COMN - Computer Communications and Networks
Assignment 2: Implement & Analyse Sliding Window Protocols

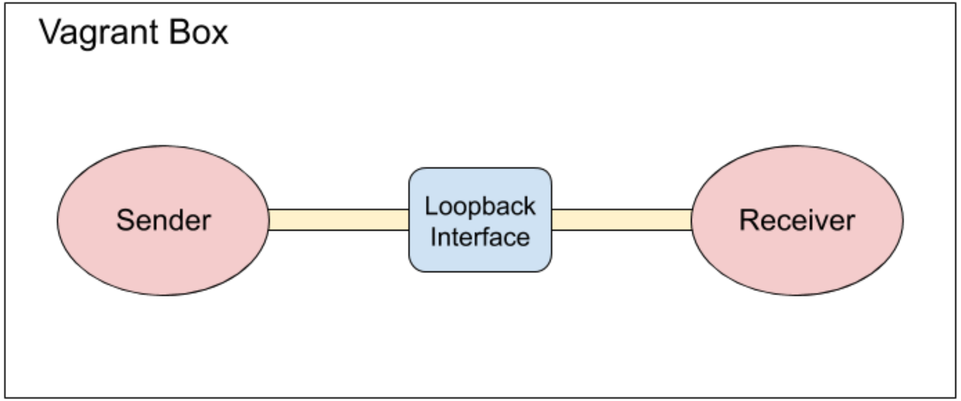
Jon Larrea and Mahesh K. Marina
School of Informatics
University of Edinburgh

15/02/2022

Coursework Overview

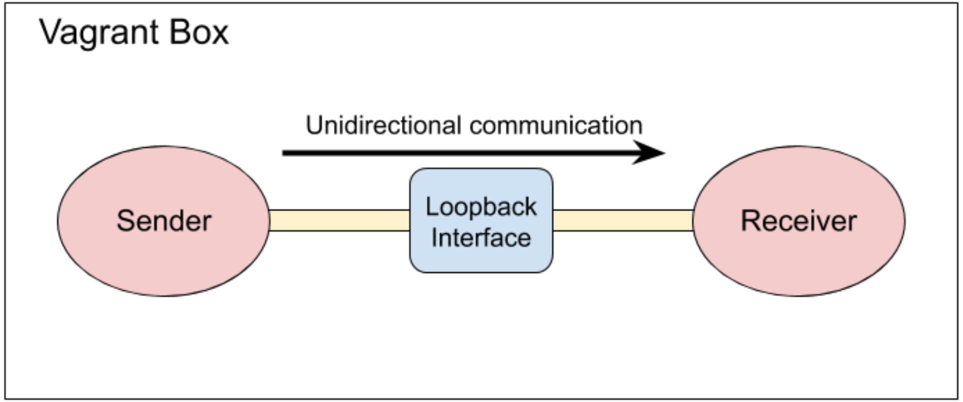
- Goal
  - Implementation and evaluation of three end-to-end reliable data transfer protocols
    - Stop-and-Wait, Go-back-N, and Selective Repeat
- Assessment: 35% of course mark
  - Part 1 (5%): rdt1.0
  - Part 2 (10%): rdt 3.0 (Stop-and-Wait)
  - Part 3 (10%): Go-back-N
  - Part 4 (10%): Selective Repeat + iPerf experiment

Conceptual Structure



- Linux Traffic Control (The *tc* utility)
  - Allows you to modify the packet scheduler for a given interface
  - Configuration of interface characteristics (bandwidth, delay, loss)
  - Command-line program: tc

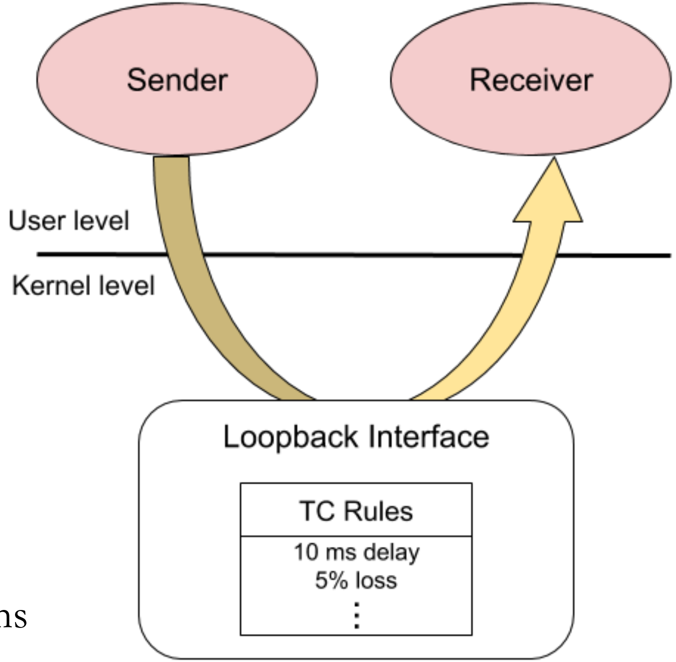
Conceptual Structure



- Sender
  - Reads the file and breaks it into a number of packets
  - Sends the packets to a receiver over the loopback interface which has the forwarding rules modified
- Receiver
  - Receives the packets; extracts data in the packets; and saves the data in a file

Packet flow

- In our scenario, sender and receiver processes within the same host (or more precisely, within the same virtual machine) communicate with each other



% sudo tc qdisc add dev lo root netem loss 5% delay 10ms

Header format

- The following formats should be used across all parts
  - Exception: no ACK packets needed in part 1
- Data packet
  - (Sender to Receiver)

0	1	2	3 ~ up to 1026
16-bit sequence number		8-bit EoF flag	Data

- ACK packet
  - (Receiver to Sender)

0	1
16-bit sequence number	

Useful tools: iPerf

- **iPerf** is a tool used to measure network performance measurement in terms of throughput and latency.

Client

```
openair@openair-1:~$ iperf -c 192.168.4.5 -i1 -t10
------------------------------------------------------------
Client connecting to 192.168.4.5, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.4.10 port 34562 connected with 192.168.4.5 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  11.2 MBytes  94.4 Mbits/sec
[  3]  1.0- 2.0 sec  11.2 MBytes  94.4 Mbits/sec
[  3]  2.0- 3.0 sec  11.1 MBytes  93.3 Mbits/sec
[  3]  3.0- 4.0 sec  11.2 MBytes  94.4 Mbits/sec
[  3]  4.0- 5.0 sec  11.2 MBytes  94.4 Mbits/sec
[  3]  5.0- 6.0 sec  11.1 MBytes  93.3 Mbits/sec
[  3]  6.0- 7.0 sec  11.2 MBytes  94.4 Mbits/sec
[  3]  7.0- 8.0 sec  11.1 MBytes  93.3 Mbits/sec
[  3]  8.0- 9.0 sec  11.2 MBytes  94.4 Mbits/sec
[  3]  9.0-10.0 sec  11.1 MBytes  93.3 Mbits/sec
[  3]  0.0-10.0 sec   112 MBytes  93.9 Mbits/sec
```

Server

```
openair@openair-1:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

Useful tools: iPerf

```
iperf -c 192.168.4.5 -i1 -t10
```

```
iperf -c 192.168.4.5 -i1 -n 30MB
```

```
iperf -c 192.168.4.5 -i1 -F test.jpg -M 1KB
```

- -c → Receiver IP address
- -i → Interval, seconds between periodic bandwidth reports
- -t → time in seconds to transmit for (default 10 secs)
- -n → number of bytes to transmit (instead of -t)
- -F → input the data to be transmitted from a file
- -M → set TCP maximum segment size

Useful tools: tcpdump

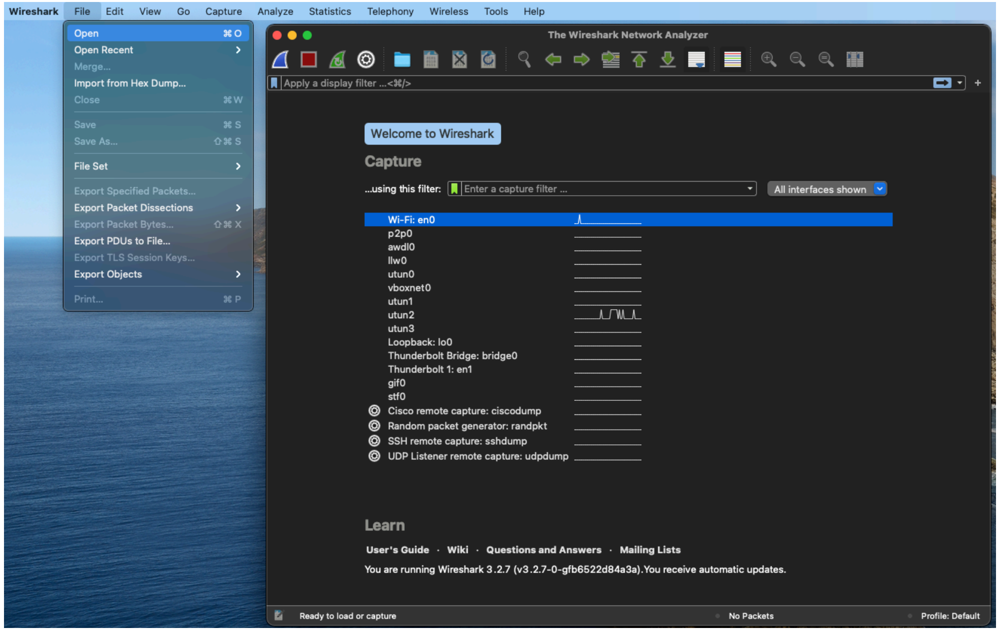
- **tcpdump** is a data-network packet analyser tool. (Command line version of **Wireshark**).
- Due to the lack of a GUI in the COMN VM, we cannot run **Wireshark** in the VM. We can however obtain the packet capture trace from inside the VM using **tcpdump**, then examine it outside the VM on our host machine using **Wireshark**.

```
sudo tcpdump -i lo -w out.pcap
```

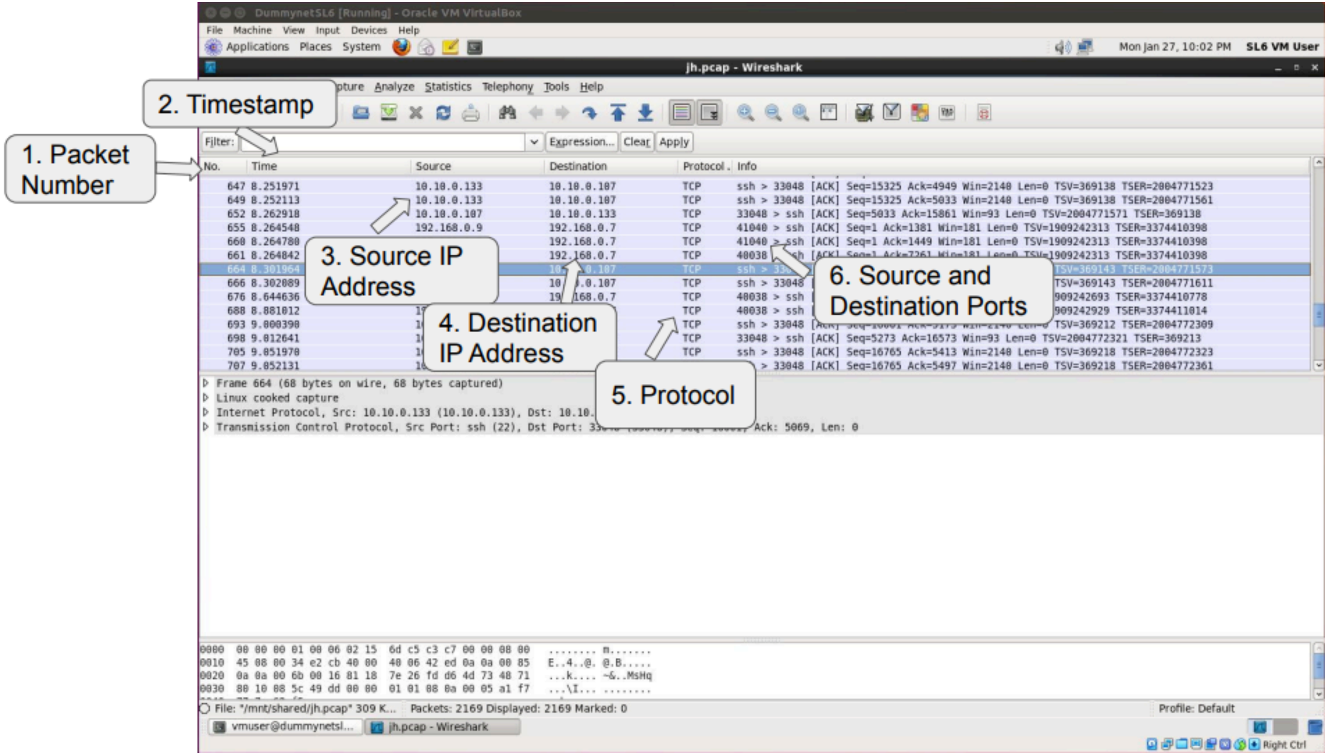
- -i → Interface (loopback).
- -w → Write the raw packets to a file.

Useful tools: Wireshark

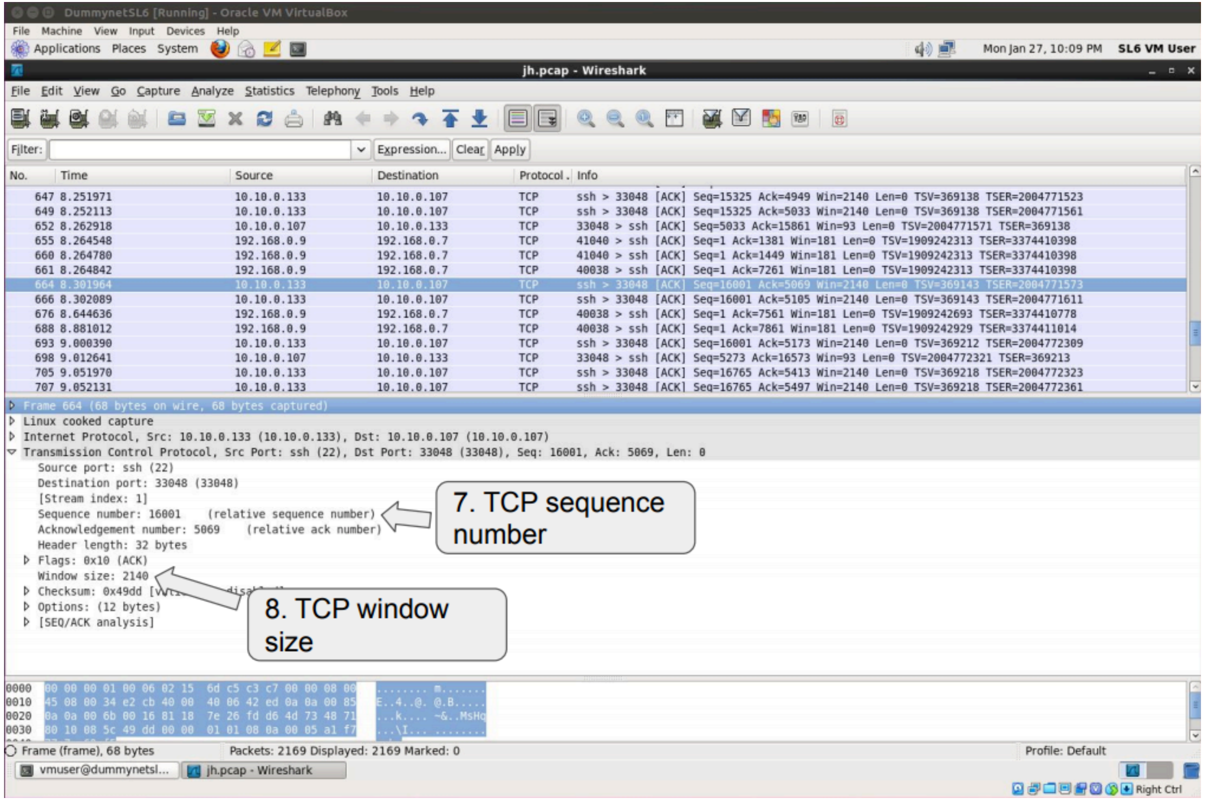
- **Wireshark** is an open-source packet analyser tool that is used to capture network packets to understand and troubleshoot network behaviour.



Useful tools: Wireshark



Useful tools: Wireshark



Miscellaneous

- Some useful Python libraries for the assignment:
  - sys
  - socket
  - math
  - time
  - Thread from threading
  - Lock from threading

Design choices for Part 3 and 4

- Both sender and receiver are implementable without multithreading
  - Definitely no need for multithreading at the receiver side
  - Multithreading may be useful for sender implementation
- Use non-blocking socket for non-multithreaded implementation
  - setblocking(0) and select()
- Many design choices for the sender are possible

What is multithreading?

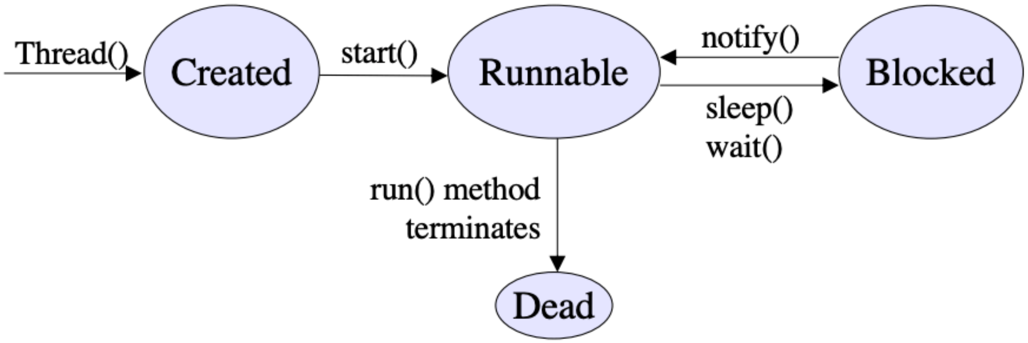
- Multithreading is similar to multi-processing
- A multi-processing OS can run several processes at the same time
  - Each process has its own address/memory space
  - Separate processes do not have access to each other's memory space
- In a multithreaded application, there are several points of execution **within the same memory space**
  - Each point of execution is called a thread
  - Threads share access to memory

Thread support in Python

- Python threading allows you to have different parts of your program running concurrently.
- Threads are represented by a Thread object
  - A thread object maintains the state of the thread
  - It provides control methods such as start, run, sleep, join
- When an application executes, the main method is executed by a single thread
  - If the application requires more threads, the application must create them

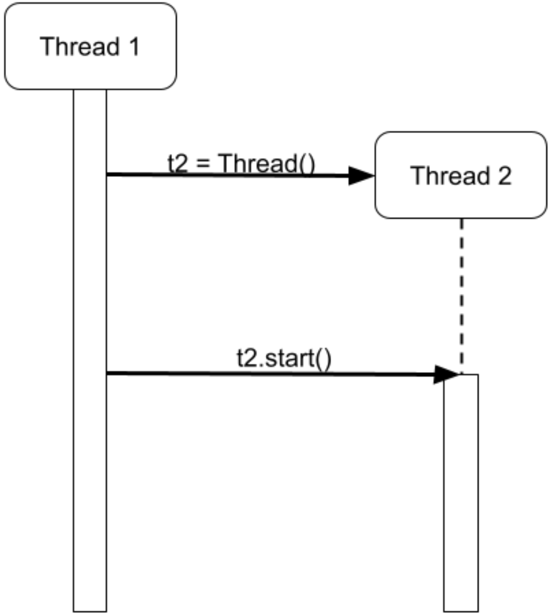
Threads States

- Threads can be in one of four states as shown in the figure below
- A thread’s state changes based on:
  - Control methods such as start, sleep, wait, notify
  - Termination of the run method



How does a thread run?

- The thread class has a run() method
  - run() is executed when the thread's start method is invoked
- The thread terminates if the run method terminates
  - run() method often has an endless loop to prevent thread termination
- One thread starts another by calling its start method



Creating your own thread

- The Python standard library provides **threading**, which contains all the primitives you'll need for this assignment.

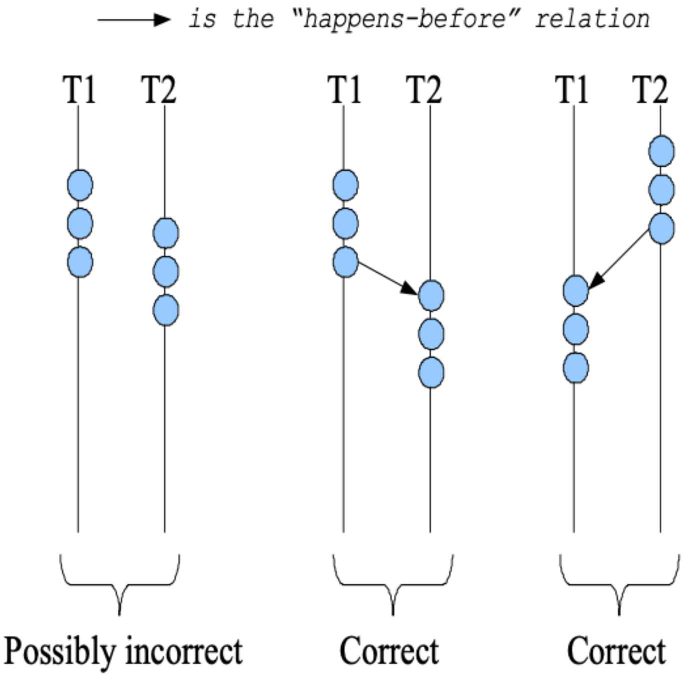
```python
import threading
import time

def thread_function(name):
    print('Hello from ' + name)

    for i in range(5):
        print(i)
        time.sleep(1)

if __name__ == '__main__':
    print('Before creating the thread')
    your_thread = threading.Thread(target=thread_function, args=('My Thread', ))
    print('Before running thread')
    your_thread.start()

    print('Wait for the thread to finish')
    your_thread.join()
    print('End')
```

```
jon@macbook:~/Desktop$ python test.py
Before creating the thread
Before running thread
Hello from My Thread
Wait for the thread to finish
0
1
2
3
4
End
```

Synchronization: Critical Sections/Mutual Exclusion

- Sequence of instructions that may get incorrect results if executed simultaneously are called **critical sections**
- We also use the term **race condition** to refer a situation in which the results depends on timing
- **Mutual exclusion** means "not simultaneous"
  - A < B or B < A
  - We don't care which
- Forcing mutual exclusions between two critical section executions is sufficient to ensure the correct execution – guarantees ordering
- One way to guarantee mutually exclusive executions is using **locks**

Critical sections



When do critical sections arise?

- One common pattern:
  - read-modify-write of a shared value (variable) in code that can be executed concurrently
- Shared variable:
  - Globals and heap-allocated variables
  - NOT local variables (which are on the stack)

Example: Shared bank account

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
  int balance = get_balance(account);    // read
  balance -= amount;         // modify
  put_balance(account, balance);     // write
  spit out cash;
}
```

- Now suppose you and your partner share a bank account with a balance of $100.00
  - What happened if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?

Example: Shared bank account

- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when that transaction is submitted

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
}
```

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

Execution sequence as seen by CPU

```
balance = get_balance(account);
balance -= amount;
```

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
spit out cash;
```

context switch

```
put_balance(account, balance);
spit out cash;
```

- What's the account's balance after this sequence?
  - Who's happy, the bank or you?

Locks

- A lock is a memory object with two operations:
  - **acquire():** obtain the right to enter the critical section
  - **release():** give up the right to be in the critical section
- **acquire():** prevents progress of the thread until the lock can be acquired