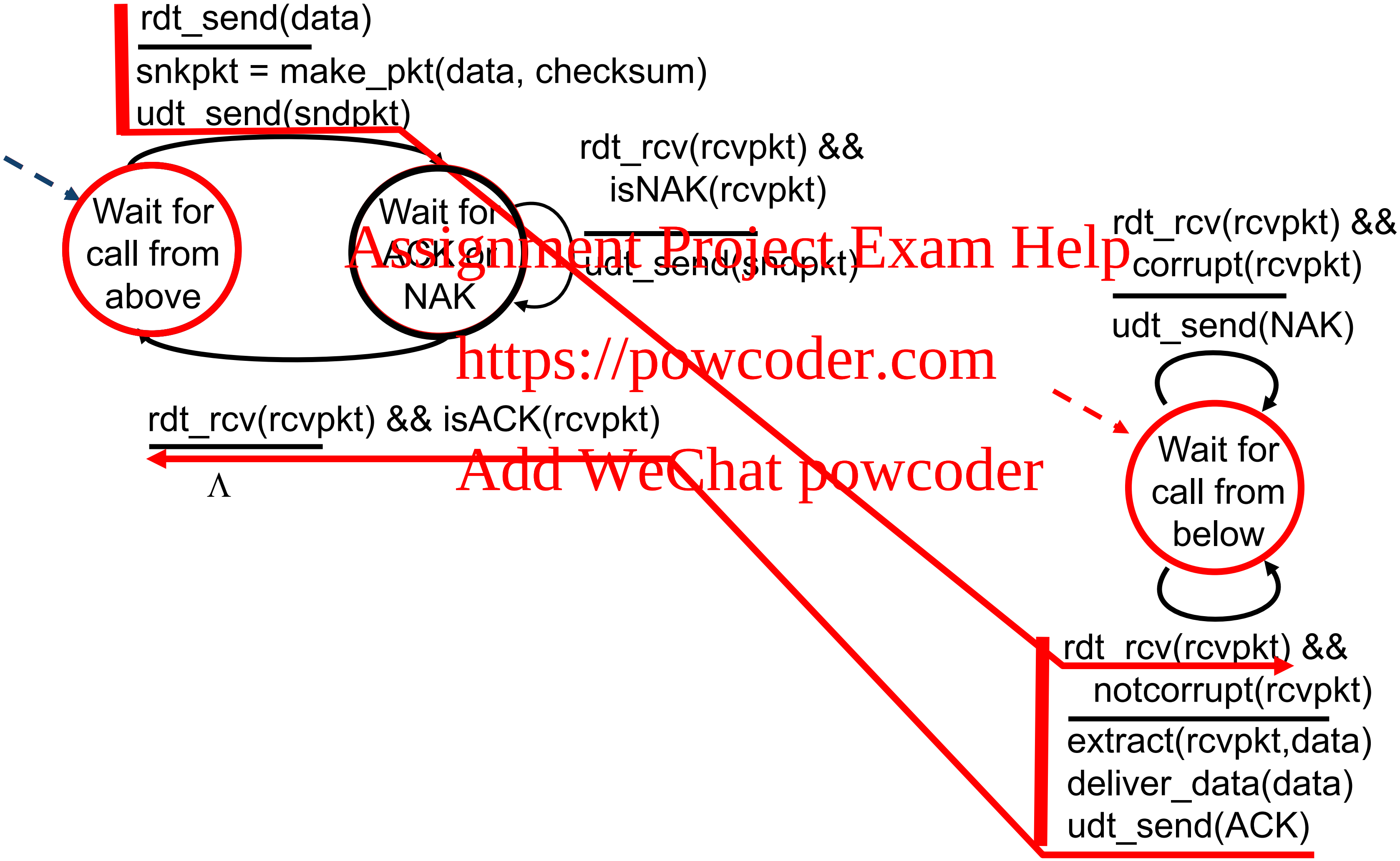
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors

  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

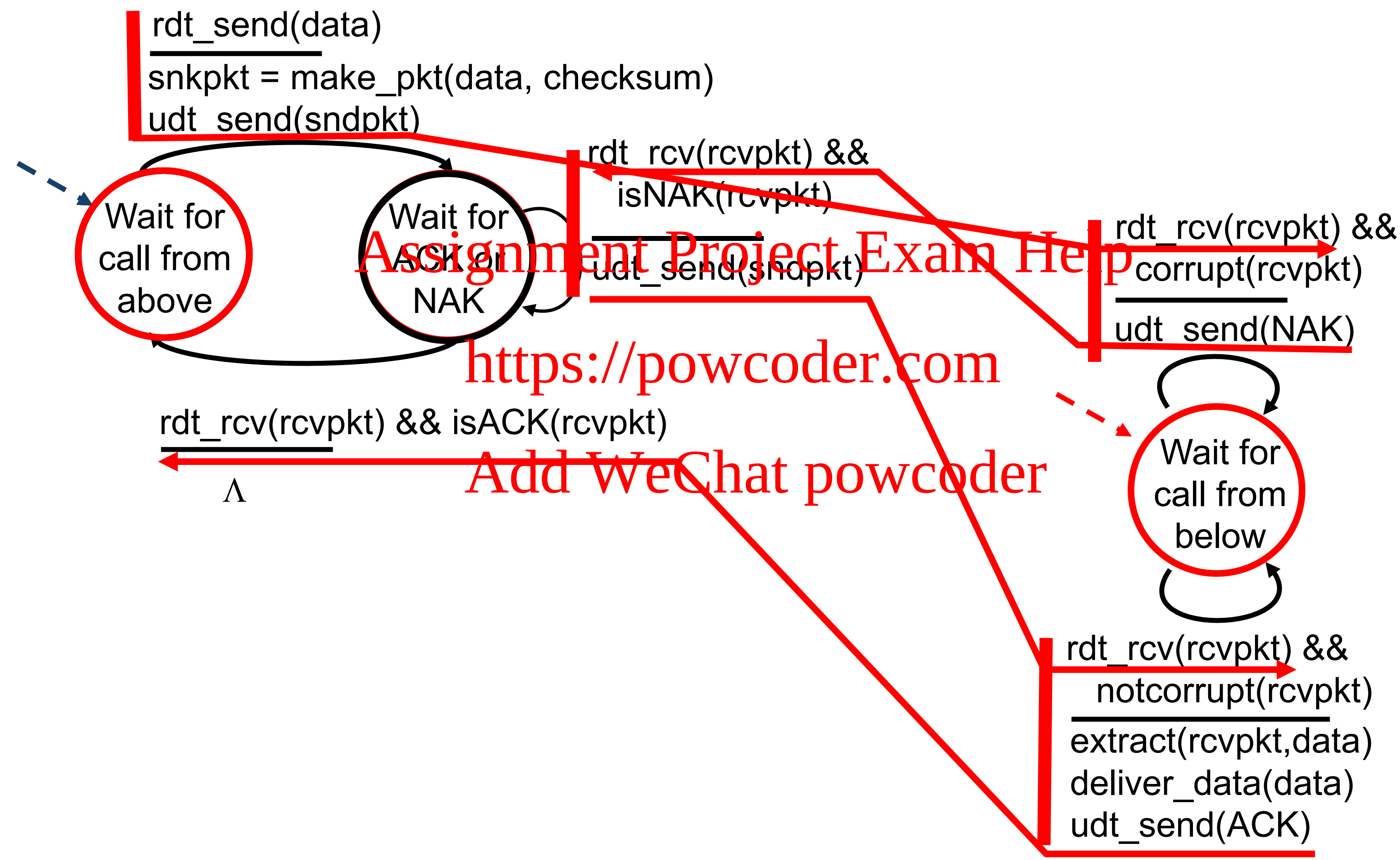
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender does not know what happened at receiver!
- cannot just retransmit: possible duplicate

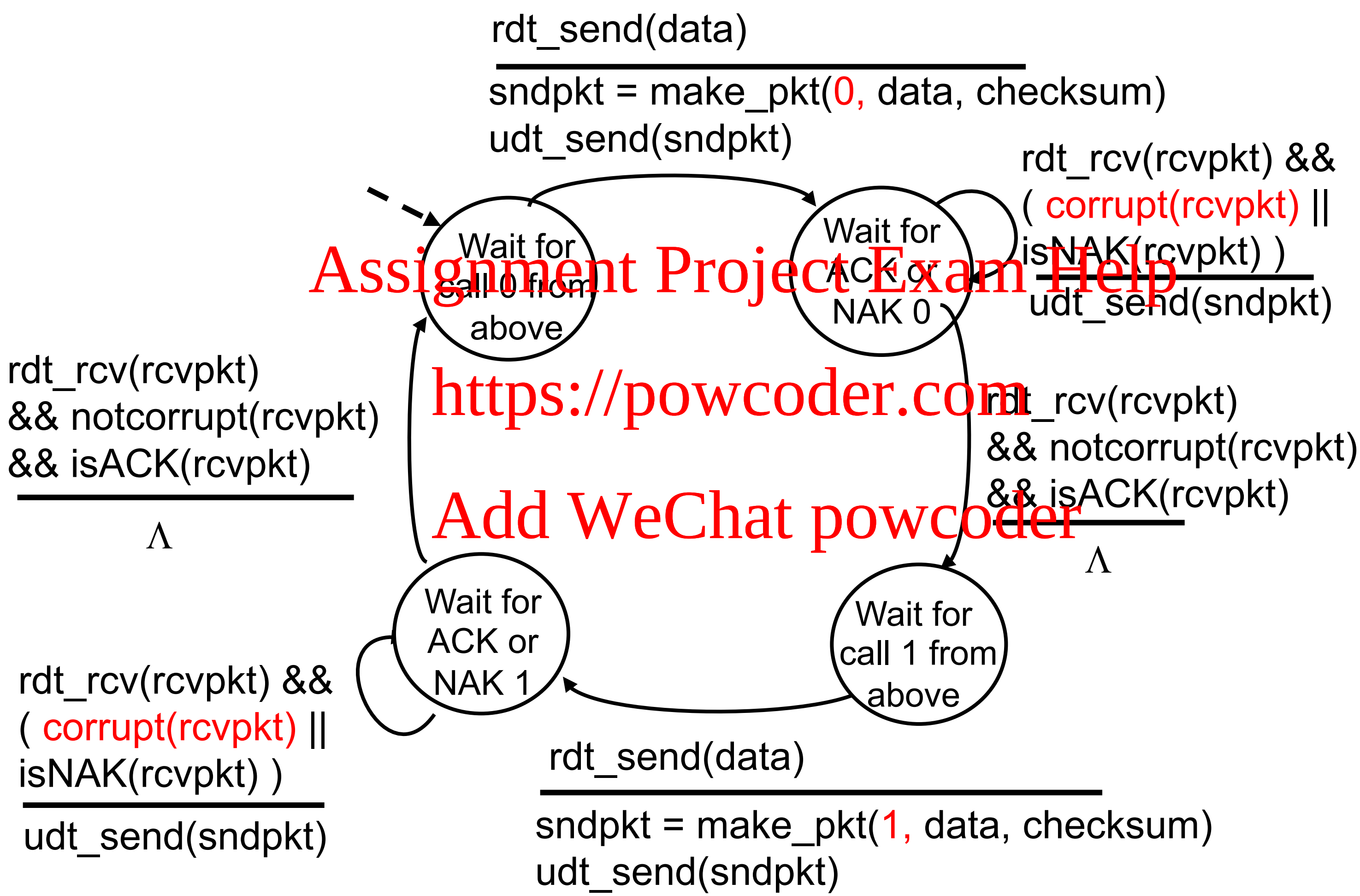
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (does not deliver up) duplicate pkt

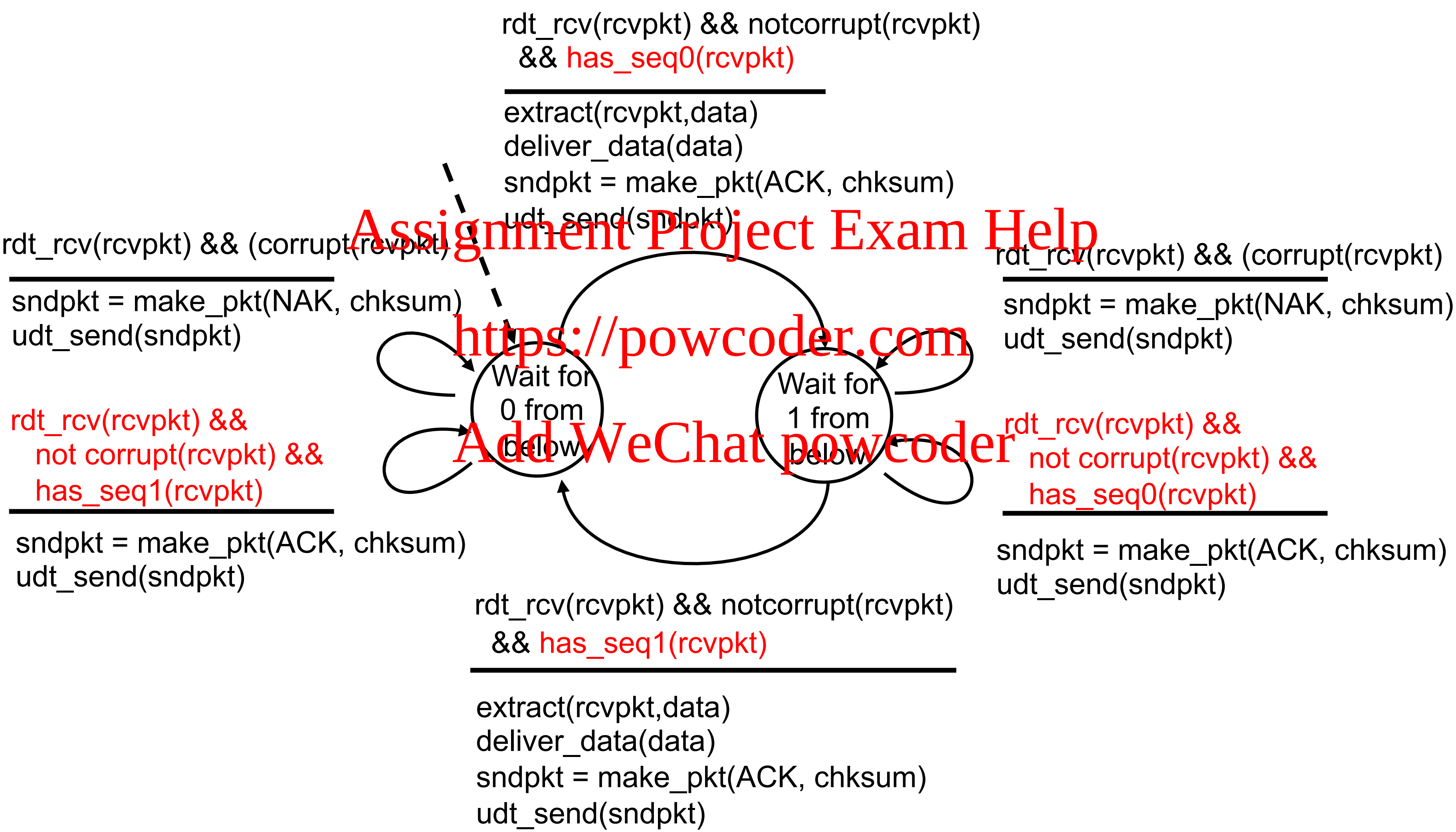
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #' s (0,1) will suffice.
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

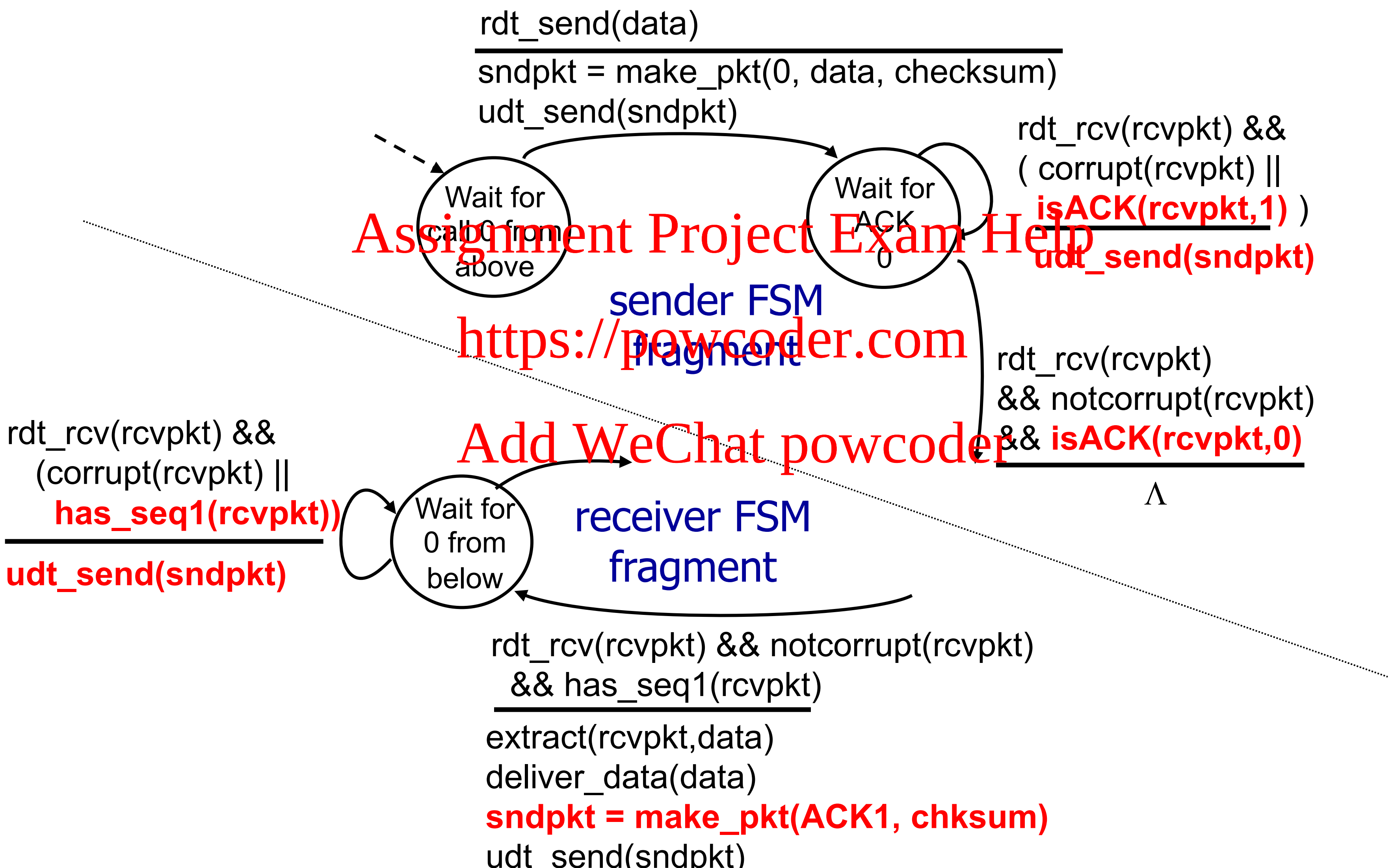
receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- "unexpected" ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender, receiver fragments



THE UNIVERSITY OF SYDNEY

rdt3.0: channels with errors and loss

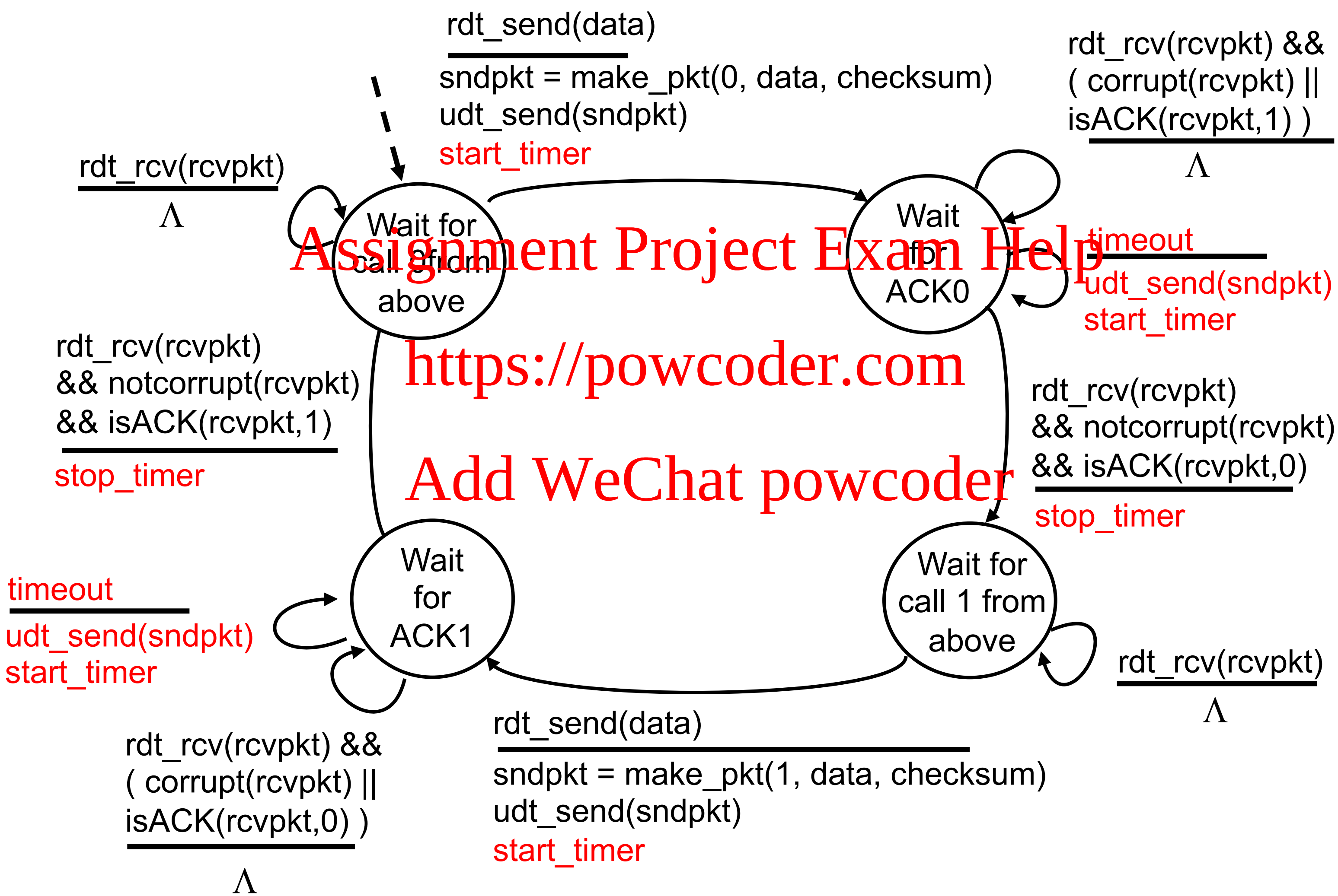
new assumption: underlying channel can also lose packets (data, ACKs)

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

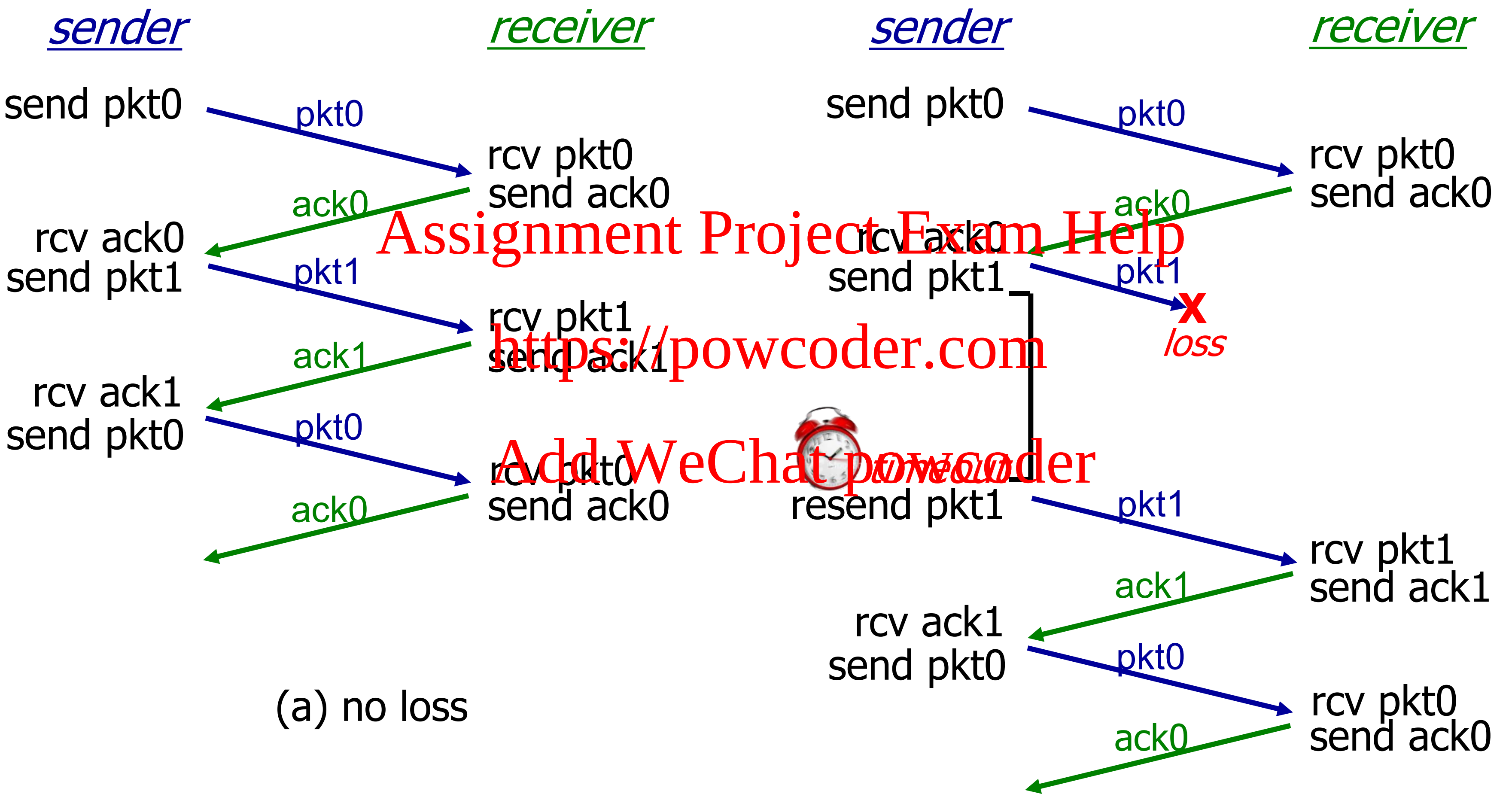
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 sender



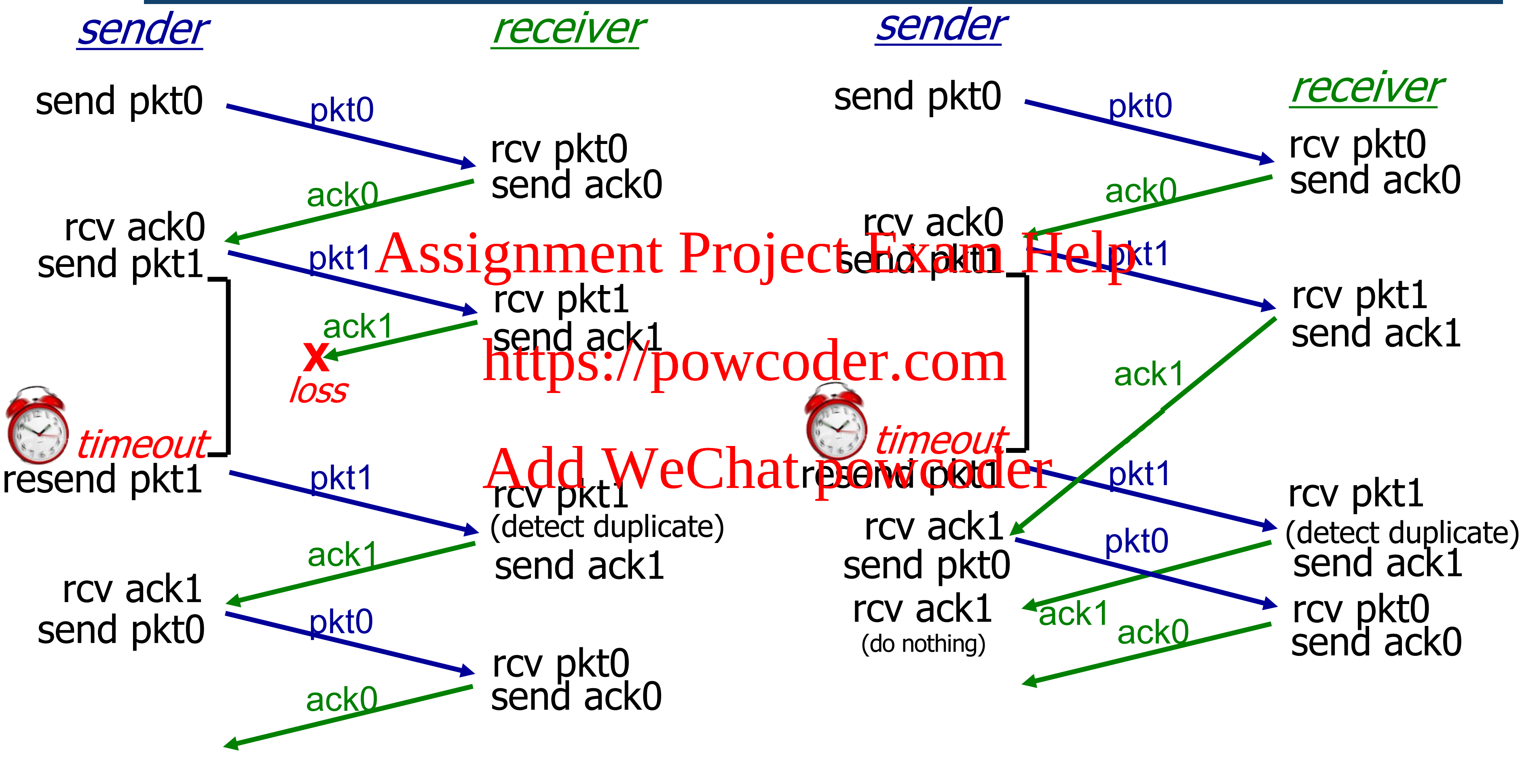
rdt3.0 in action



(a) no loss

(b) packet loss

rdt3.0 in action



(c) ACK loss

(d) premature timeout/ delayed ACK

Performance of rdt3.0

› rdt3.0 is correct, but performance stinks

› e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

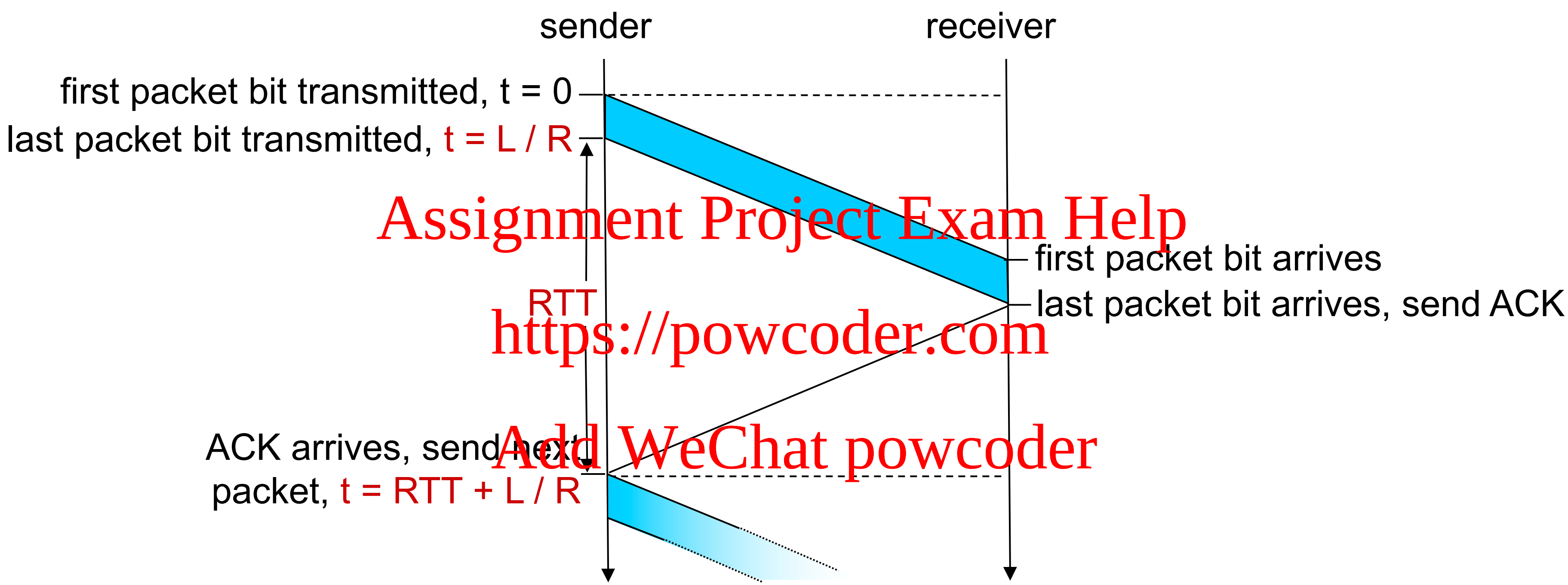
- U_{sender}: *utilization* – fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link

❖ network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

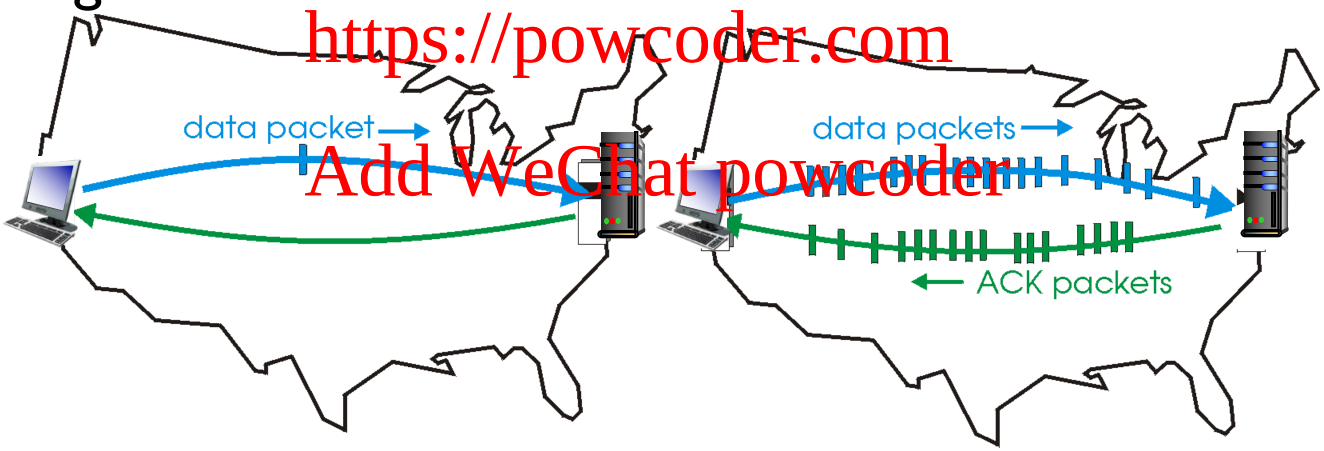


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

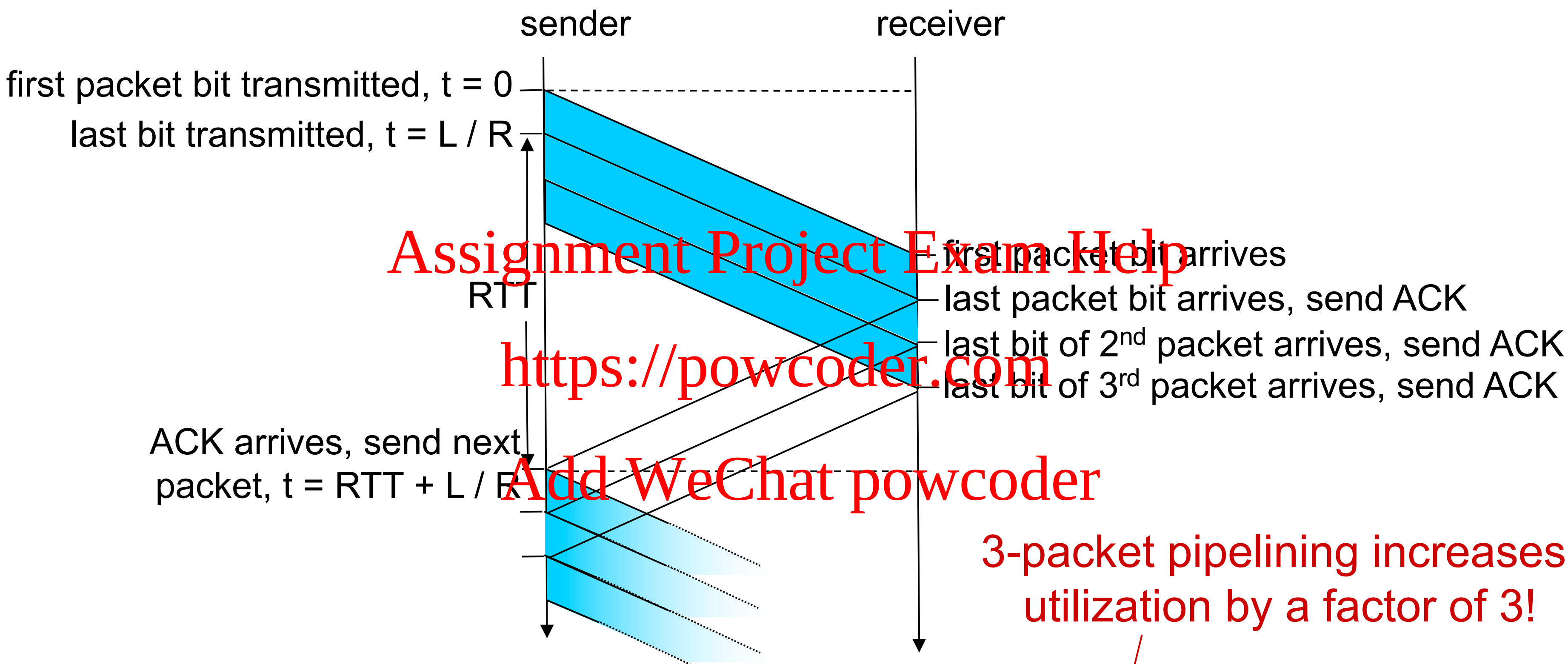


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

› two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined protocols: overview

Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - does not ack packet if there is a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unacked packets in pipeline
- receiver sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

› "window" of up to N, consecutive unacked pkts allowed

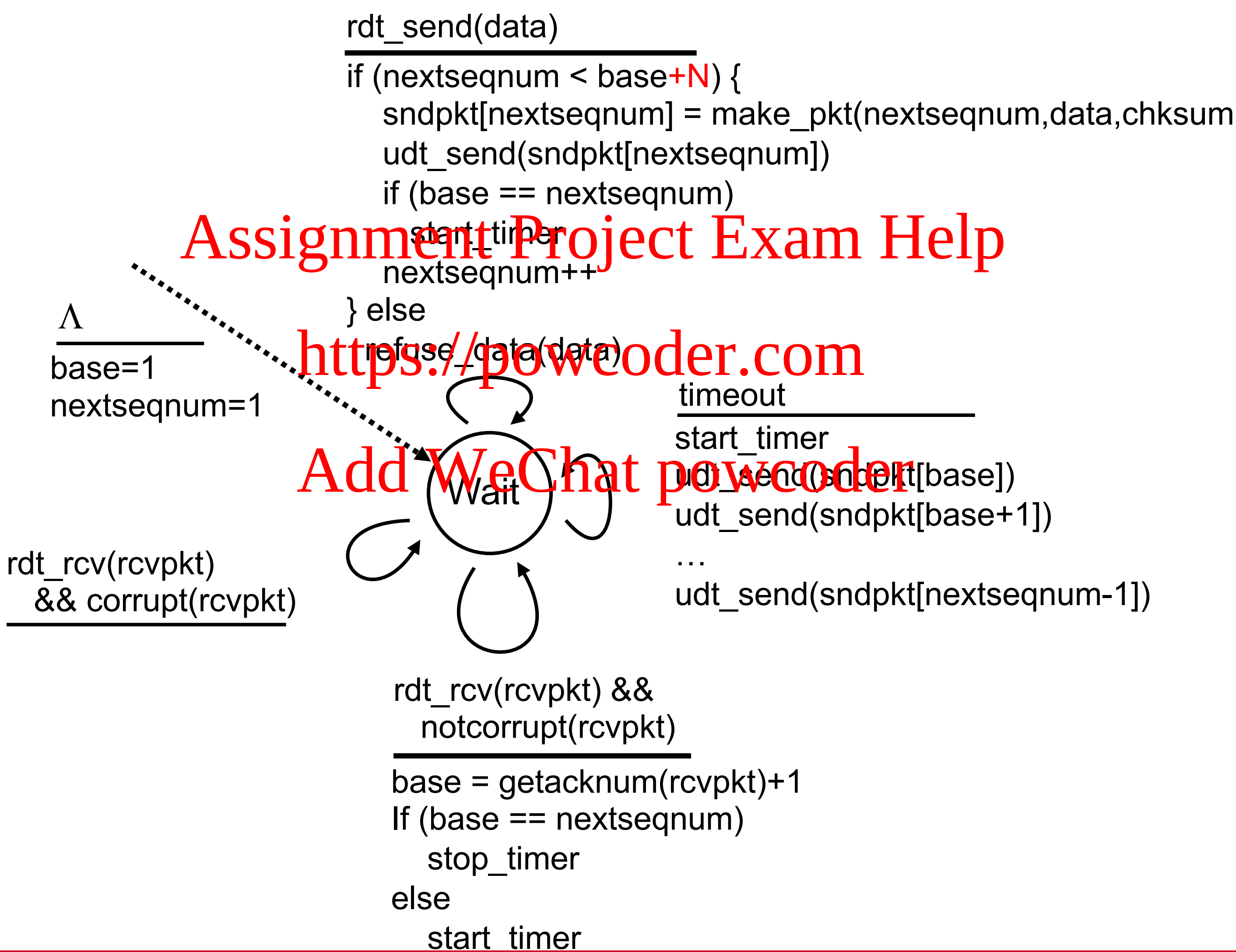
- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

Go-Back-N: sender

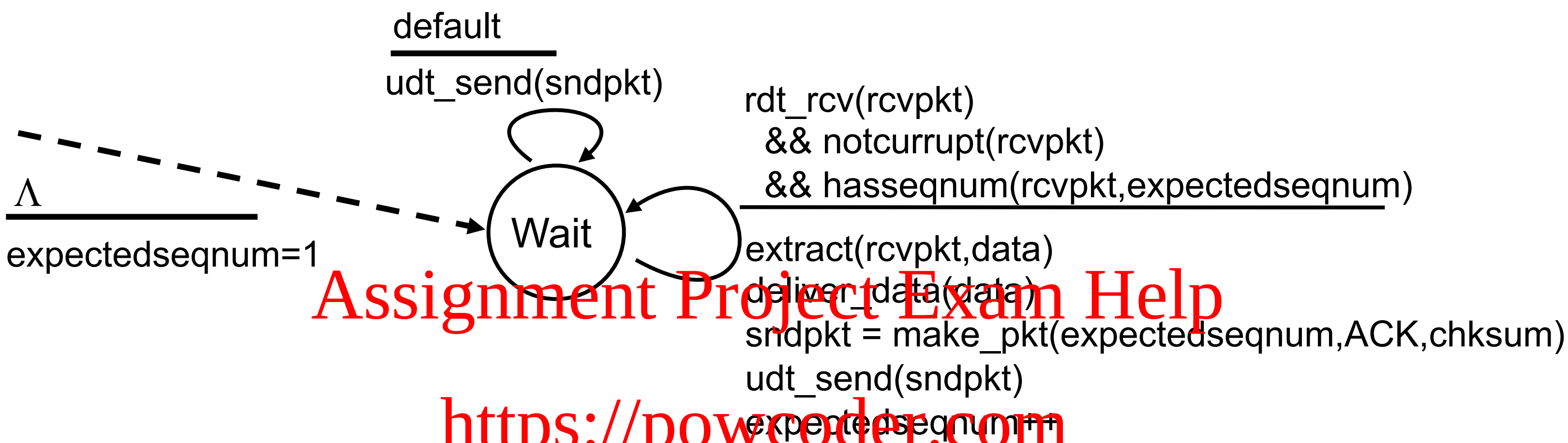
› "window" of up to N, consecutive unacked pkts allowed

- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM



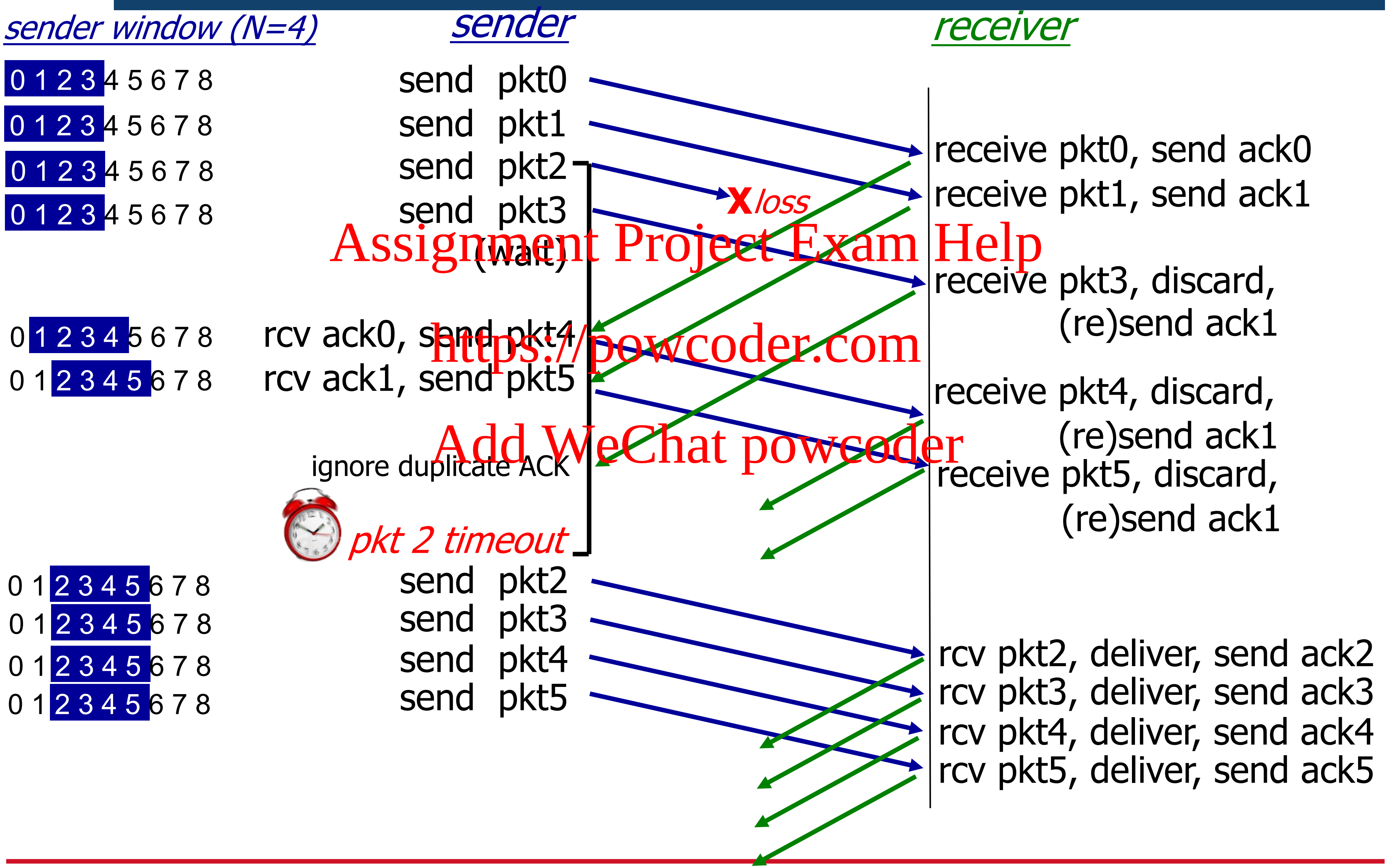
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**

› out-of-order pkt:

- discard (don't buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #

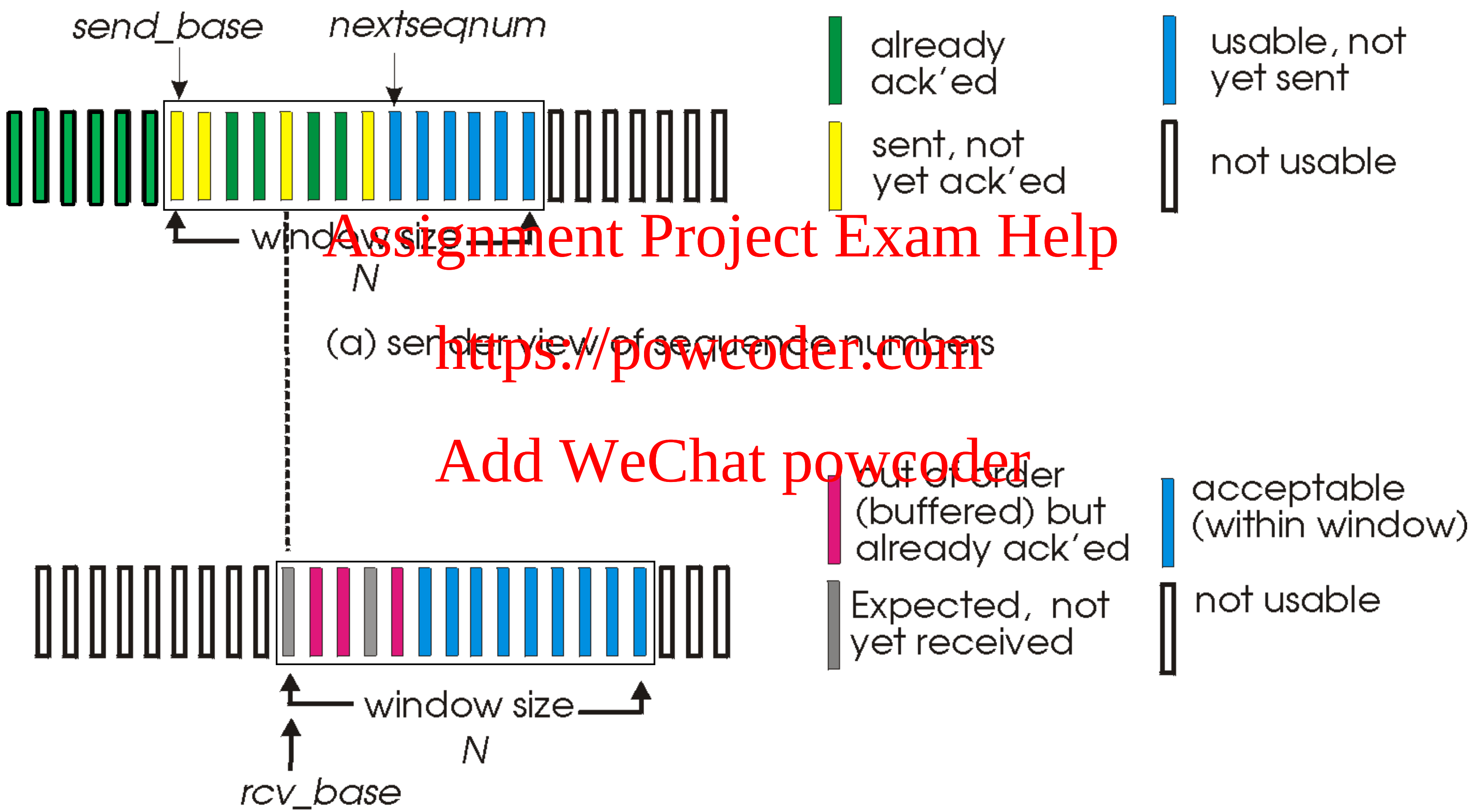
GBN in action



Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
- receiver window

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N-1]:

- mark pkt n as received
- if n is smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

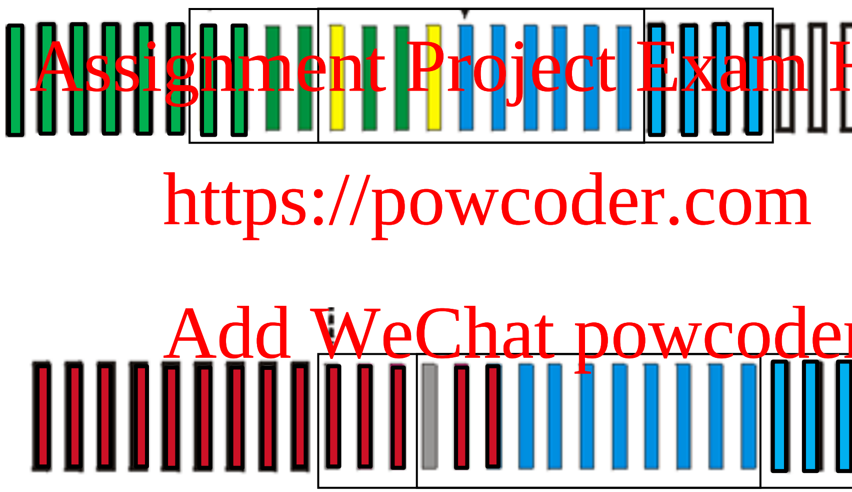
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore



Selective repeat in action