Practical Discrete Event Simulation and The Python Simulator

Michele Segata, Renato Lo Cigno

ANS Group – DISI – University of Trento, Italy

http://disi.unitn.it/locigno/index.php/teaching-duties/spe

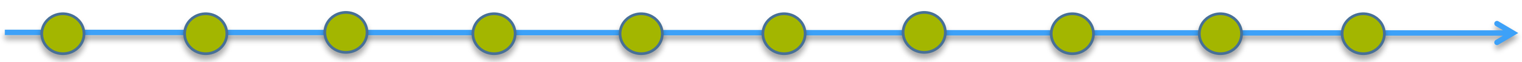
A little bit of background

DISCRETE EVENT SIMULATION

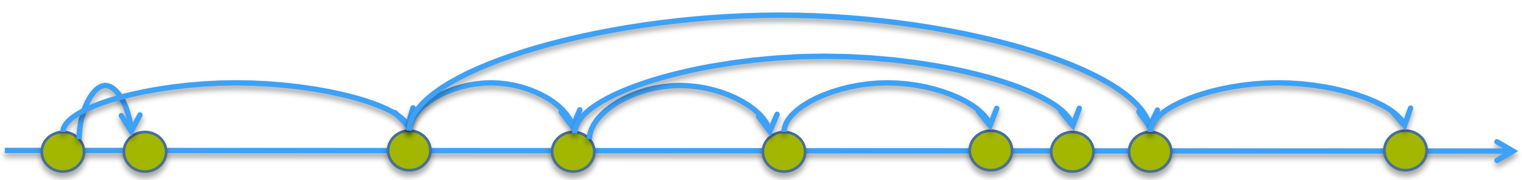
What is a discrete simulation?

- Simulation: reproducing the behavior of a real-world system
  - mathematical
    - a(t) = a0
    - v(t) = a0 * t + v0
    - x(t) = a0/2 * t^2 + v0 * t + x0
  - numerical
    - a[k] = a0
    - v[k] = v[k-1] + a0 * Ts, with v[0] = v0
    - x[k] = x[k-1] + (v[k] + v[k-1])/2 * Ts, with x[0] = x0

- Discrete simulation: simulation "exists" only in specific time moments
  - time driven: sampled with a certain frequency (e.g., 10 Hz)



  - event driven: evolution by the generation and the consumption of events



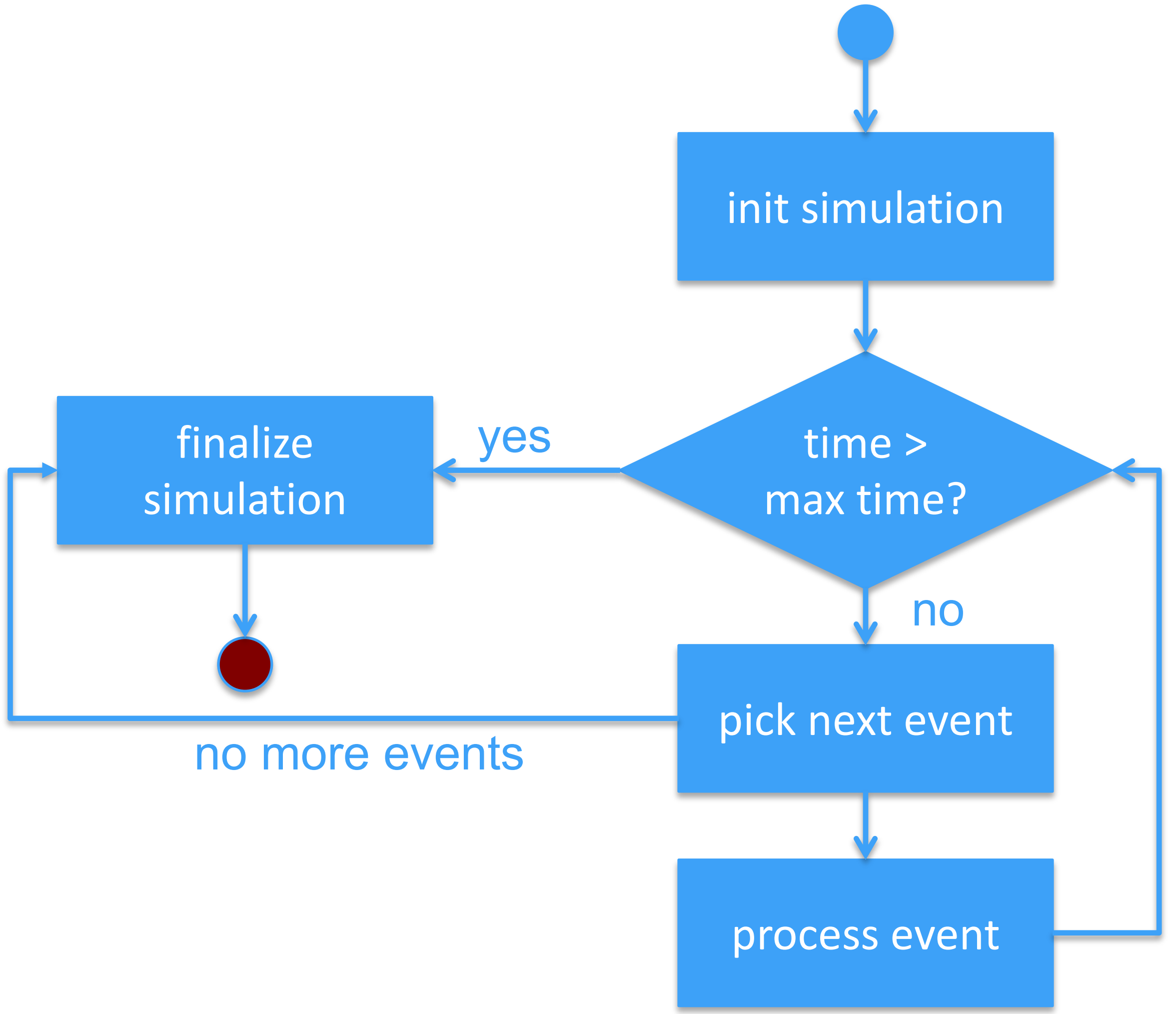
DES: Notions

- State: represents the state of the system
  - In a G/G/1 queue: a single integer (number of clients in the queue)
  - In a chain of N queues: N integers (number of clients in each queue)
  - In a wireless network with N nodes: a complex set of variables
    - x, y, and z position of each node
    - Radio status (e.g., IDLE, TXing, RXing)
    - Protocol-dependent variables (e.g., backoff counter in a WiFi card)
- Events: change (or might not) the state of the system
  - In a G/G/1: queue: the arrival or the departure of a client
  - In a wireless network: the generation of a packet at the application, the beginning and the end of a transmission, the beginning and the end of a reception, …
  - Events evolve the simulation by changing the state and/or generating new events
- Time: updated according to events

DES simulator structure

- Components and variables:
  - A queue of events
  - Current time
  - Variables for performance monitoring (e.g., # of events)
  - Modules implementing the behavior of system components (models)
- Working principle:
  - Initialize simulation modules
  - Pick the first (in terms of time) event from the queue
  - Update current time and check for terminating condition
  - Invoke the event handling of the destination module
  - Repeat

DES – A generic view

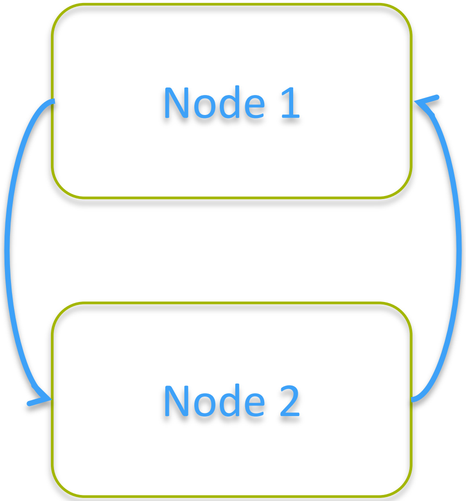


- Very easy example: two nodes communication

```
on init:
  scheduleEvent(sendMsg, now + exp(1))
  messageCount = 0

on event(event):
  if (event == sendMsg) {
    send(packet)
    scheduleEvent(sendMsg, now + exp(1))
  } else {
    if (random() > 0.5)
      messageCount++
  }

on finish:
  saveToFile(messageCount)
```



DES philosophy

- Be careful: philosophy change needed
  - EVERYTHING is an event
    - schedule events
    - handle events
  - events are atomic
    - no duration

WRONG!

```
onStartRx:
  beginRx = now
  wait(endOfTransmission)
  rxDuration = now - beginRx
```

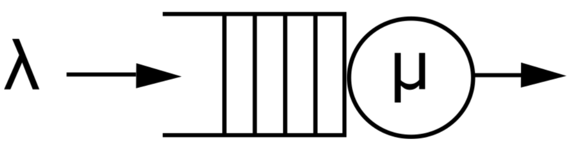
CORRECT!

```
onStartRx:
  beginRx = now

onEndRx:
  rxDuration = now - beginRx
```

An Example: Queue Simulator

- Input parameters:
  - arrival rate distribution A (e.g., exp(1/λ))
  - service rate distribution B (e.g., U(μ - 1, μ + 1), μ > 1)
  - queue length L (0 for infinite)
  - single server
- Output:
  - queue length over time
  - jobs dropped over time
- Possible events:
  - arrival of a job
  - service of a job



An Example: Queue Simulator

```
on init:
    jobId = 0
    arrival.jobId = jobId
    scheduleEvent(arrival, now + A)
    queue = emptyQueue()
```

CAREFUL! This is the queue we are simulating, NOT the event queue of the simulator.
The event queue is managed by the simulator, you don't see it.

- Input parameters:
  - arrival A
  - service B
  - queue length L
- Output:
  - queue length over time
  - jobs dropped over time
- Possible events:
  - arrival of a job
  - service of a job

An Example: Queue Simulator


```
on event(event):
    if (event == arrival) {
        if (queue.length() < L OR L == 0) {
            queue.add(arrival.jobId)
            if (queue.length() == 1)
                scheduleEvent(service, now + B)
        } else {
            logDrop(now, arrival.jobId)
        }
        logQueueLength(now, queue.length())
        jobId++
        arrival.jobId = jobId
        scheduleEvent(arrival, now + A)
    } else if (event == service) {
        queue.removeFirst()
        logQueueLength(now, queue.length())
        if (queue.length() != 0)
            scheduleEvent(service, now + B)
    }
}
```

- Input parameters:
  - arrival A
  - service B
  - queue length L
- Output:
  - queue length over time
  - jobs dropped over time
- Possible events:
  - arrival of a job
  - service of a job

An Example: Queue Simulator

- Simplest possible implementation
  - except for logging of job drops (why?)
  - number of drops can be derived from queue length log
    - search the right balance!
- Can be done in other ways
  - e.g.,: you can compute the service time beforehand
  - it requires you to store additional information
  - on arrival:
    - `queue.add({jobId=arrival.jobId, serviceTime=B})`
  - on service:
    - `scheduleEvent(service, queue[0].serviceTime)`
- Question:
  - what if I want to sample queue length with a constant sampling time?
  - e.g., t=0s length=0, t=1s length=0, t=2s length=1, t=3s length=4, …
  - without post-processing the current output file

An Example: Queue Simulator

```
on init:
    jobId = 0
    arrival.jobId = jobId
    scheduleEvent(arrival, now + A)
    queue = emptyQueue()
    logQueueLength(now, queue.length())
    scheduleEvent(logQueue, now + Ts)
```

- Input parameters:
  - arrival A
  - service B
  - queue length L
  - sampling time Ts
- Output:
  - queue length over time
  - jobs dropped over time
- Possible events:
  - arrival of a job
  - service of a job
  - log queue length

An Example: Queue Simulator

```
on event(event):
    if (event == arrival) {
        if (queue.length() < L OR L == 0) {
            if (queue.length() == 0)
                scheduleEvent(service, now + B)
            queue.add(arrival.jobId)
        } else {
            logDrop(now, arrival.jobId)
        }
        logQueueLength(now, queue.length())
        jobId++
        arrival.jobId = jobId
        scheduleEvent(arrival, now + A)
    } else if (event == service) {
        queue.removeFirst()
        logQueueLength(now, queue.length())
        if (queue.length() != 0)
            scheduleEvent(service, now + B)
    } else if (event == logQueue) {
        logQueueLength(now, queue.length())
        scheduleEvent(logQueue, now + Ts)
    }
}
```

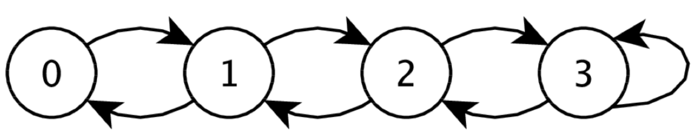
- Input parameters:
  - arrival A
  - service B
  - queue length L
  - sampling time Ts
- Output:
  - queue length over time
  - jobs dropped over time
- Possible events:
  - arrival of a job
  - service of a job
  - log queue length

What are the differences between the two approaches?

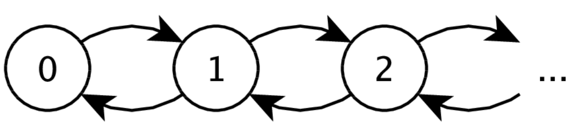
What Are We Doing?

- In practice, we are performing a random walk through the states of a Discrete Time (Semi-Markov) Chain
- For the queue example, the state is the number of jobs in the queue
- Transition probabilities depend on the distributions of arrival and service times
  - Might be simply unfeasible to compute for some distributions

Finite queue length

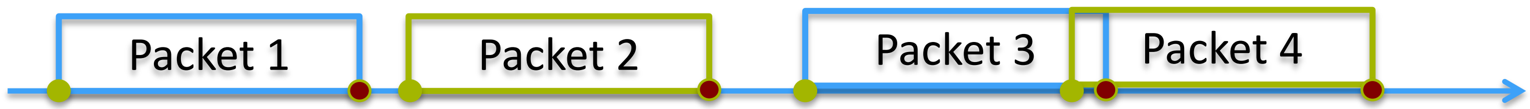


Infinite queue length



DES philosophy

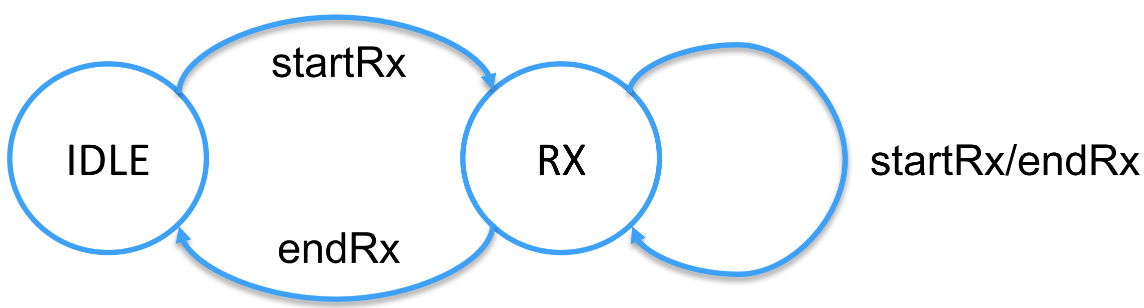
- Consider again a network simulation
  - managing collisions: when two packets overlap, they both can’t be received




```
onInit:
  state = IDLE
  recvPackets = {}
```

```
onStartRx(packet):
  recvPackets.add(packet)
  if (state == IDLE)
    state = RX
  else
    for p in recvPackets
      p.setLost()
```

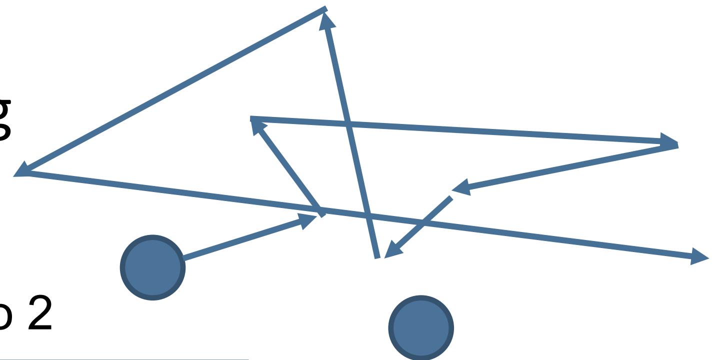
```
onEndRx(packet):
  if (not packet.isLost())
    sendUp(packet)
  recvPackets.del(packet)
  if (|recvPackets| == 0)
    state = IDLE
```



Repeatability and PRNGs

- Running the SAME simulation twice MUST give the same result
  - Statistical confidence is obtained through repetitions
  - Change the seed of PRNGs to obtain different runs
  - NEVER use a really random number to seed PRNGs (e.g., seed(time()))
    - Cannot reproduce results
    - Cannot reproduce bugs
  - Common practice: use repetition number as seed
- In general, use different PRNG instances for different random processes (see next slide)

- Example: imagine simulating a communication system where
  - One node is static, one randomly moves around
  - In one scenario the moving node sends one message per second
  - In the other it uses a random interval
- Assume we use a single PRNG extracting
  - 0.2, 0.5, 0.3, 0.1, 0.6, 0.9, 0.1, 0.8, 0.4, 0.7



Scenario 1

Position (x, y)	Interval (s)
(0.2, 0.5)	1
(0.3, 0.1)	1
(0.6, 0.9)	1
(0.1, 0.8)	1
(0.4, 0.7)	1

Scenario 2

Position (x, y)	Interval (s)
(0.2, 0.5)	0.3
(0.1, 0.6)	0.9
(0.1, 0.8)	0.4
(0.7, …)	…

PYTHON NETWORK SIMULATOR

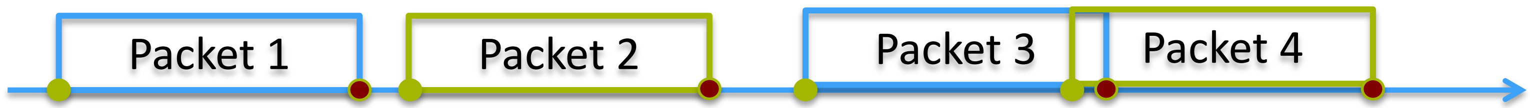
First SPE DES assignment

- Build a small discrete event network simulator
  - implementing a simple ALOHA protocol
  - given network topology (10 nodes)
- Assignment
  - draw the flow chart
  - write the simulator (with some characteristics)
  - run simulations
  - analyze system behavior (throughput, collisions)
- Problems discovered
  - not reading the assignment (e.g., config file, README text file)
  - programming!?!?
  - what is a flow chart???
  - too many concepts in one single assignment: easy to mess it up

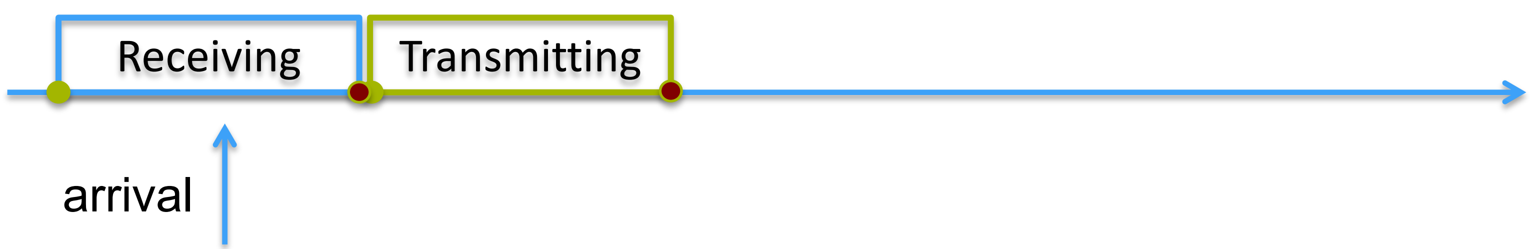
This year assignment

- We give you a small, home-made network simulator
- Goal:
  - extend the simulator to implement a protocol feature
  - analyze and compare the results w.r.t. standard implementation

- Very simple medium access protocol:
  - when you have a packet to send, send it!
  - no carrier sensing
  - if two (or more) packets overlap at a SPECIFIC receiver, they collide

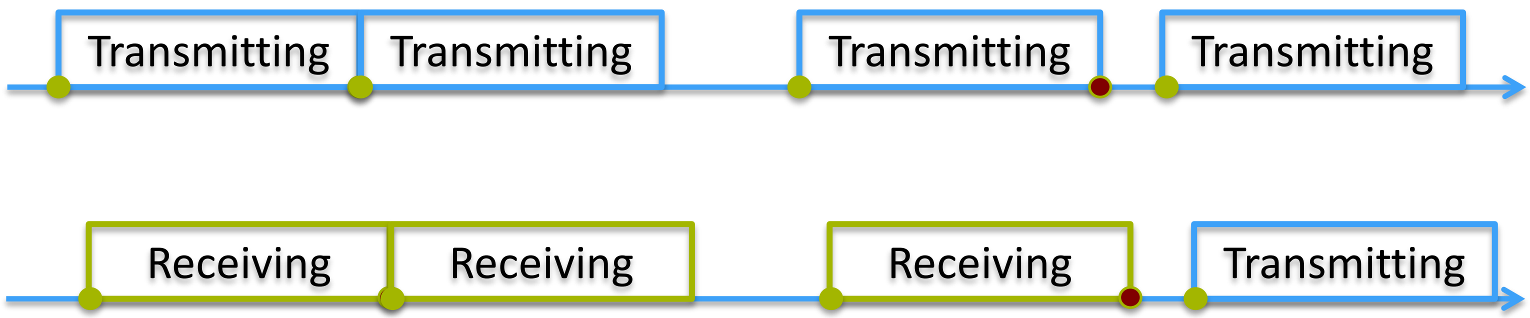


- In the simulator we also make additional assumption(s):
  - while receiving a packet (or more), we do not transmit (somewhat CSMA 1p)
  - some others (see later)



Events

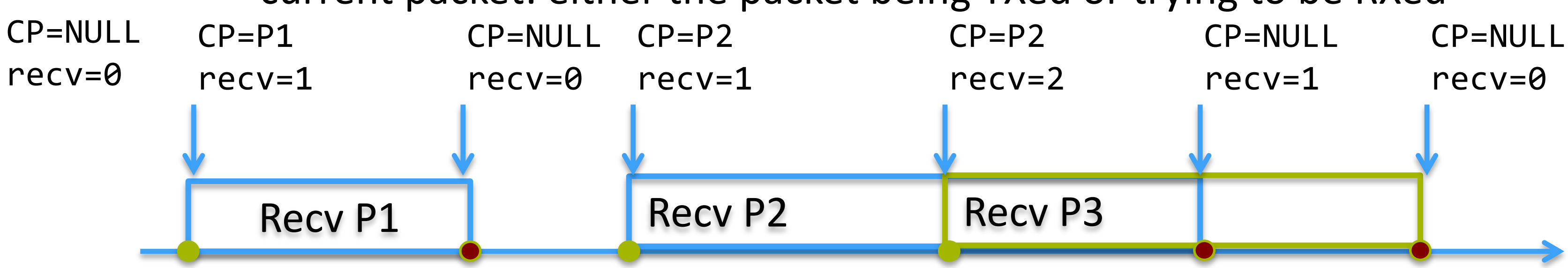
- arrival: a new packet to be sent is generated by a node
- end_tx: used by a node to know when it's done transmitting
- start_rx: notifies the node the beginning of an incoming packet
- end_rx: notifies the node the end of an incoming packet
- end_proc: used by a node to know when processing is over
  - used to avoid channel capture
  - example: imagine two nodes having always a packet to transmit



- rx_timeout: used to avoid getting stuck into reception (see later)

Nodes' state and variables

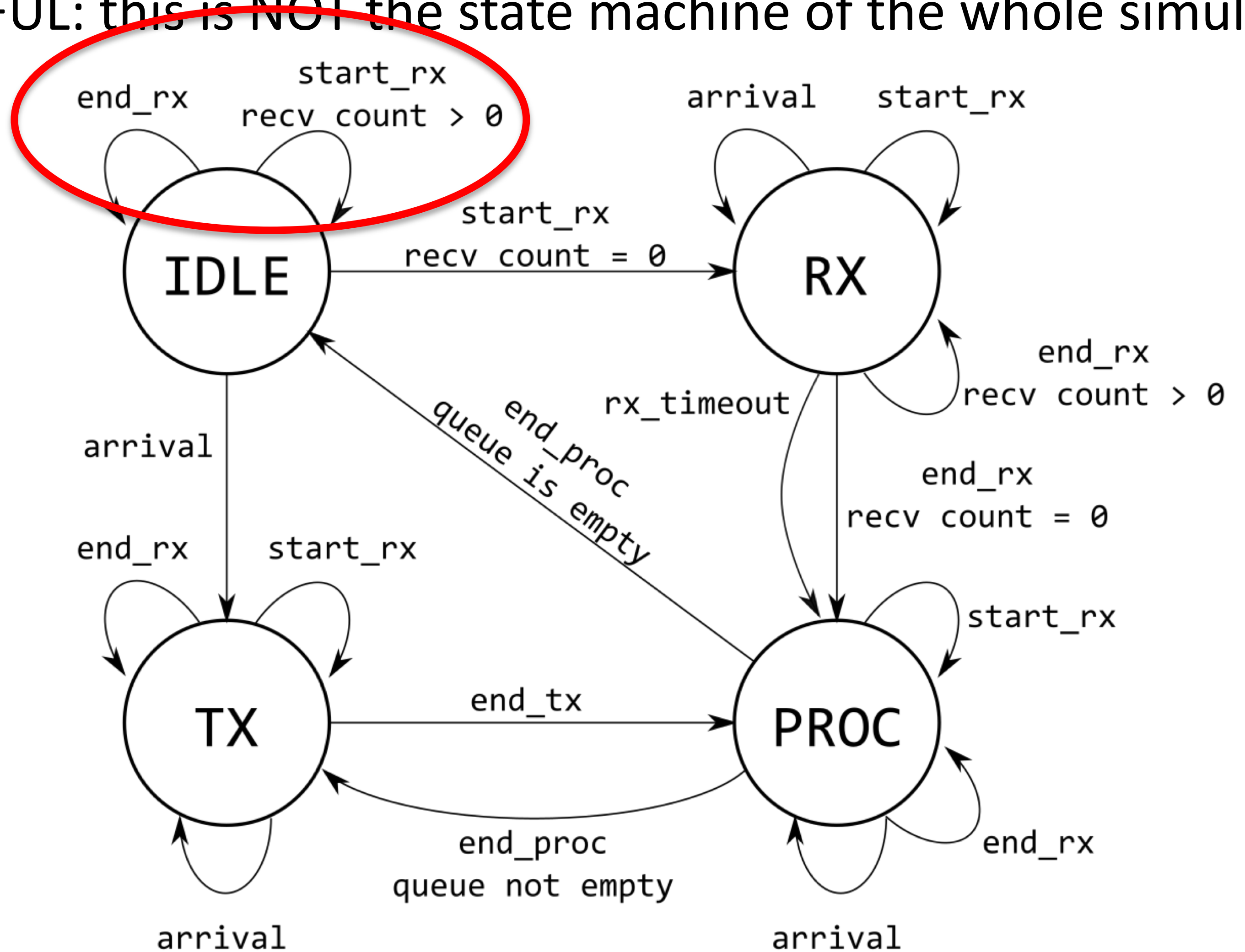
- Each node in the simulator has a current state, plus some variables
  - States:
    - IDLE: the node THINKS the channel is free
    - TX: the node is currently transmitting a packet
    - RX: the node is currently receiving one (or more!) packets. It is aware that there is something being transmitted in the channel
    - PROC: the node is performing a little processing after a TX or an RX
  - Variables:
    - queue: queue of packets that needs to be sent
    - recv count: number of packets in the air
    - current packet: either the packet being TXed or trying to be RXed



State transitions

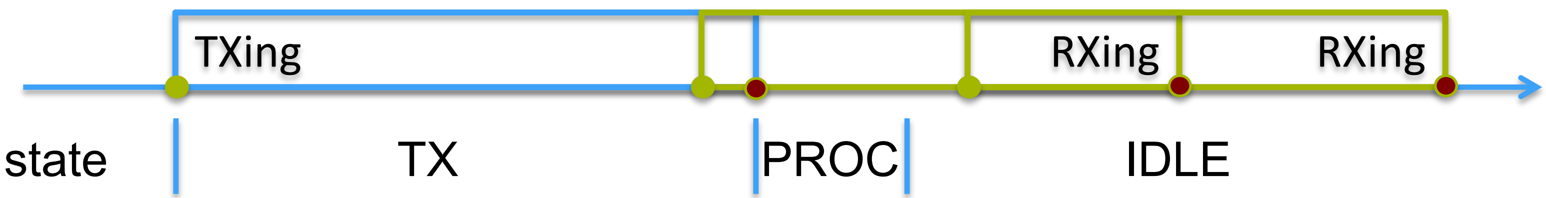
- CAREFUL: this is NOT the state machine of the whole simulator

???

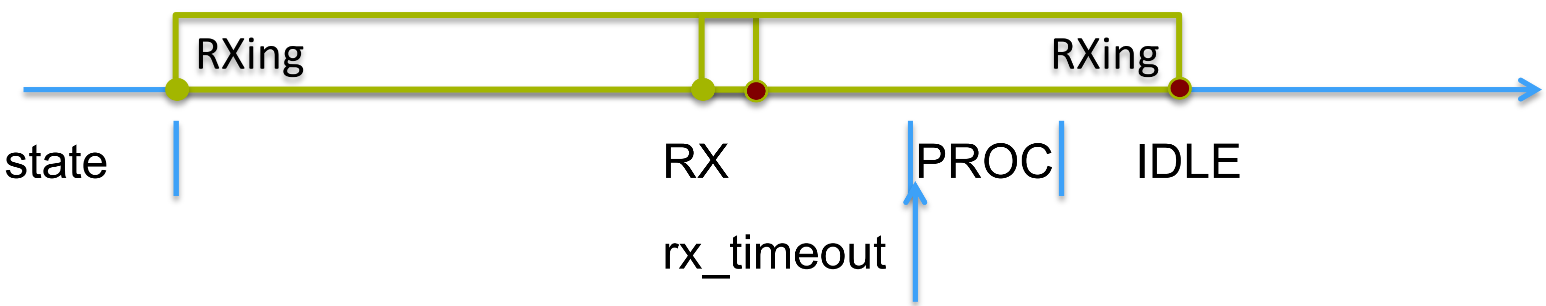


end_rx while being IDLE

- We assume base ALOHA, performing no carrier sensing
  - while transmitting, we assume a node is not able to detect an incoming packet. In this condition, a new packet is also not detected

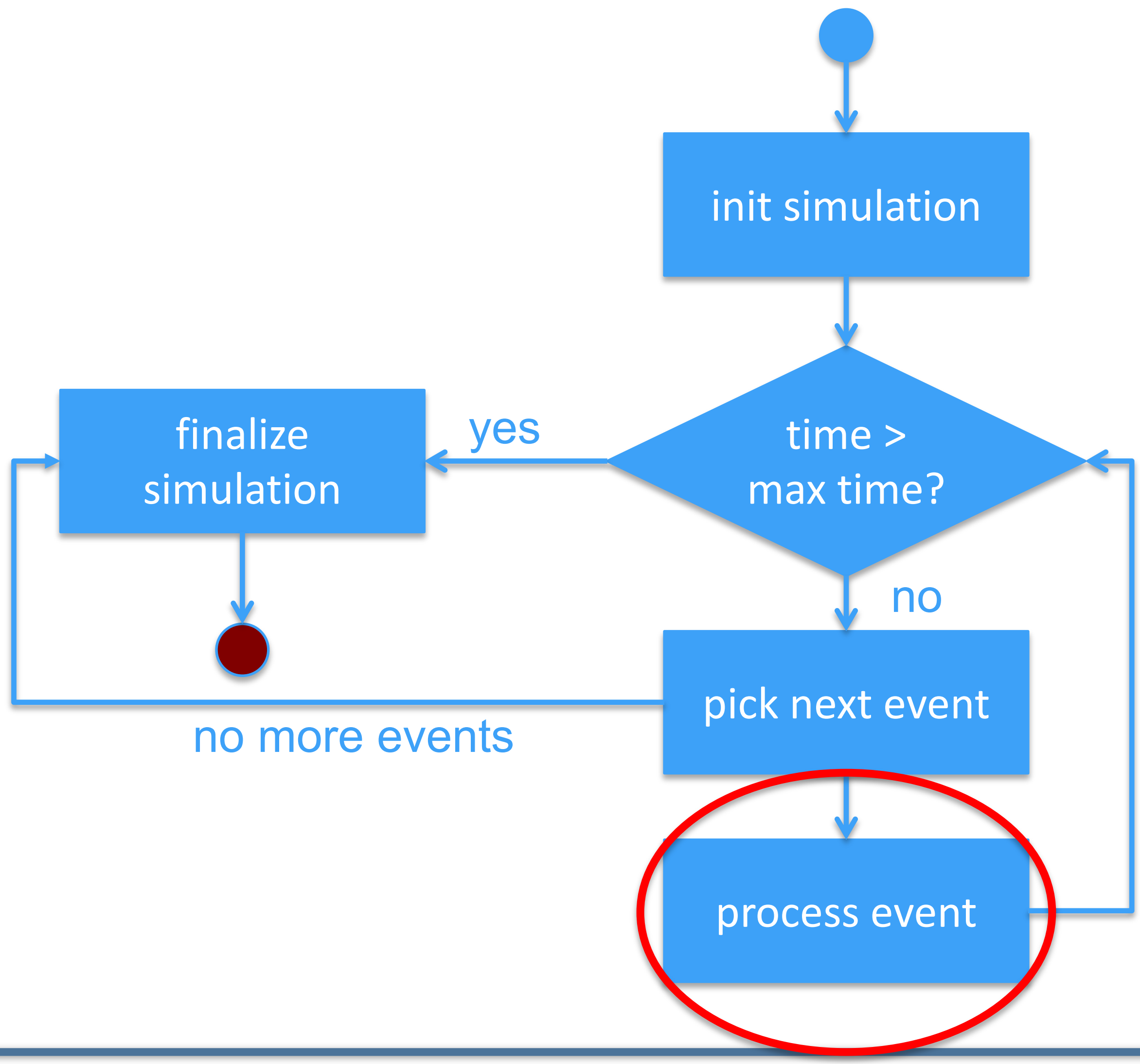


  - while receiving multiple packets, we assume a node does not know when the first packet ends, but we consider an RX timeout corresponding to the maximum packet size (plus a small delta)

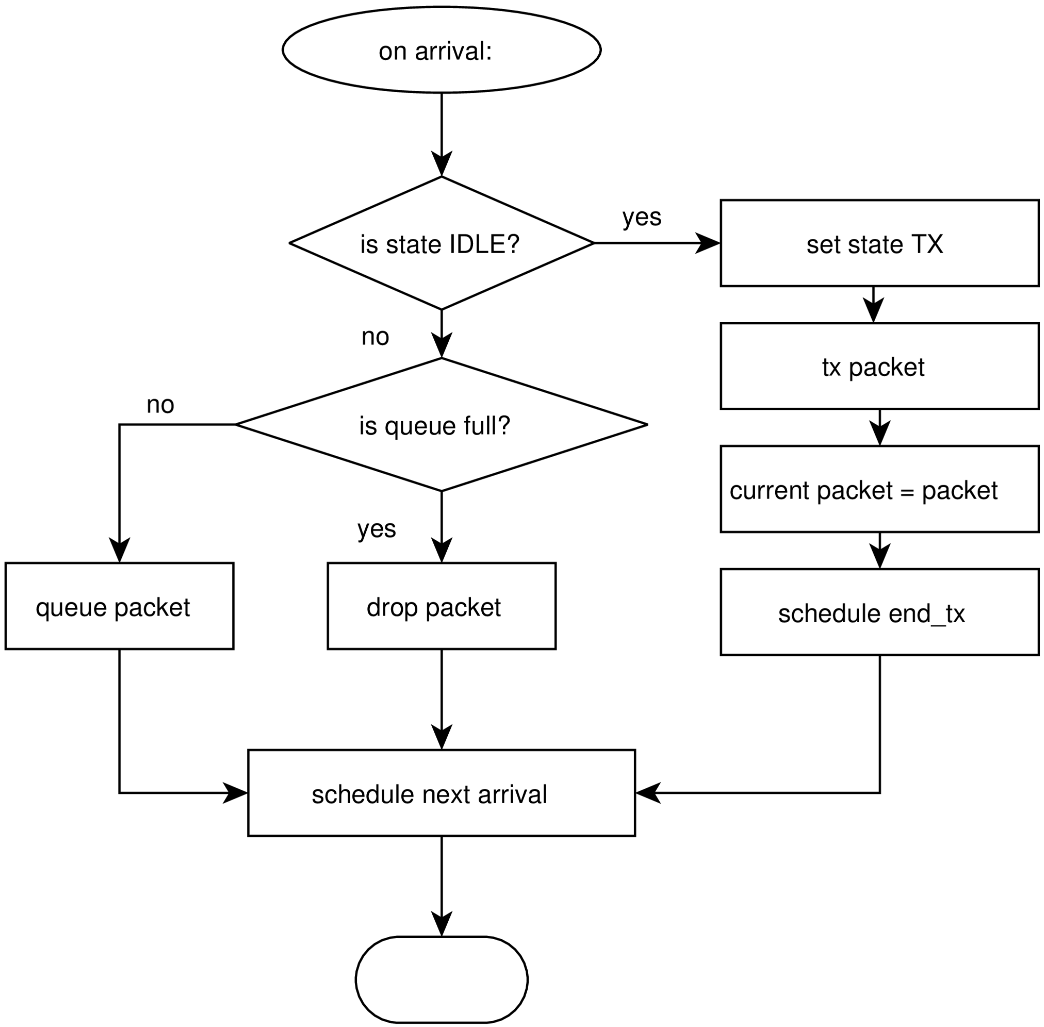


THESE ARE ALL ASSUMPTIONS!

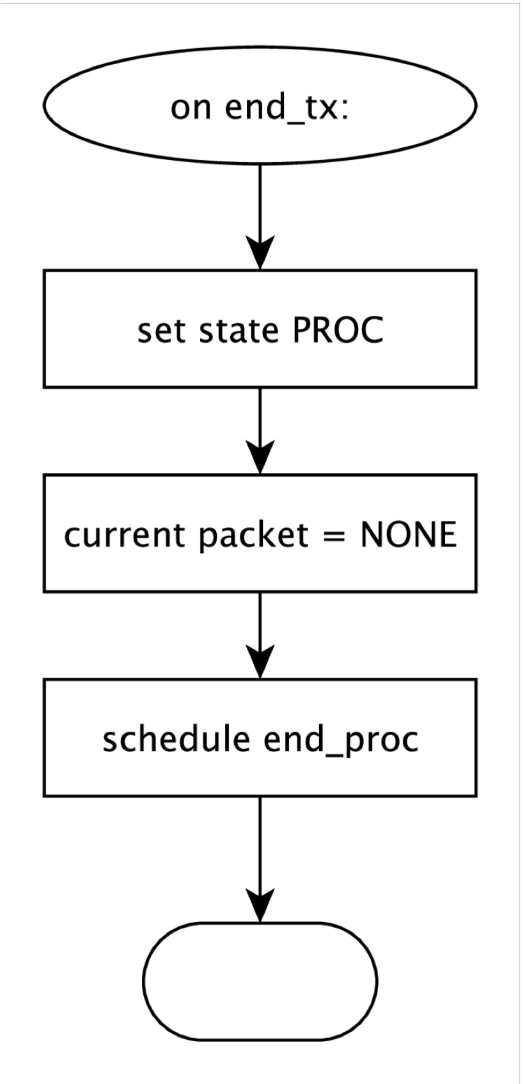
Recall: DES Flow chart



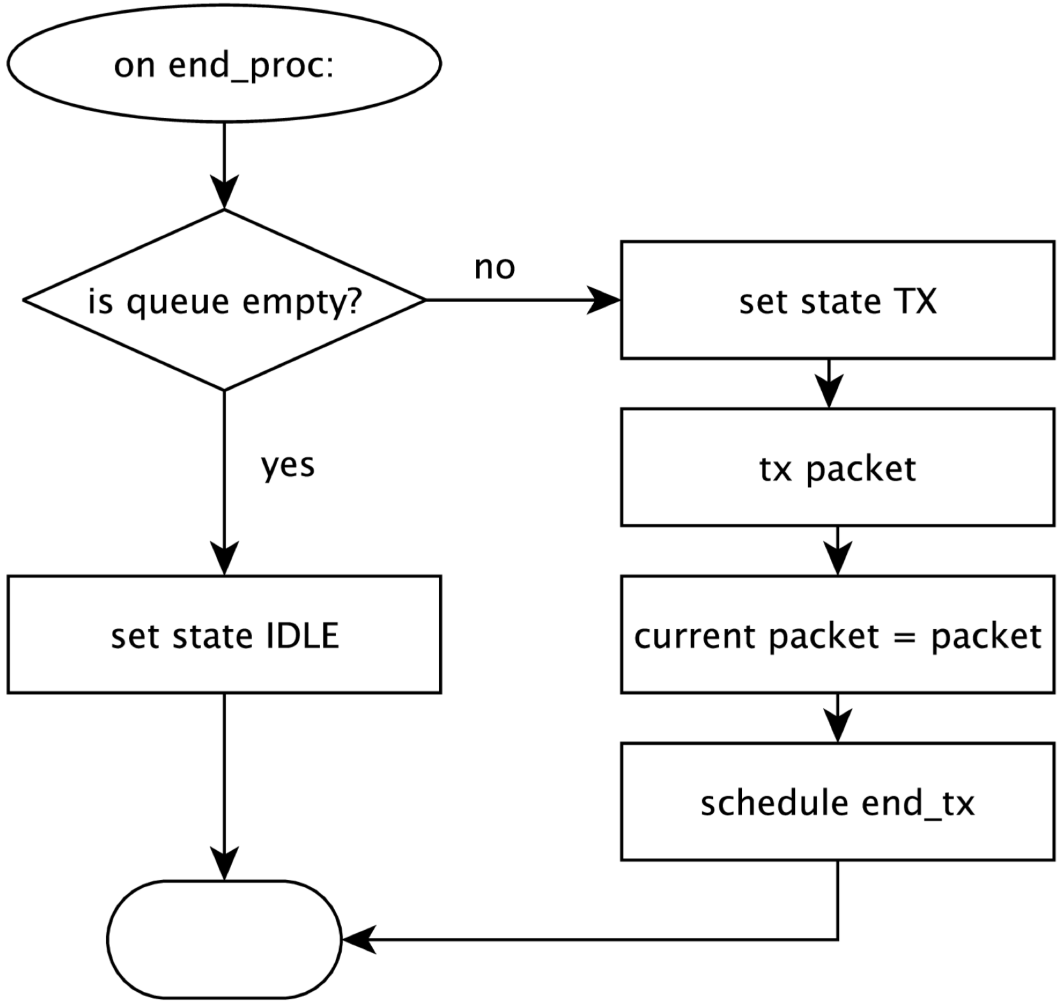
Flow chart: arrival event



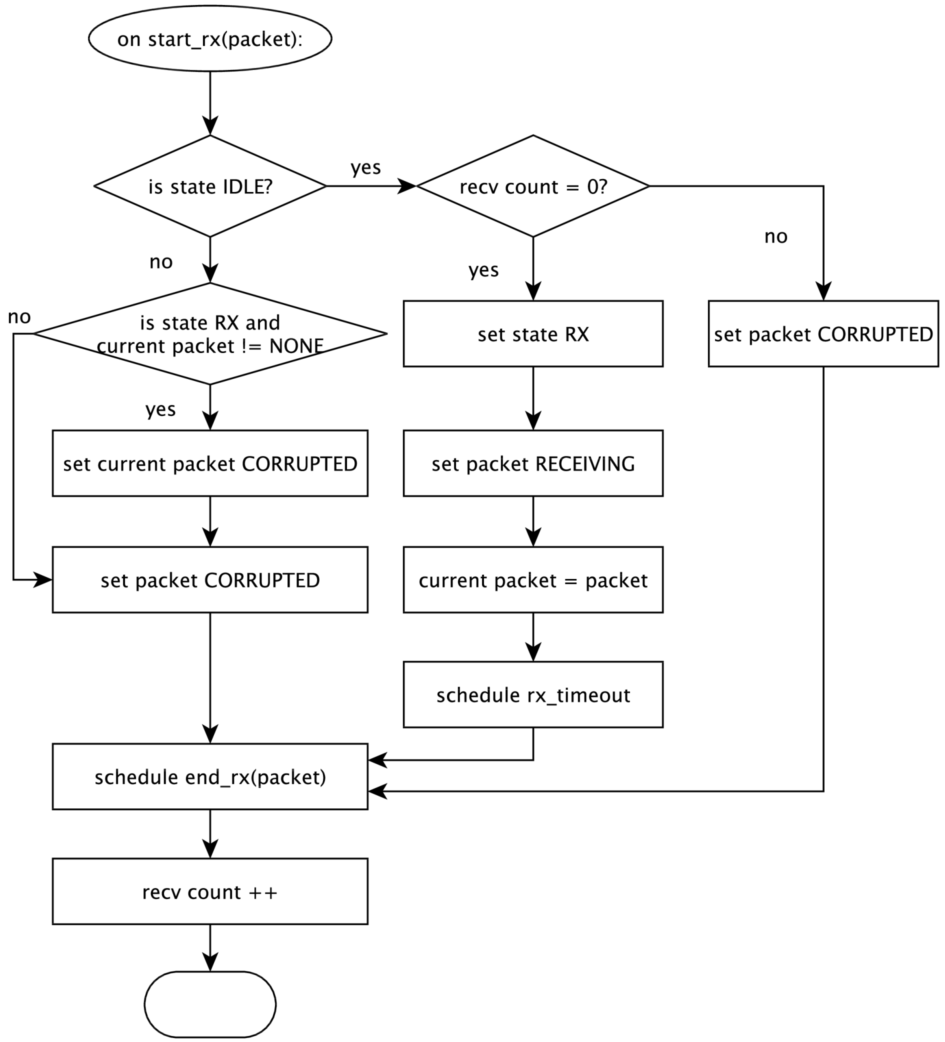
Flow chart: end_tx



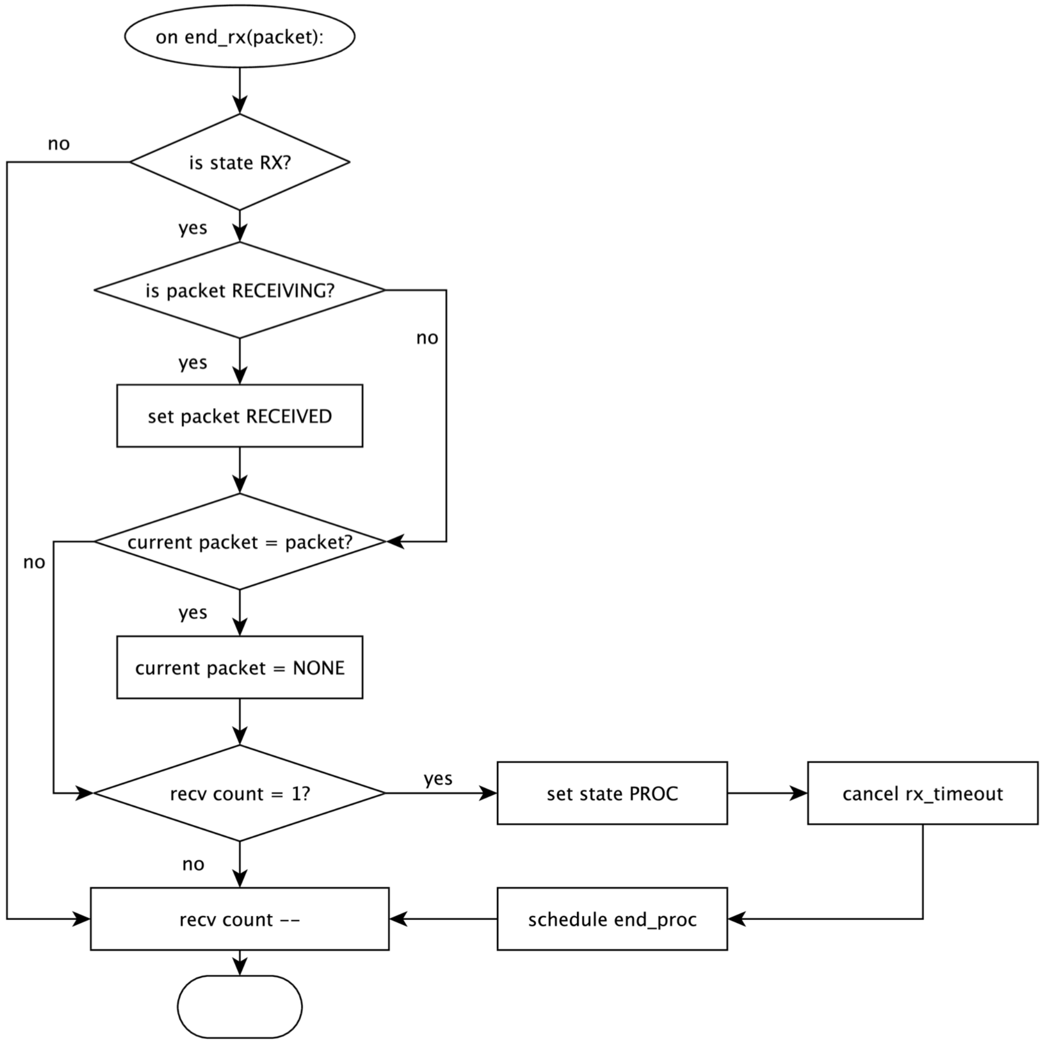
Flow chart: end_proc



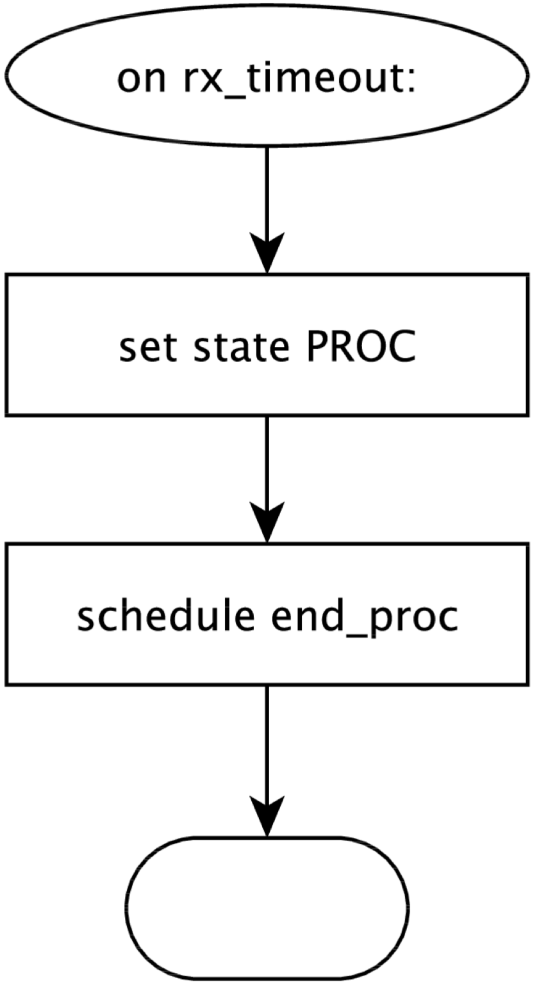
Flow chart: start_rx



Flow chart: end_rx



Flow chart: on_rxtimeout



The simulator: Files

- Written in Python
  - no need to compile
  - cross platform
  - one of the most known scripting languages
- Files
  - main.py: script to actually run the simulator
    - ./main.py -l : get the list of all runs
    - ./main.py -L : list of all runs with associated parameters
  - sim.py: event manager and scheduler that runs the simulation
  - channel.py: class that, when beginning transmission, schedules the start_rx event to all nodes within communication range
  - node.py: implements the logic of a node (all the flow charts so far)
  - distribution.py: random distributions used in the simulator
  - other files, not at the core
- Output: csv file with list of packet events

The simulator: Downloading

- Downloading a zip file
  - The zip file is (will be) published in classroom
- Cloning the git repository
  - git clone https://ans.disi.unitn.it/redmine/spe-network-simulator.git
  - master branch: simulator only
  - plot branch: simulator plus process.R script
  - exploit git to track your changes, maybe in different branches
- Updated today (13 May 2019)
  - some bugfixes
  - python 2 and 3 compatibility

Configuring the simulator: config.json

```
{    "simulation" : {
        // seed(s) to initialize PRNGs
        "seed" : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        // duration of each simulation in seconds
        "duration" : 30,
        // communication range in meters
        "range" : 10,
        // physical layer datarate in bits per second
        "datarate" : 8000000,
        // packet queue size. set to 0 for infinity
        "queue" : 2,
        // packet inter-arrival distribution in 1/seconds
        "interarrival" : [
            {"distribution" : "exp", "lambda" : 10},
            [...]
            {"distribution" : "exp", "lambda" : 1510}
        ],
        // packet size distribution in bytes
        "size" : {"distribution" : "unif", "min" : 32, "max" : 1500, "int" : 1},
        // maximum packet size in bytes to compute the RX timeout
        "maxsize" : 1500,
        // processing time after end of reception or transmission before starting operations again
        "processing" : {"distribution" : "const", "mean" : 0.000001},
        // position of nodes, list of x,y pairs
        "nodes" : [
            [[1,1], [2,3], [0, 0]]
        ],
        // log file name using configuration parameters
        "output" : "output_{interarrival.lambda}_{seed}.csv"
    }
}
```

Coding style

- Coding follows Python style guide:
  - https://www.python.org/dev/peps/pep-0008/
  - indentation: 4 spaces
  - width 80 characters
  - some more things
  - try to keep the same style as much as possible
- Code is well documented:
  - function purpose
  - function parameters
  - reason for particular choices
  - DOCUMENT YOUR CODE AS WELL

Simulation setup:

- set of nodes transmitting packets
  - size uniformly distributed within 32 and 1460 B
  - exponentially distributed inter-arrival times with lambda from 10 to 1510 arrivals/s
  - 8 Mbps physical layer bitrate
  - queue size: 2 packets

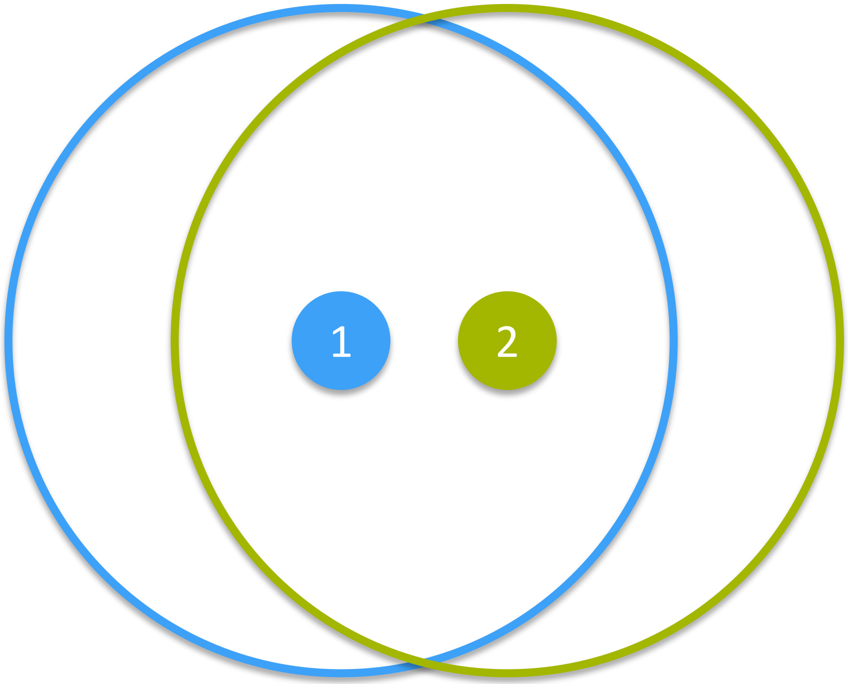
DISCLAIMER

The following plots are given as an example. They might not be suitable for a formal report!

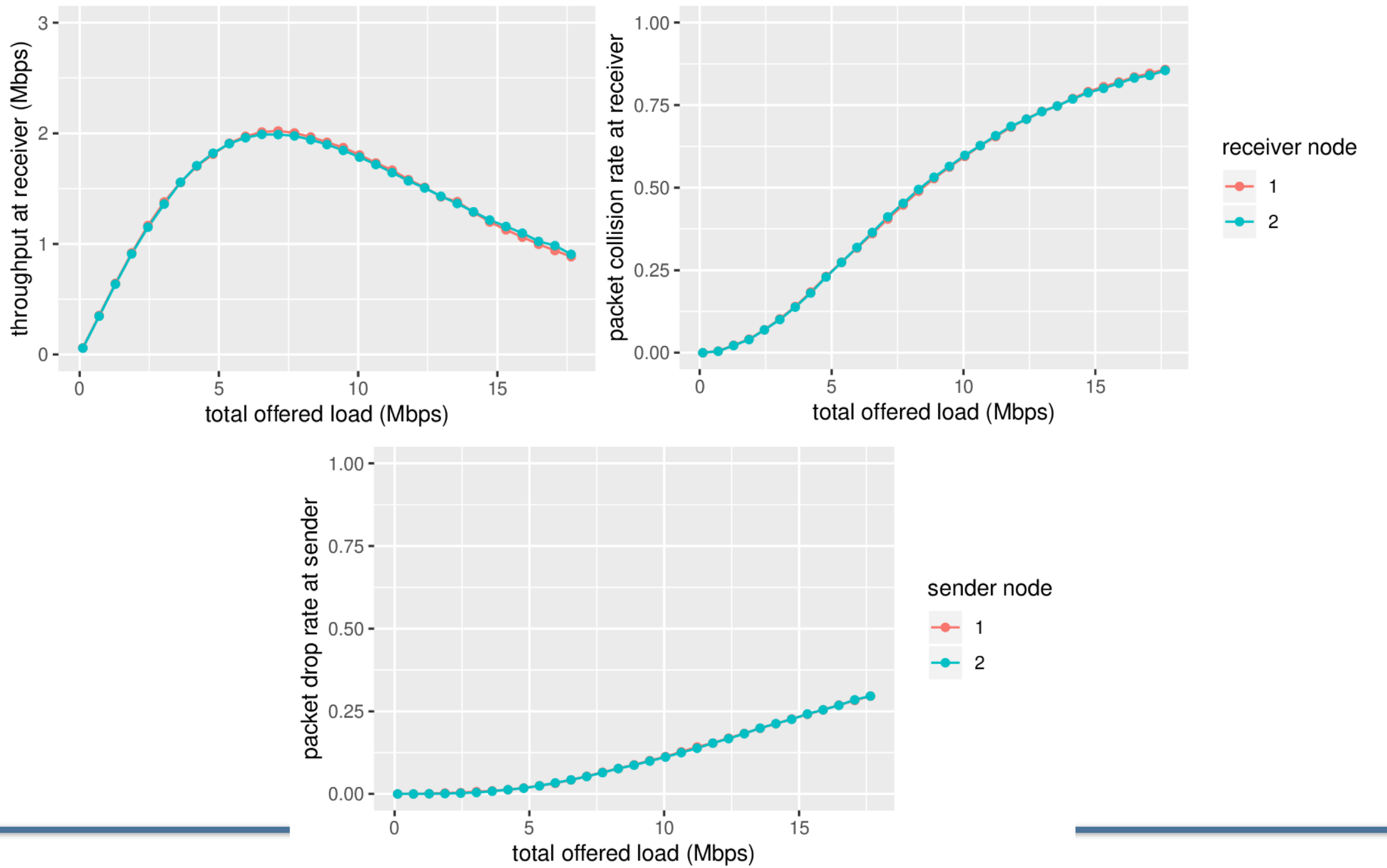
Metrics

- Total offered load: sum of the offered load from all stations
  - lambda * (32+1500)/2 * N * 8 / 1024^2 (Mbps)
- Throughput at receiver: correctly received bytes over simulation time
  - sum the size of all the packets marked as “RECEIVED” and divide it by the simulation time
- Collision rate at receiver: ratio of collided packets over total incoming packets
  - N.CORRUPTED / (N.CORRUPTED + N.RECEIVED)
- Drop rate at sender: ratio of packet dropped at the queue over total generated
  - N.DROPPED / N.GENERATED

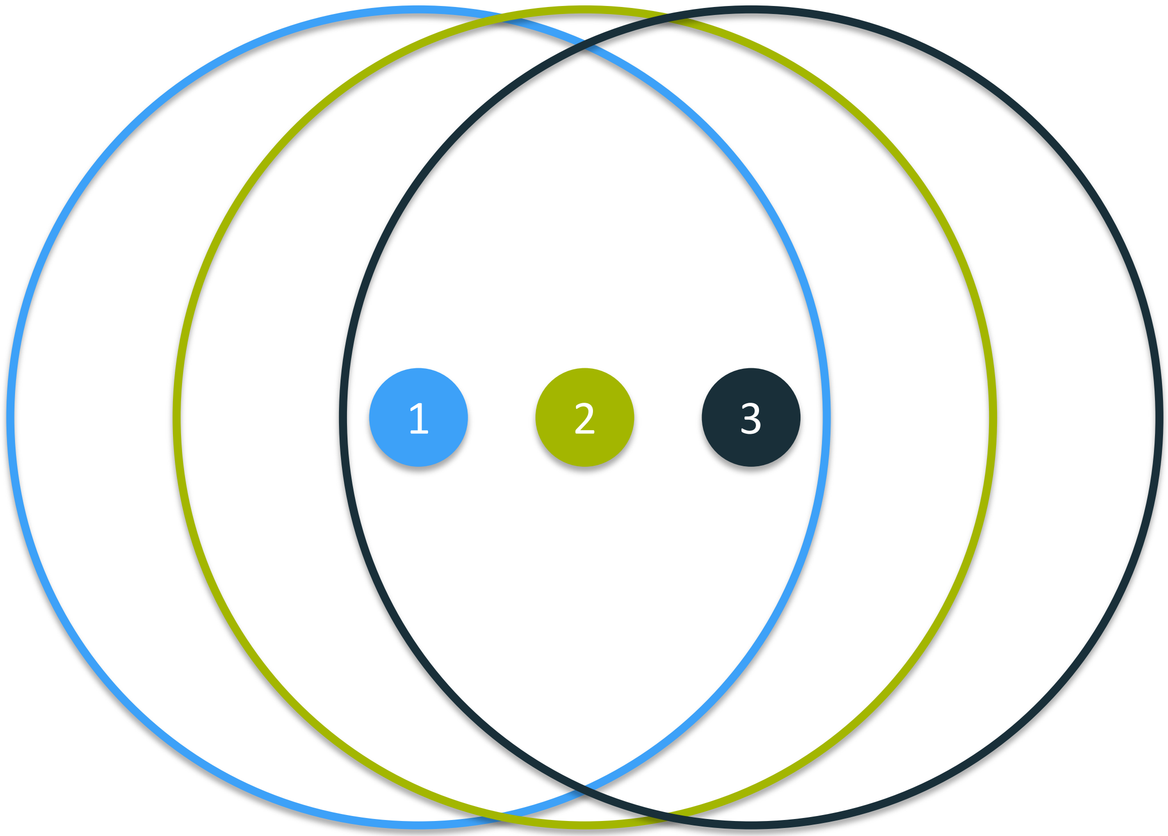
Some results: 2 nodes

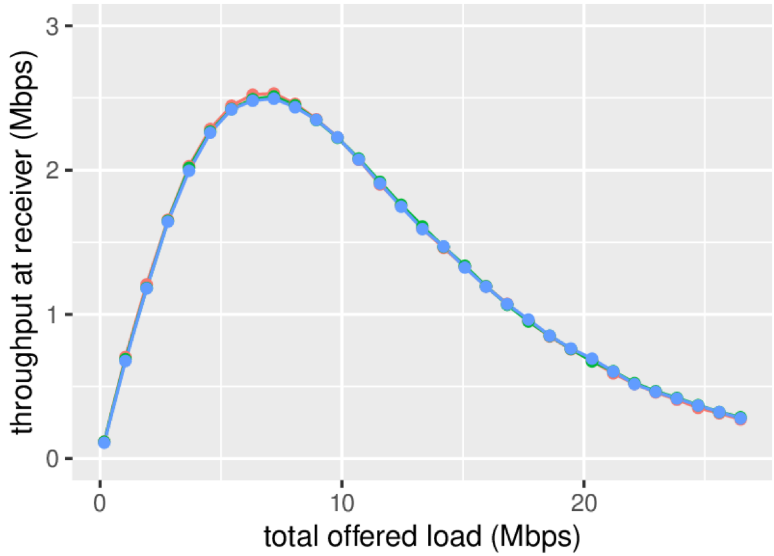


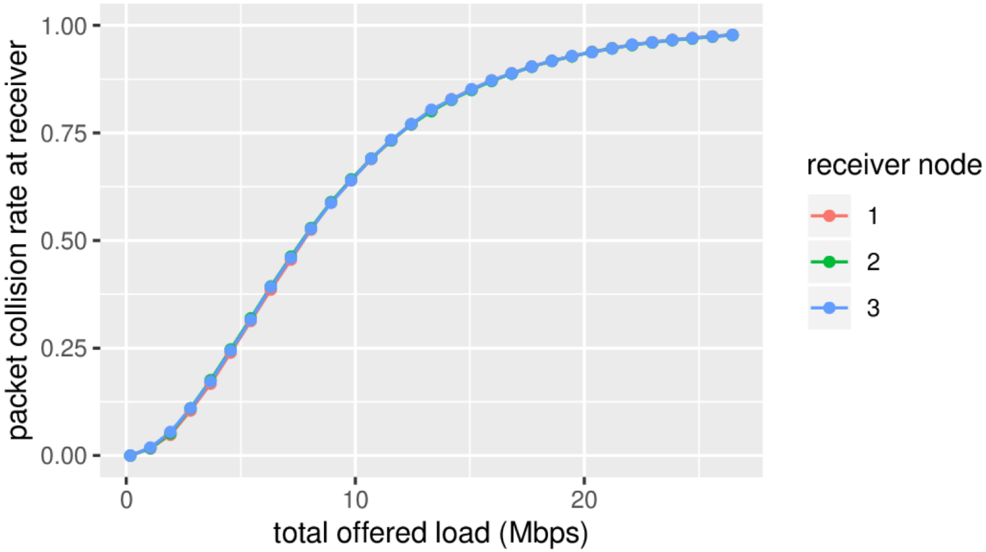
Some results: 2 nodes

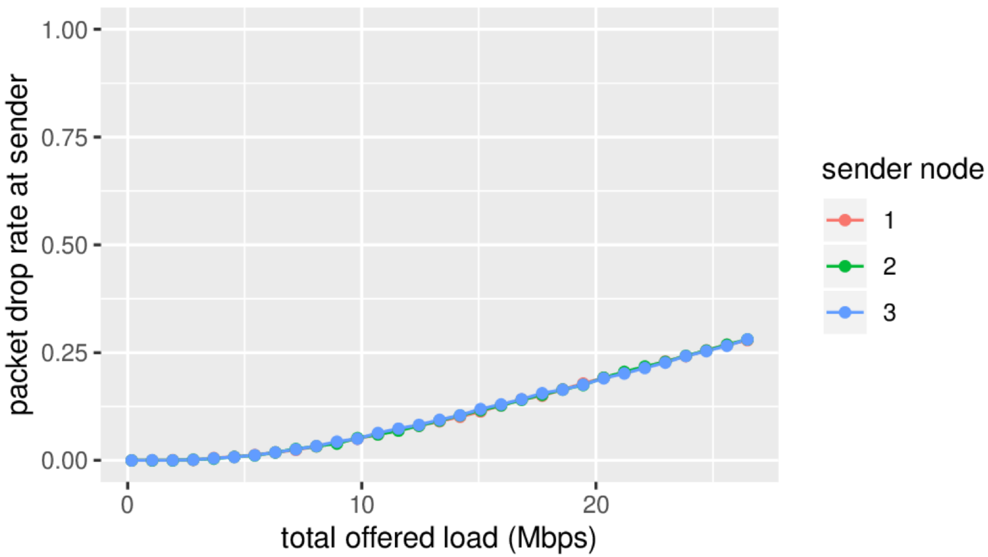


Some results: 3 nodes

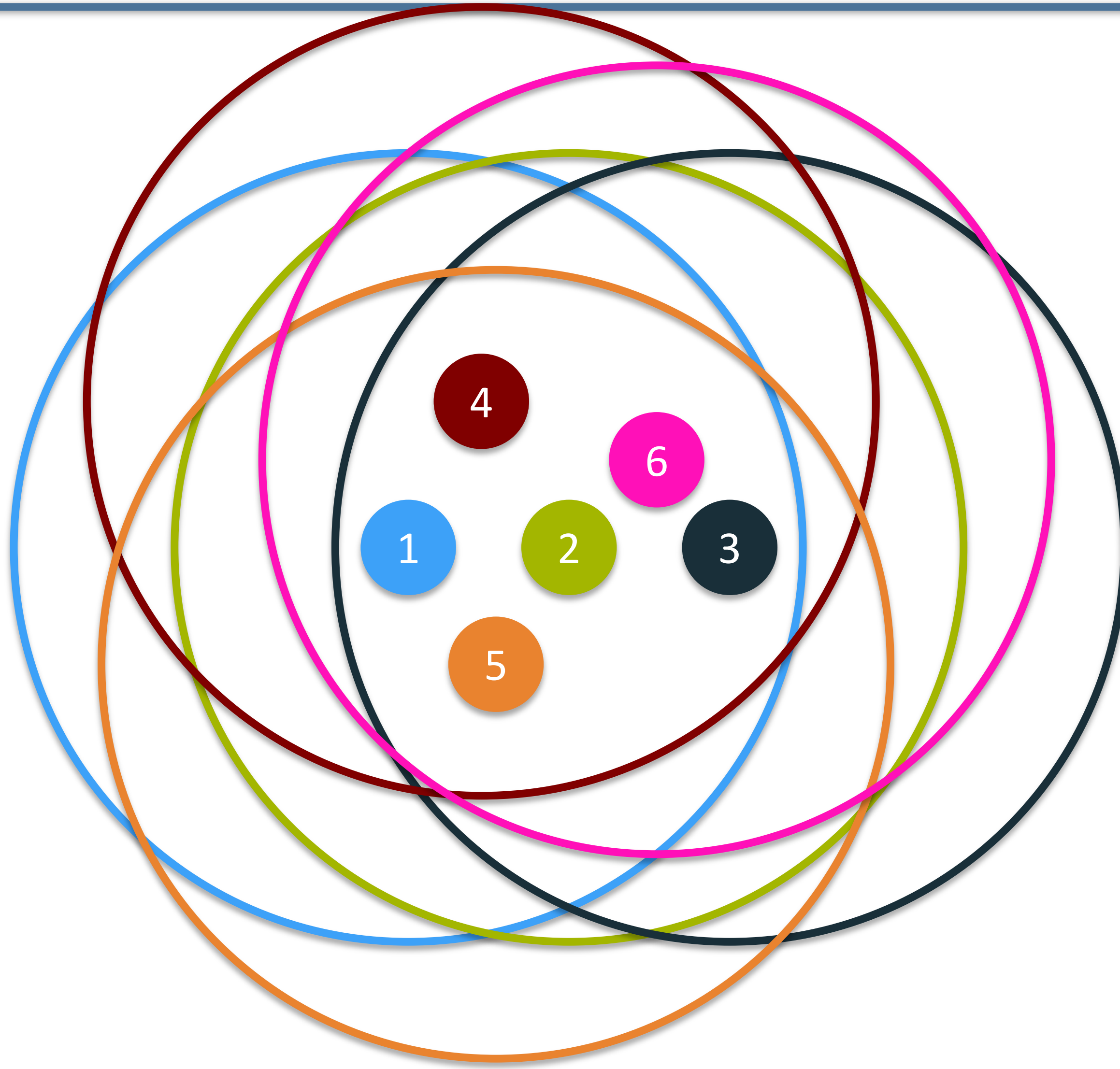



throughput at receiver (Mbps)
0
1
2
3
0
10
20
total offered load (Mbps)

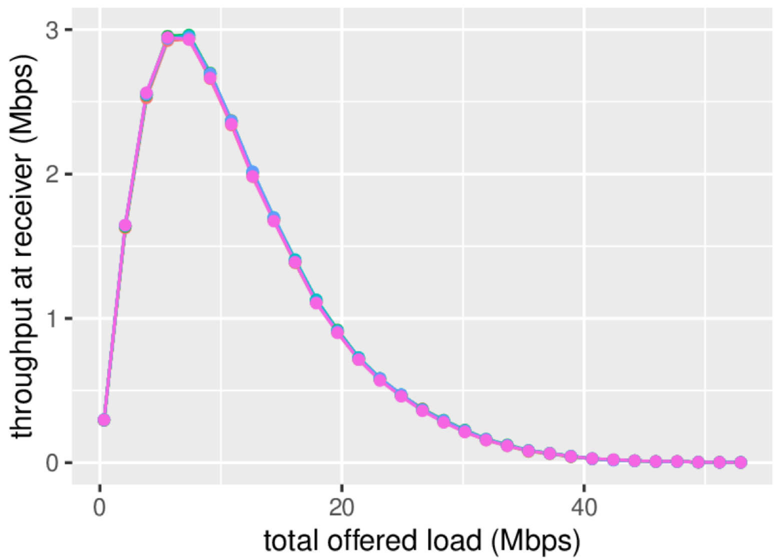

packet collision rate at receiver
0.00
0.25
0.50
0.75
1.00
0
10
20
total offered load (Mbps)
receiver node
1
2
3

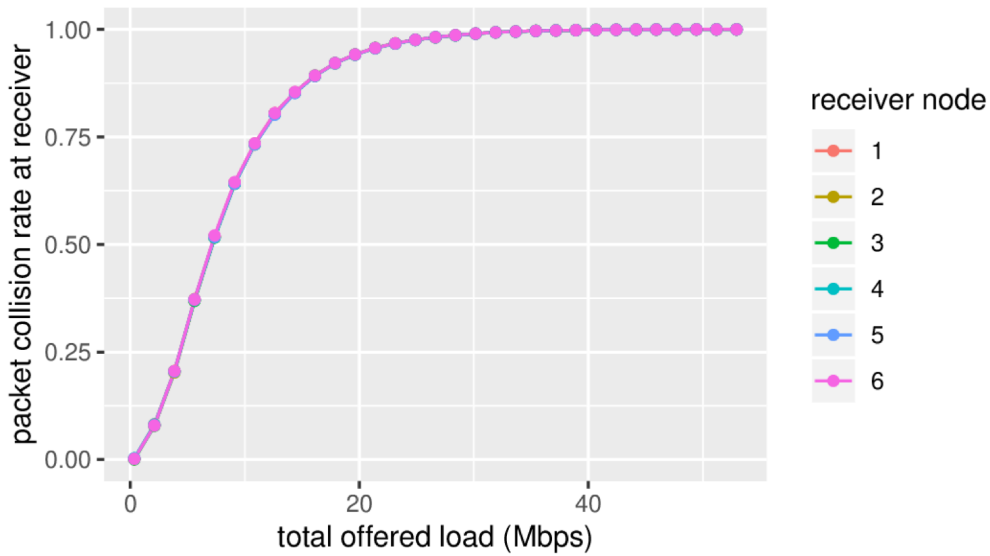

packet drop rate at sender
0.00
0.25
0.50
0.75
1.00
0
10
20
total offered load (Mbps)
sender node
1
2
3

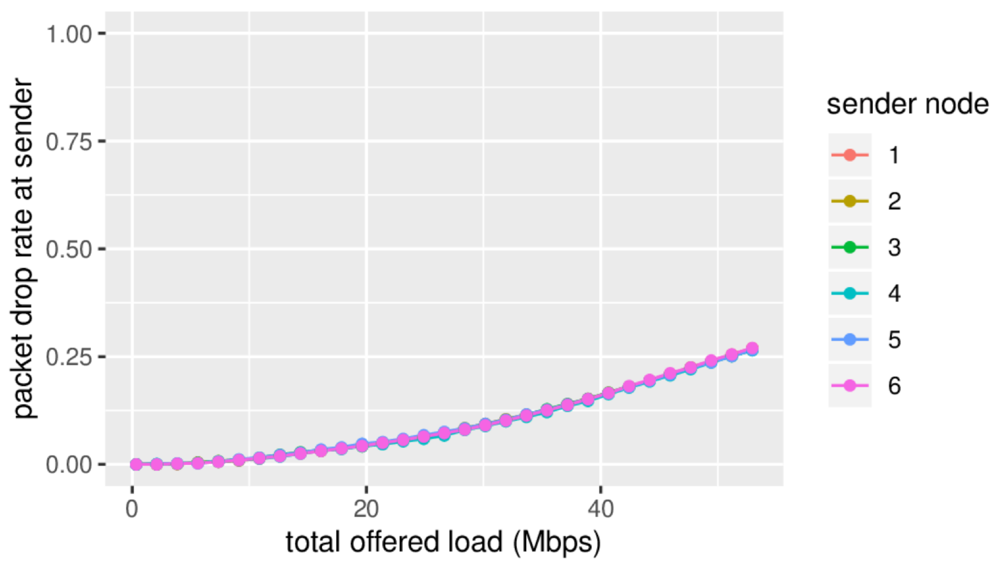
Some results: 6 nodes



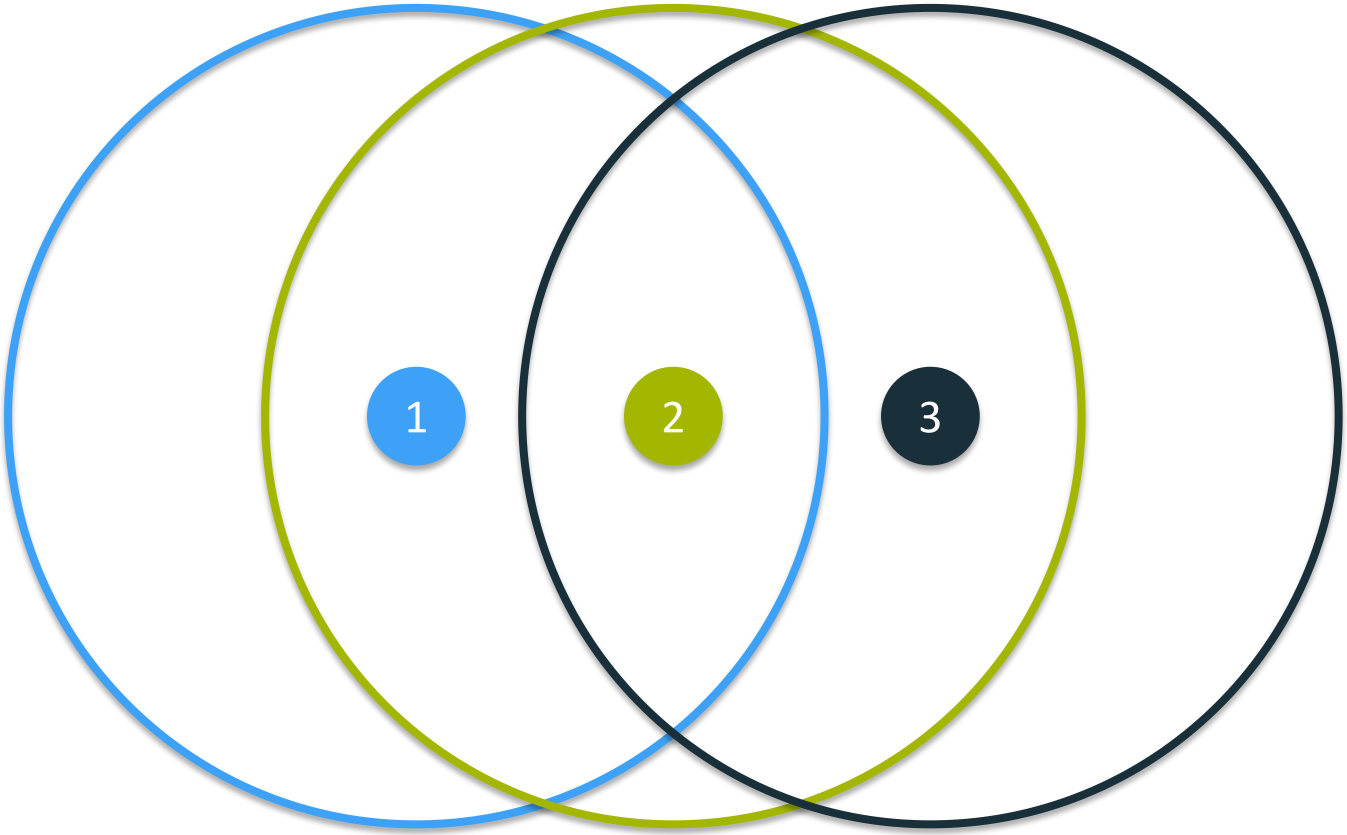
Some results: 6 nodes



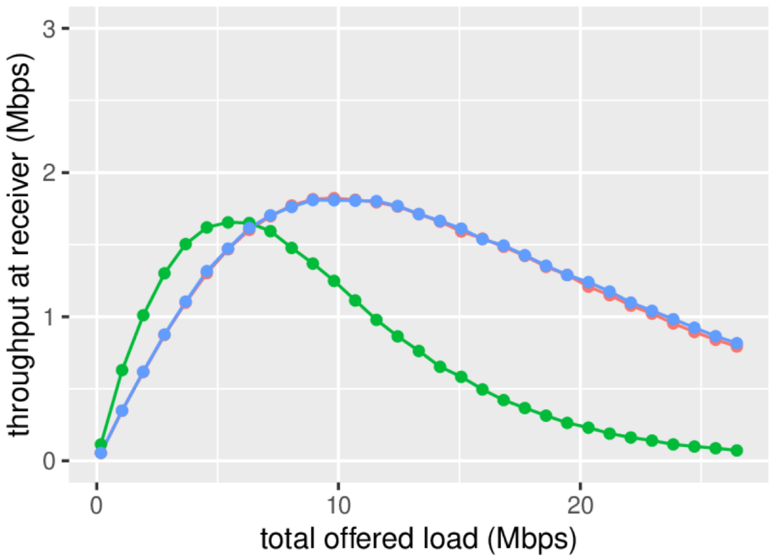


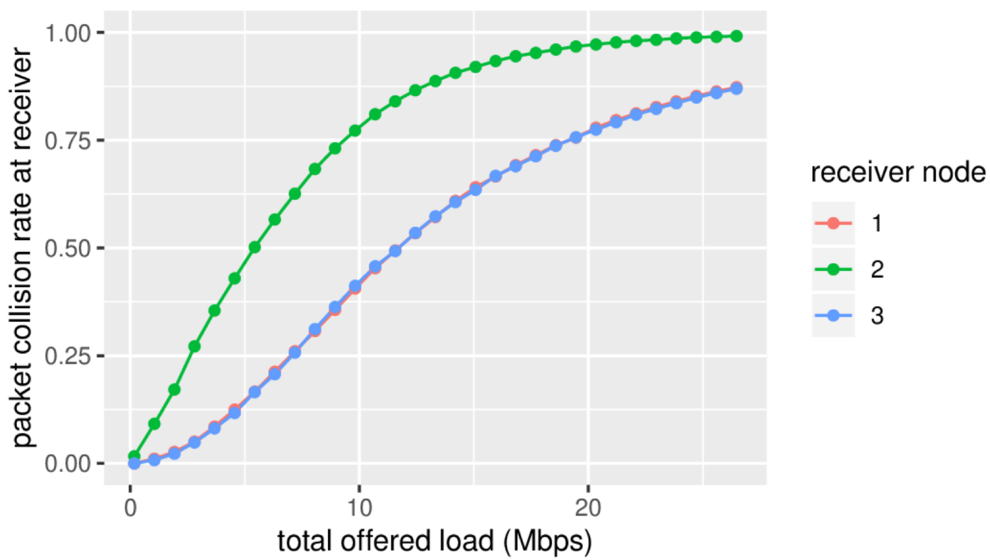


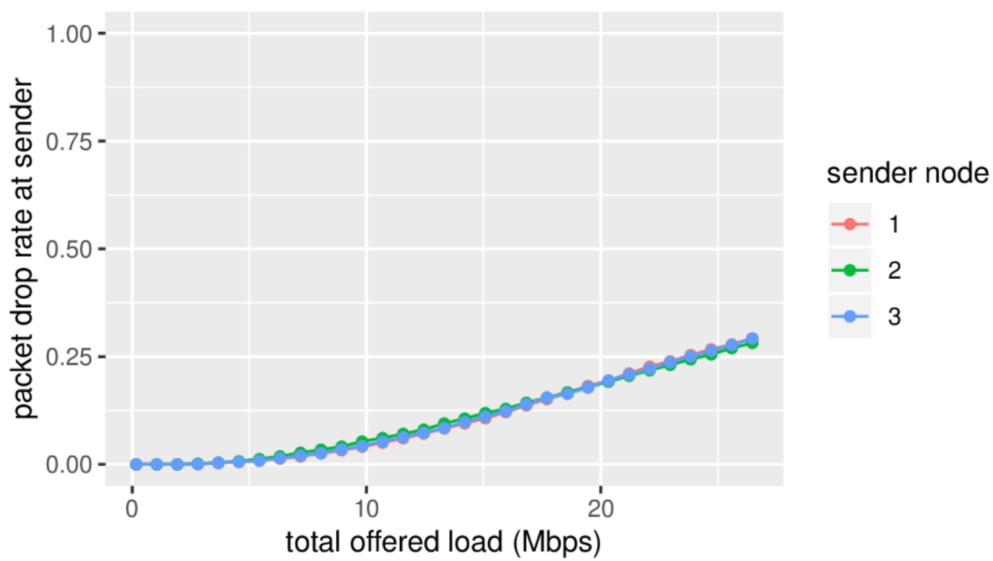
Some results: 3 nodes



Some results: 3 nodes







For the time being

- Download the simulator from classroom

- Play around with it
  - ./main.py -h
  - run some simulations
  - try to get some plots
  - get familiar with the code

- Is the code 100% bug free?
  - thoroughly tested, but can't never be sure ☹
  - if you find something strange, let me know!!