# Student Online Teaching Advice Notice

The materials and content presented within this session are intended solely for use in a context of teaching and learning at Trinity.

Any session recorded for subsequent review is made available solely for the purpose of enhancing student learning.

Students should not edit or modify the recording in any way, nor disseminate it for use outside of a context of teaching and learning at Trinity.

Please be mindful of your physical environment and conscious of what may be captured by the device camera and microphone during videoconferencing calls.

Recorded materials will be handled in compliance with Trinity's statutory duties under the Universities Act, 1997 and in accordance with the University's policies and procedures.

Further information on data protection and best practice when using videoconferencing software is available at https://www.tcd.ie/info_compliance/data-protection/.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

# CSU33031 Computer Networks

## Programming Concepts

Stefan Weber

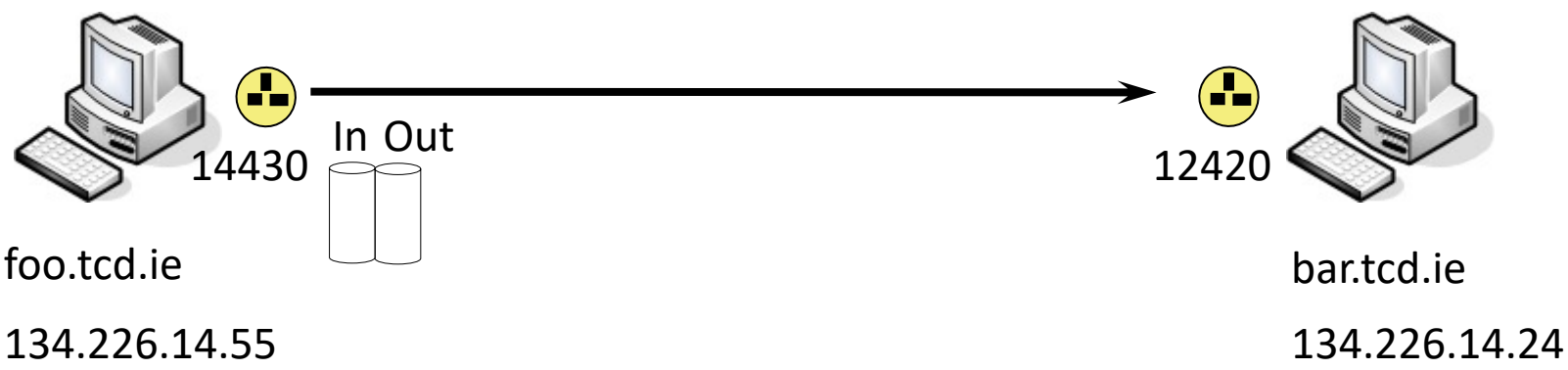
email: sweber@tcd.ie

Office: Lloyd 1.41

# Programming Concepts

- Sockets
- Multi-threading
- Event-based Programming
  - Callbacks

# Sockets & Ports




socket= new DatagramSocket();

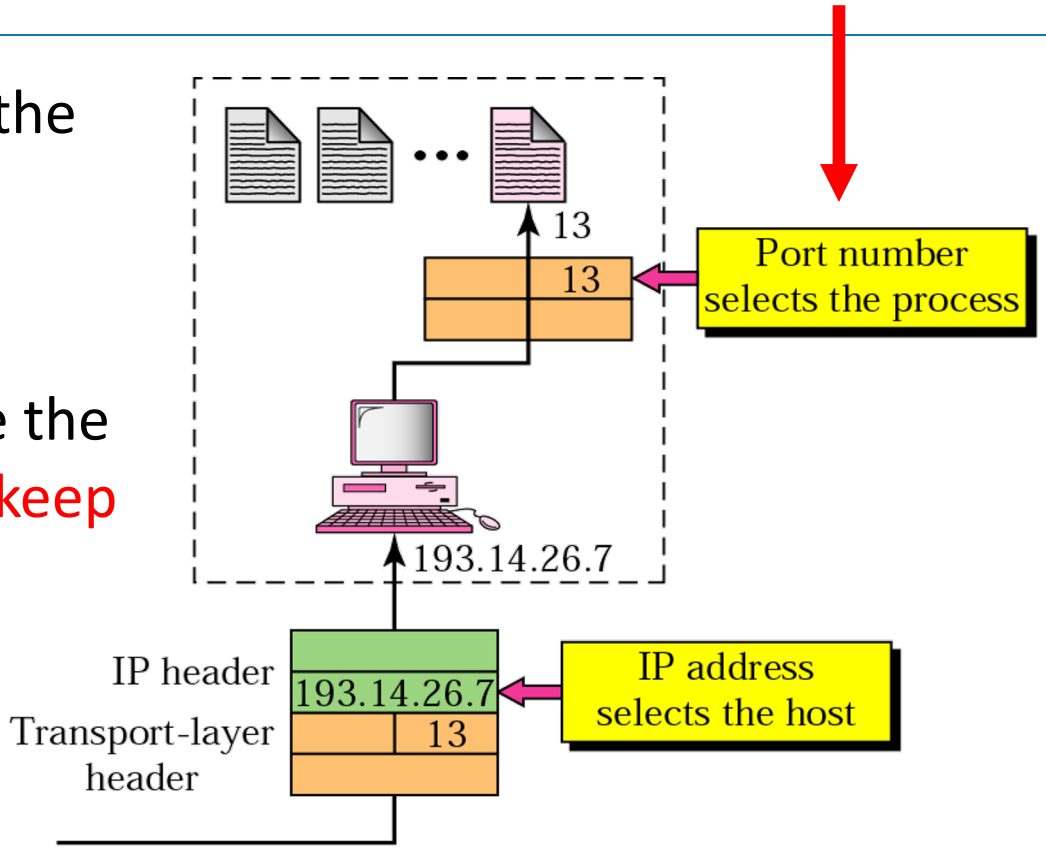
"OS, give me a UDP socket with the next available port number"

```
dstAddress= new InetSocketAddress("bar.tcd.ie", 12420);
packet= new DatagramPacket(data, data.length, dstAddress);
socket.send(packet);
```

# IP Addresses & Port Numbers

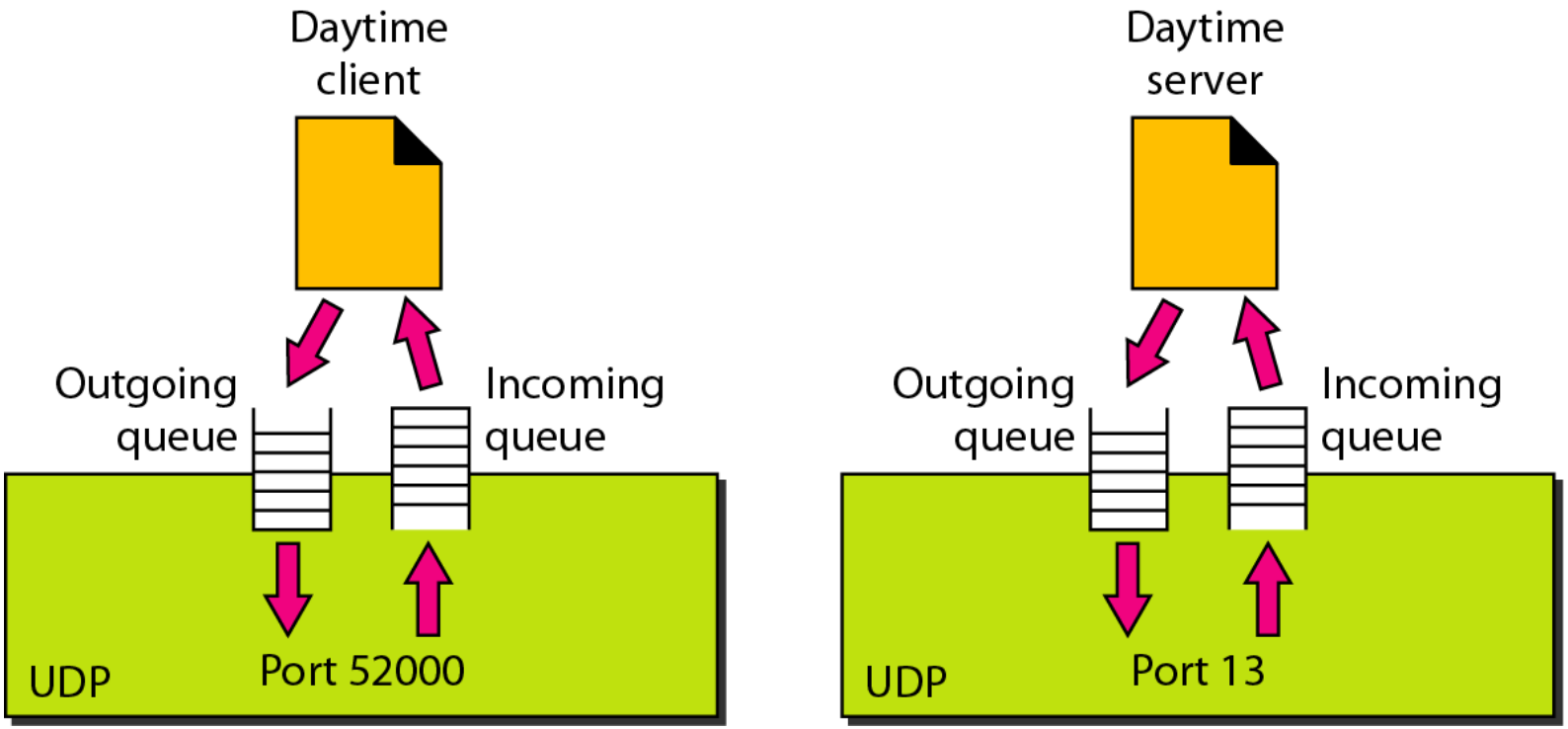
No it does not!!!

- IP Addresses determine the host
- Port Numbers determine the storage where UDP/TCP keep datagrams/data



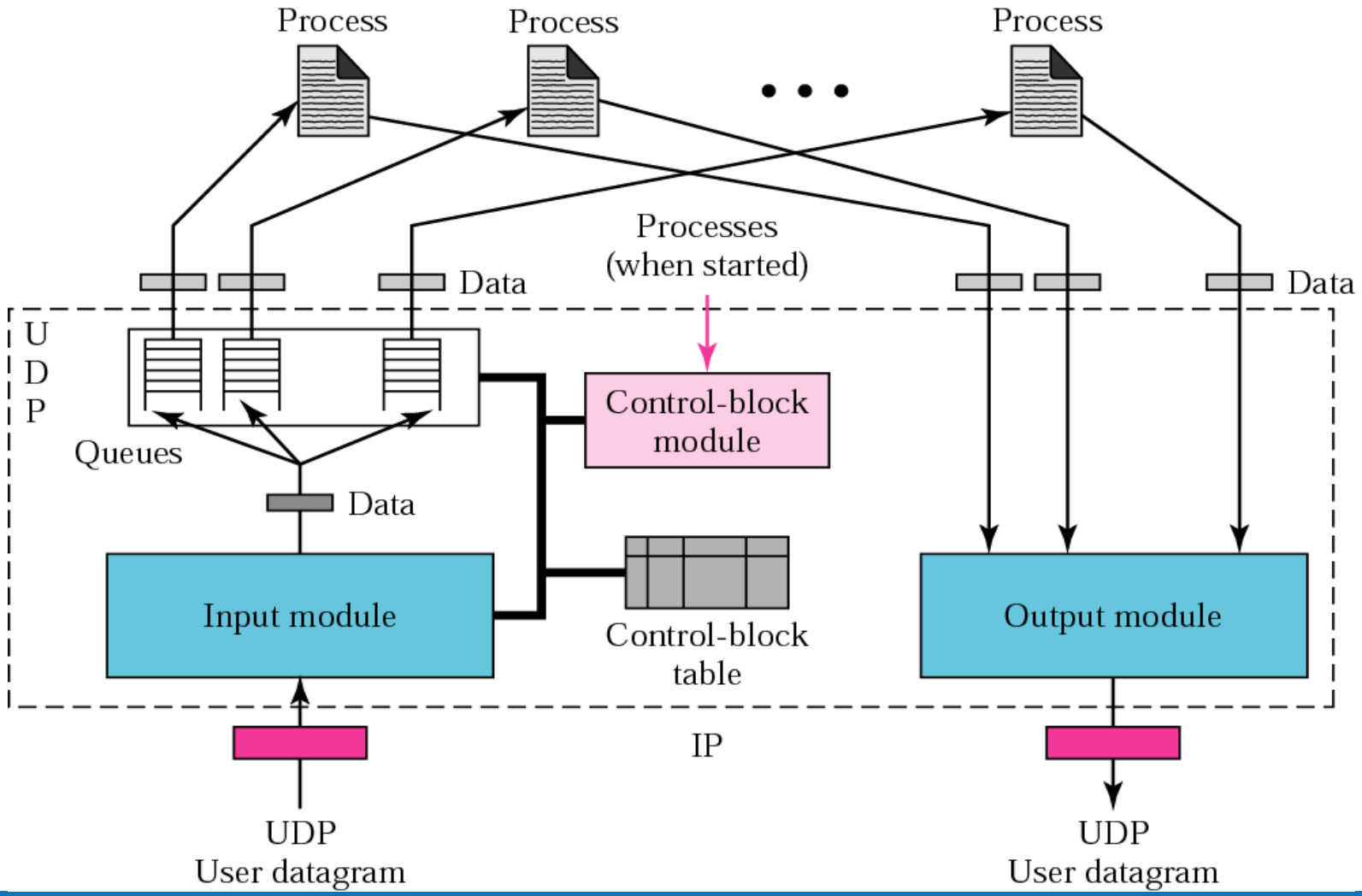

* Figure is courtesy of B. Forouzan

# Queuing in UDP



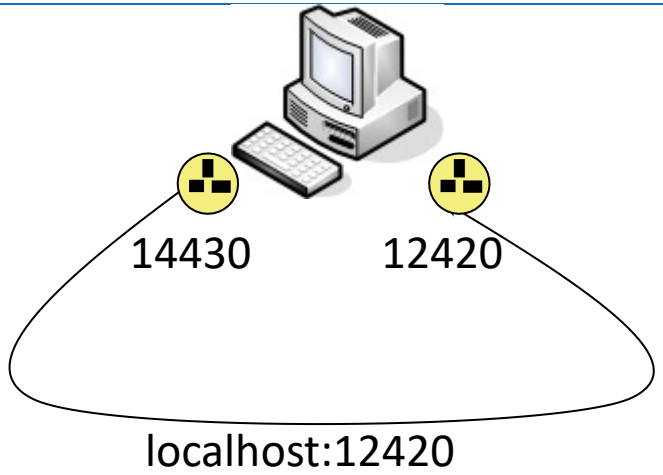

* Figure is courtesy of B. Forouzan

# Processes and UDP Queues



# Sockets & Ports

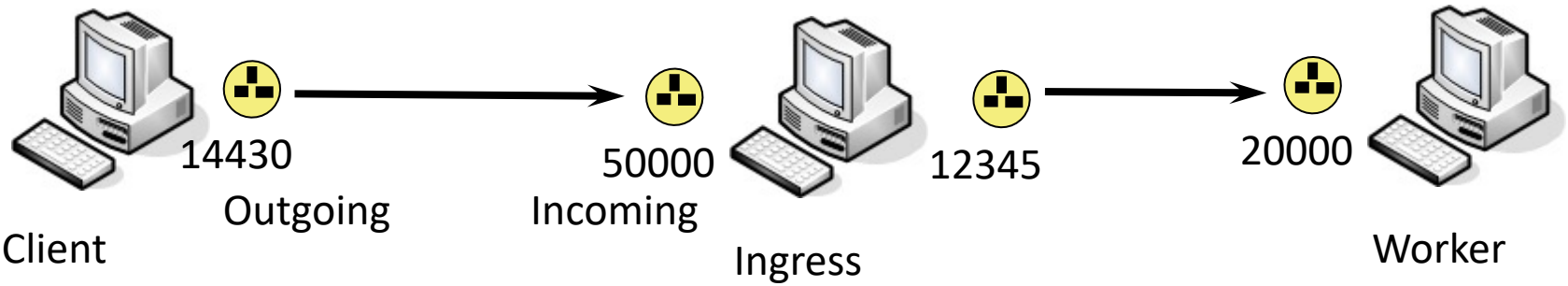
(localhost = 127.0.0.1 – lo interface)




```
socket= new DatagramSocket(14430);

dstAddress= new InetSocketAddress("localhost", 12420);
packet= new DatagramPacket(data, data.length, dstAddress);
socket.send(packet);
```

# Sockets & Ports

(Assignment setup on a local machine)




Setup on localhost:

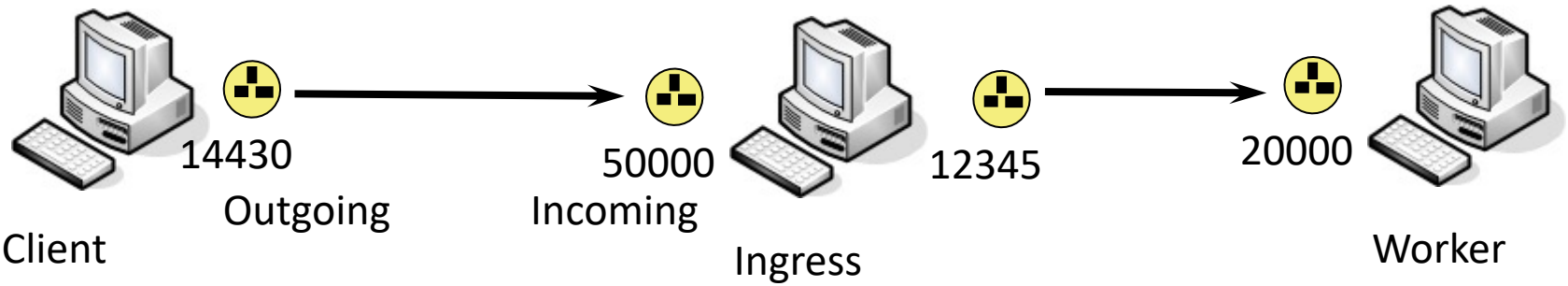
127.0.0.1

127.0.0.1

127.0.0.1

# Sockets & Ports

(Using docker and multiple networks – see walkthrough)

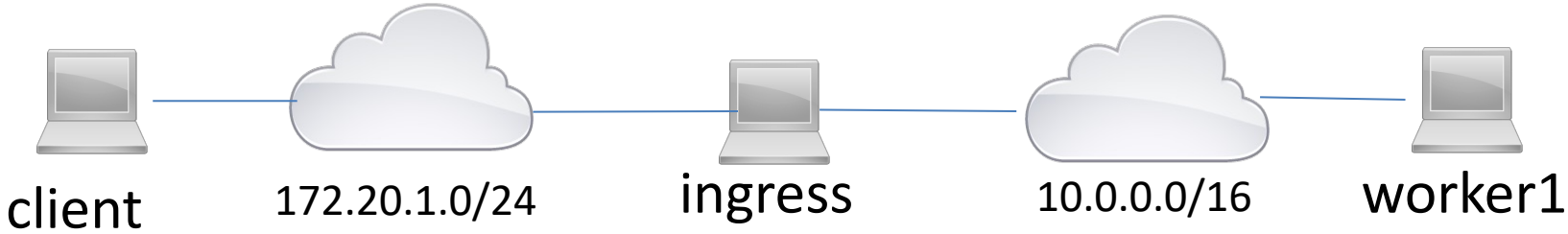



Setup on localhost:

127.0.0.1 127.0.0.1 127.0.0.1

Setup on docker:

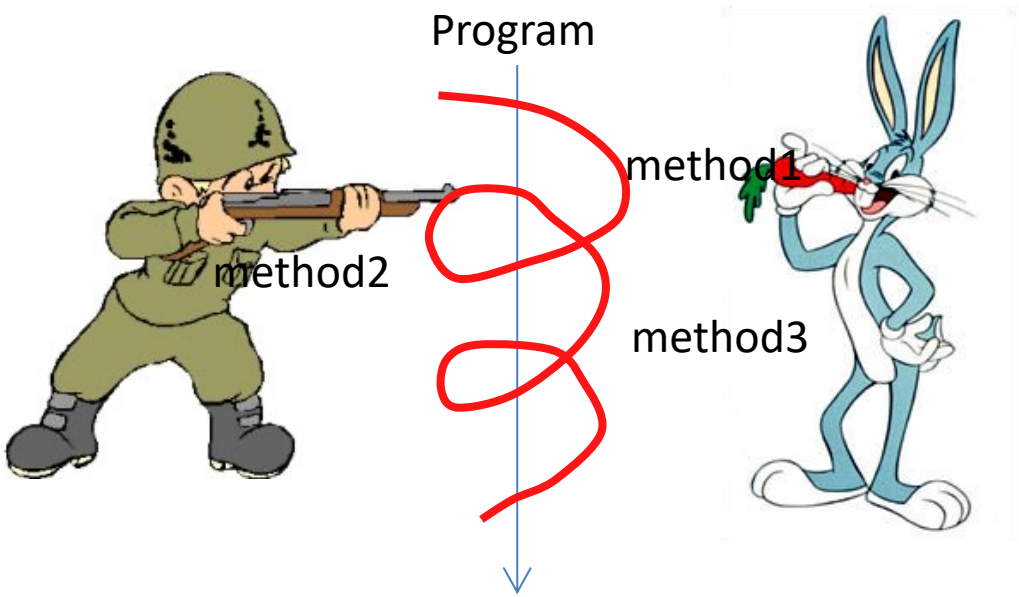
eth1 172.20.1.2 | eth1 172.20.1.3 | eth2 10.0.0.2 | eth1 10.0.0.3



# Threads

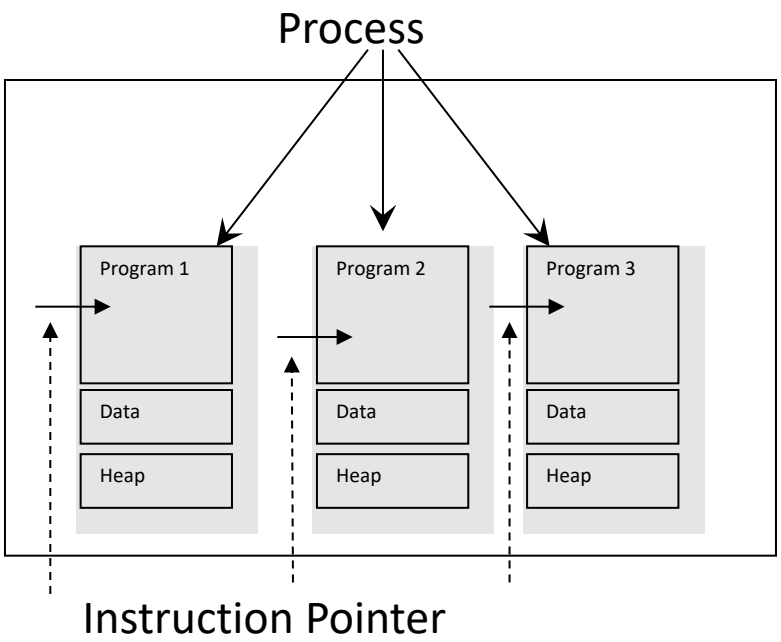
(How to handle traffic w/ sockets)

- Threads of Execution
  - Lightweight Processes



# Processes

- Separate address spaces
- Registers per process




- Problem:
  - Switching between processes

Processor

Register 1 Register 2 Register 3 Register 4 Register 5 Register 6 Register 7 Register 8 Register 9 Register 10 Register 11 Register 12 Register 13 Register 14 Register 15 Register 16 Register 17 Register 18 Register 19 Register 20 Register 21 Register 22 Register 23 Register 24

# Per-Process Details

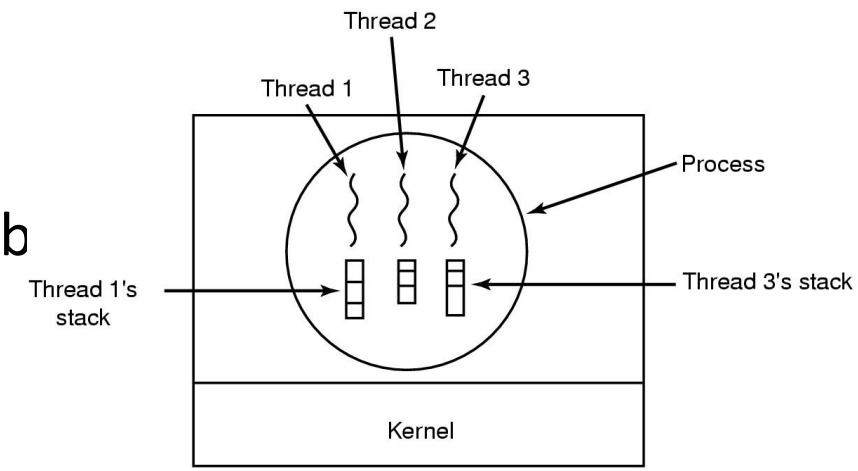
(That need to be saved at context switch)

| Process management | Memory management | File management |
|---|---|---|
| Registers | Pointer to text segment | UMASK mask |
| Program counter | Pointer to data segment | Root directory |
| Program status word | Pointer to bss segment | Working directory |
| Stack pointer | Exit status | File descriptors |
| Process state | Signal status | Effective uid |
| Time when process started | Process id | Effective gid |
| CPU time used | Parent process | System call parameters |
| Children's CPU time | Process group | Various flag bits |
| Time of next alarm | Real uid | |
| Message queue pointers | Effective uid | |
| Pending signal bits | Real gid | |
| Process id | Effective gid | |
| Various flag bits | Bit maps for signals | |
| | Various flag bits | |

* Figure is courtesy of A. Tanenbaum

# Threads

- Lightweight processes
- Share same address space
- Less overhead for switching b
  between processes

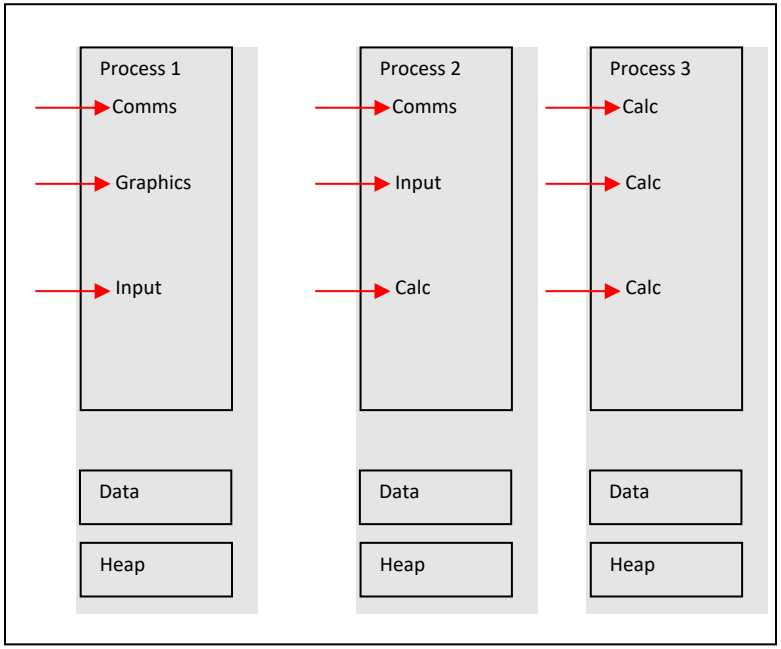



| **Per process items** | **Per thread items** |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |

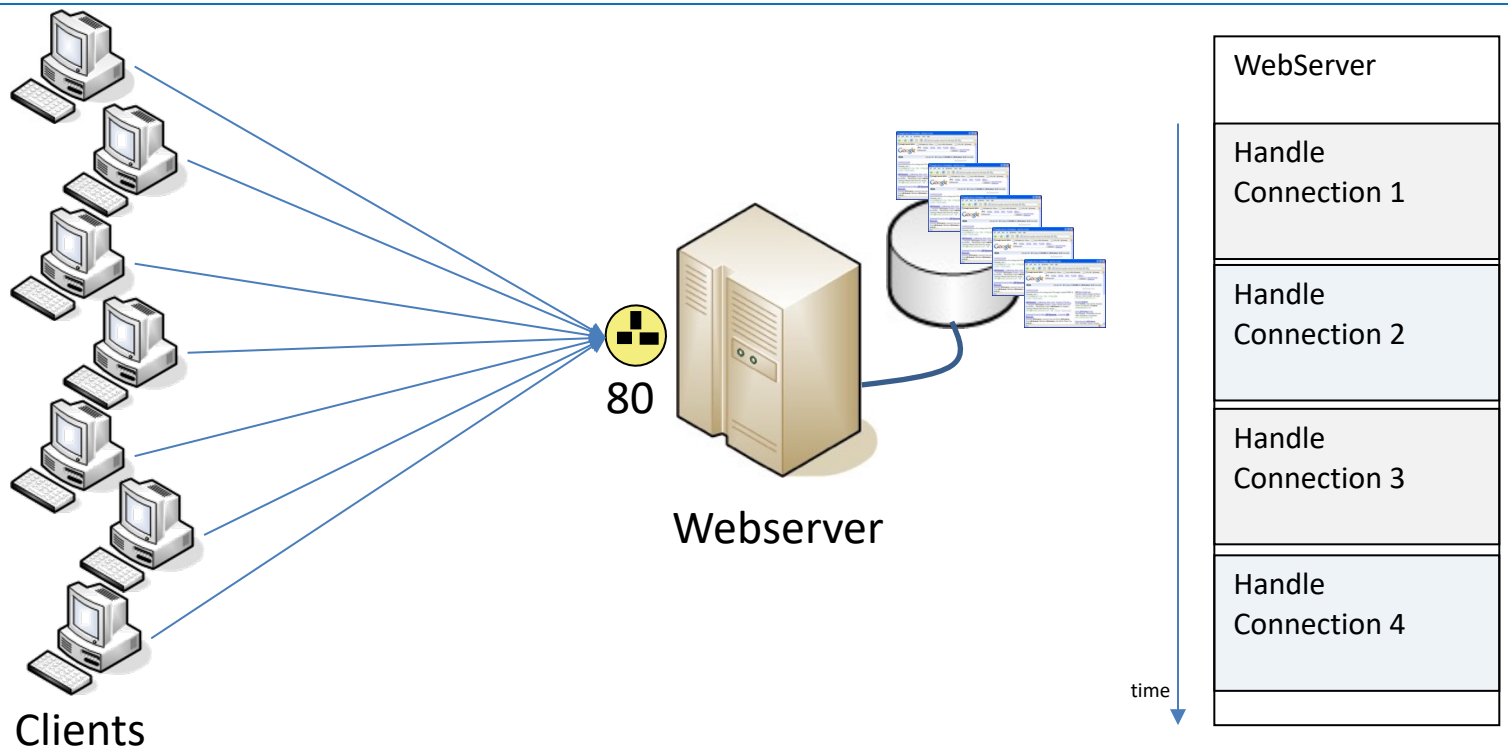
* Figure is courtesy of A. Tanenbaum

# Multi-Threaded

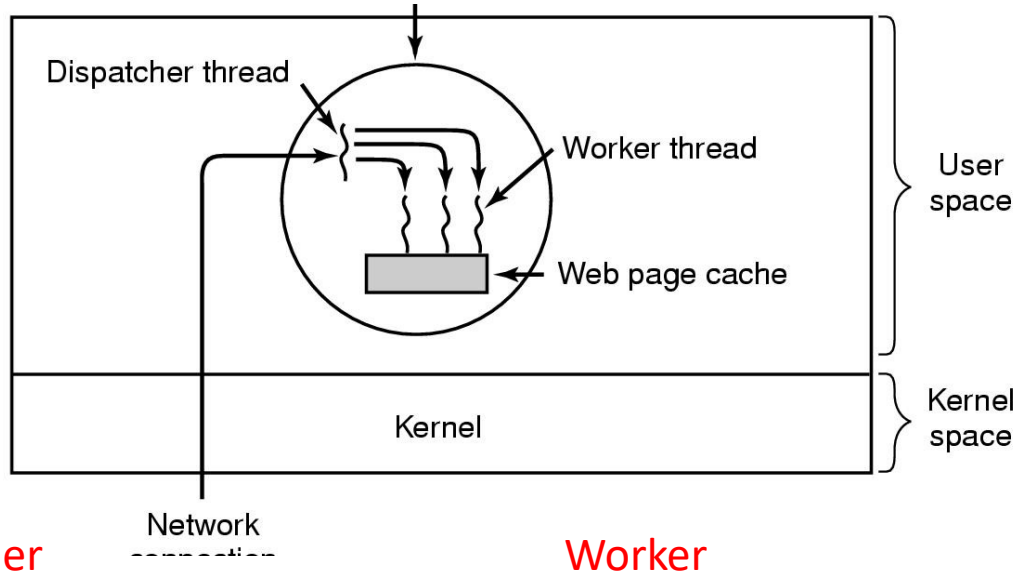
**Per thread items**
Program counter
Registers
Stack
State



# Threads & Webserver



# Application of Threads




Dispatcher

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

Worker

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

* Figure is courtesy of A. Tanenbaum

# Java Threads

```
class Thread {
    public Thread (String name);
    public Thread (Runnable target)
    …
    public void start ();
    static void sleep (long millis)
}
```

Selection of methods of class “Thread”

# Inheriting from Threads + Overload run()

```
class Thread {
    public Thread (String name);
    …
    public void start ();
    public void run();
}
```

Selection of methods of class "Thread"

```
class XYZ extends Thread {
    public void run() {
    }
}
```

Class that extends "Thread" needs to implement the **`run`** method

# Java Thread – Socket Example I

```
class SocketThread extends Thread {
    DatagramSocket socket;

    SocketThread (String name, int port) {
        super (name);
        socket= new DatagramSocket(port);
    }
}
```

```
t1 = new SocketThread (“Socket1”, 50000);
```

# Java Thread – Socket  Example II

```
class SocketThread extends Thread {

    DatagramSocket socket;

    SocketThread (String name, int port) {
        super (name);
        socket= new DatagramSocket(port);
    }

    public void run() {
        while(TRUE) {
            packet= socket.receive();
            System.out.println (name + ": " + packet.getData());
        }
    }
}
```

# Creating & Starting Threads I

```
SocketThread t1, t2, t3;

t1 = new SocketThread (“Socket1”, 50000);
t2 = new SocketThread (“Socket2”, 50200);
t3 = new SocketThread (“Socket3”, 55000);
```

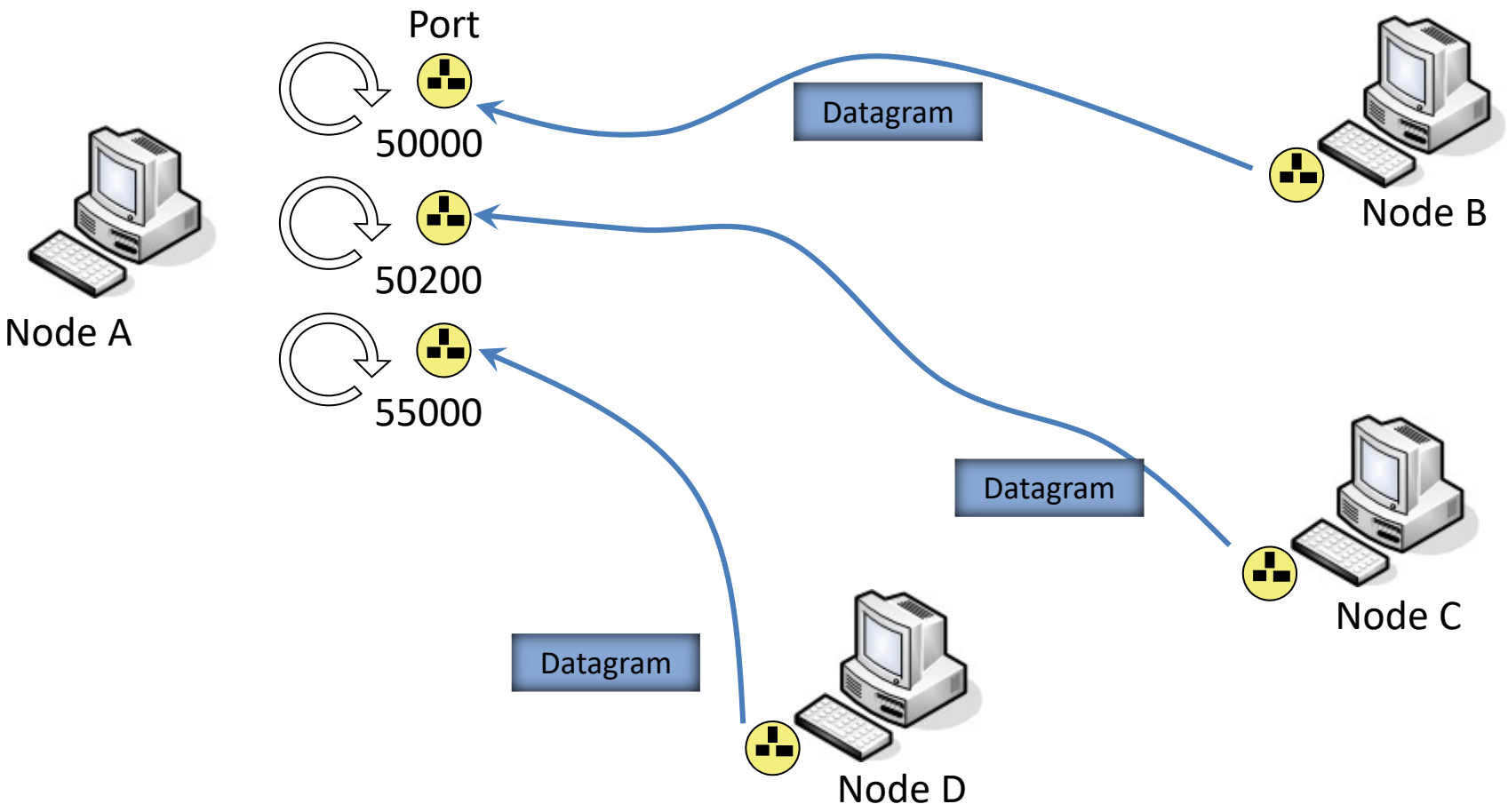
# Creating & Starting Threads II

```
SocketThread t1, t2, t3;

t1 = new SocketThread ("Socket1", 50000);
t2 = new SocketThread ("Socket2", 50200);
t3 = new SocketThread ("Socket3", 55000);

t1.start();
t2.start();
t3.start();
```

Insert thread into list of running threads and execute "run" method

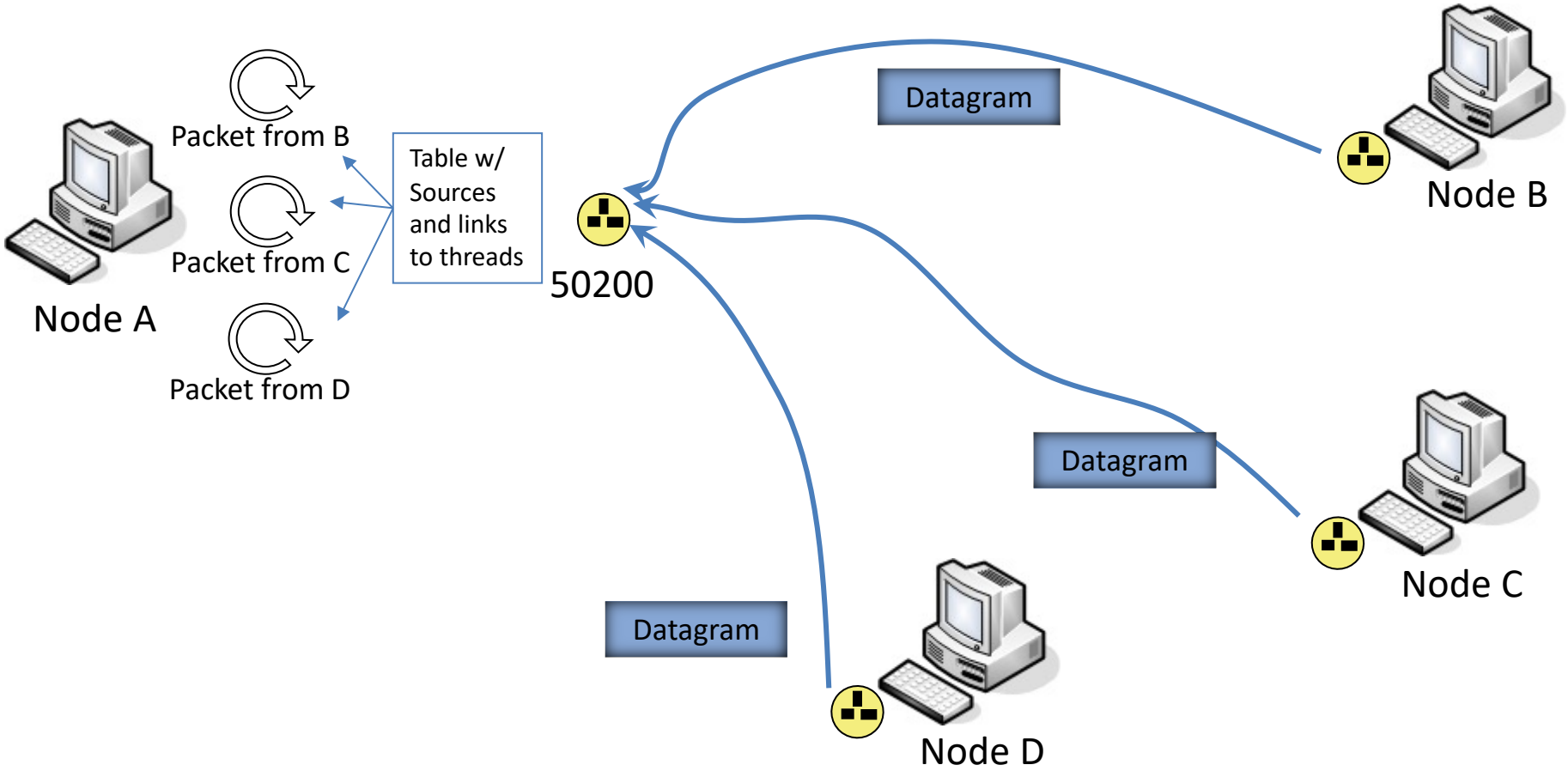
# Concurrent Communication

(Example is wasteful on number of ports used!!!)



# Concurrent Communication

(Multiplexing over one socket is more efficient & common)



# Thread Execution Example I

```
class CounterThread extends Thread {
    long counter;

    CounterThread (String name, long counter) {
        super (name);
        this.counter = counter;
    }
}
```

```
t1 = new CounterThread (“T1”, 10);
```

# Thread Execution Example II

```
class CounterThread extends Thread {
    long counter;

    CounterThread (String name, long counter) {
        super (name);
        this.counter = counter;
    }
    public void run() {
        while(TRUE) {
            counter++;
            System.out.println (name + ": " + counter);
            Thread.sleep (Math.random() * 5000);
        }
```

# Thread Execution Example III

```
CounterThread t1, t2, t3;

t1 = new CounterThread ("T1", 10);
t2 = new CounterThread ("T2", 10);
t3 = new CounterThread ("T3", 10);

t1.start();
t2.start();
t3.start();
```

Insert thread into list of running threads and execute "run" method

# Possible Output

time

T1: 11
T2: 11
T3: 11
T1: 12
T2: 12
T3: 12
...

or

time

T1: 11
T1: 12
T3: 11
T3: 12
T2: 11
T3: 13
...

or

time

T1: 11
T3: 11
T3: 12
T2: 11
T1: 12
T3: 13
...

Execution is **non-deterministic**!

# Interface: java.lang.Runnable

**Java doesn't support Multiple Inheritance:**

```
class AccountThread extends Thread, Account {...
```

**ERROR**

**Java doesn't support multiple inheritance**

# Interface: java.lang.Runnable

**Java doesn't support Multiple Inheritance:**

```
class AccountThread extends Thread, Account {...
```

**ERROR**

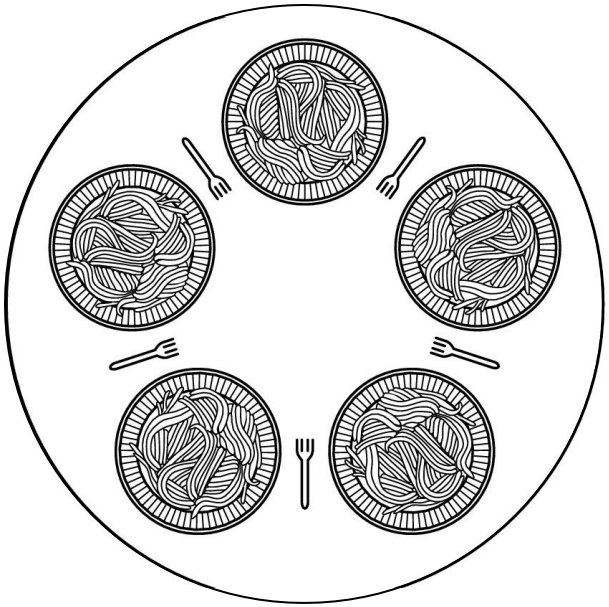
**Java doesn't support multiple inheritance**

```
class CounterThread implements Runnable {
        ...
    public void run() {
}

new Thread (new CounterThread("T1", 10)).start;
```

# Problems with Concurrency

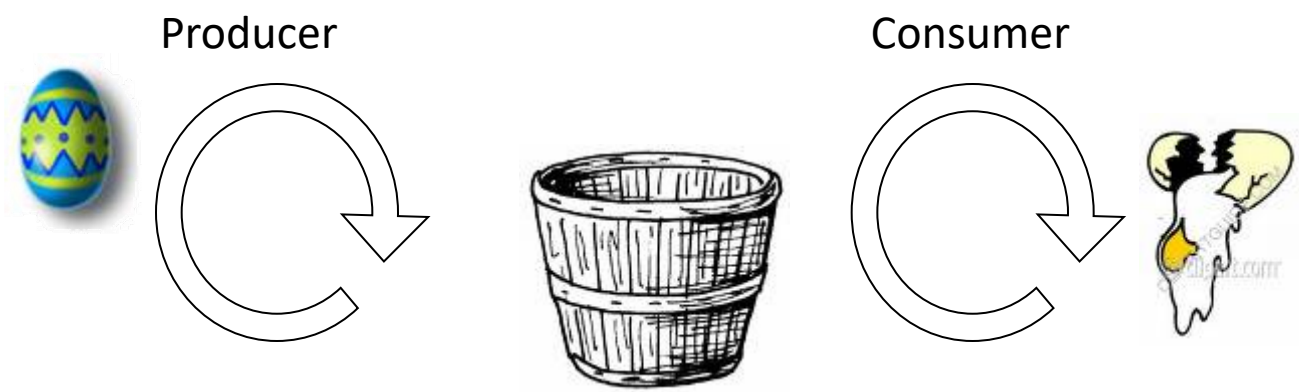
- Concurrent access to global variables, etc
- Requires synchronization
- Approaches
  - Monitors
  - Semaphores
  - Barriers

(see Principles of Concurrent Programming, M. Ben-Ari)



Dining Philosophers

# Producer-Consumer Problem




- Producer delivers 1 egg at a time
- Basket can hold exactly 1 egg
- Consumer can only consume an egg if an egg is in the basket

# Producer-Consumer in Java I

```
class TestSystem {
	Basket basket;

	TestSystem() {
		basket= new Basket(0);
	}

	class Basket {
		int content;

		public Basket (int content) {
			this.content= content;
		}
	}
}
```

# Producer-Consumer in Java II

```
class TestSystem {
    …
    class Basket {
        int content;
        …

        public void putEgg () {
            content++;
        }

        public void takeEgg() {
            content--;
        }
    }
}
```

# Producer-Consumer in Java III

```
class TestSystem {
    Basket basket;

    class Producer extends Thread {
        public void run() {
            while (true)   basket.putEgg();
        }
    }

    class Consumer extends Thread {
        public void run() {
            while (true) basket.takeEgg();
        }
    }
}
```

# Producer-Consumer in Java IV

```
class TestSystem {

    public static void main (String[] args) {
        Producer producer;
        Consumer consumer;

        producer= new Producer();
        consumer= new Consumer();

        producer.start();
        consumer.start();
    }
}
```

# Problem???

Producer

Execution is non-deterministic!

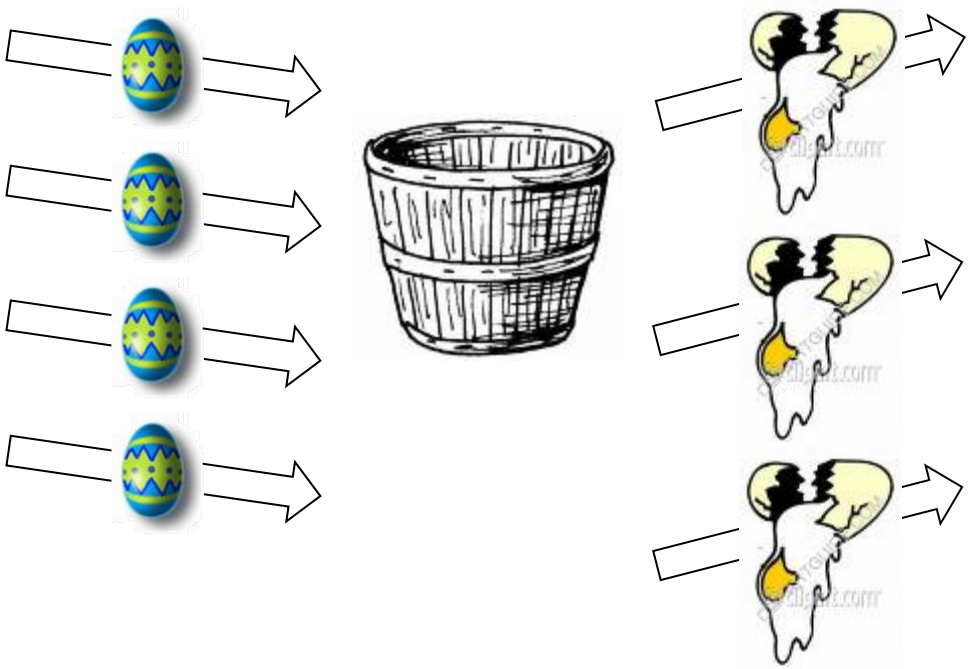
Consumer

putEgg

putEgg

putEgg

putEgg



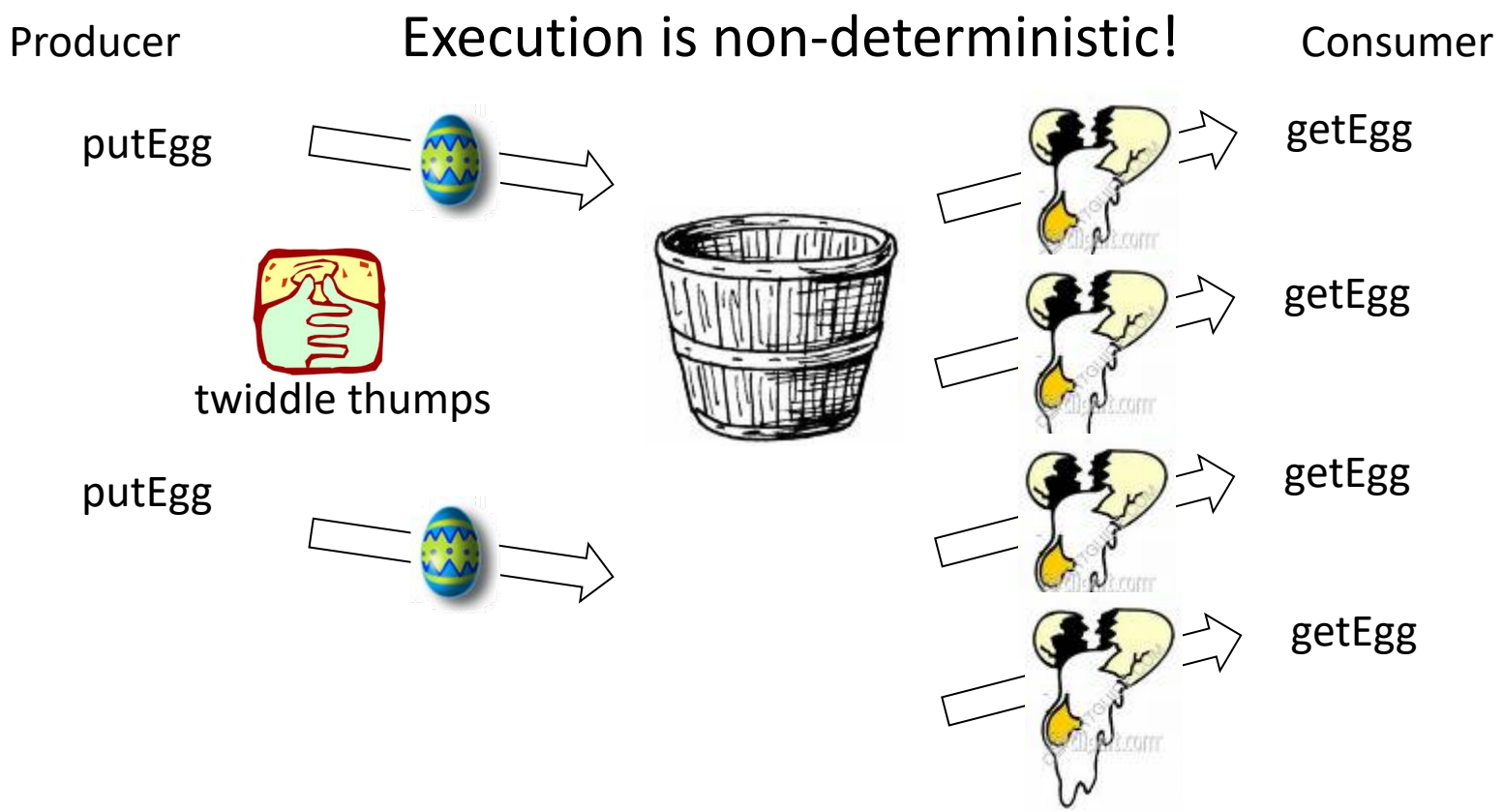
getEgg

getEgg

getEgg

# Problem???

(What if getEgg executes before or more than putEgg???)



# Producer-Consumer in Java V

(synchronized = only one thread can execute method at a time)

```
class TestSystem {
    …
    class Basket {
        int content;
        …
        public synchronized void putEgg () {
            while (content!=0) wait();
            content++;
            notify();
        }
    }
}
```

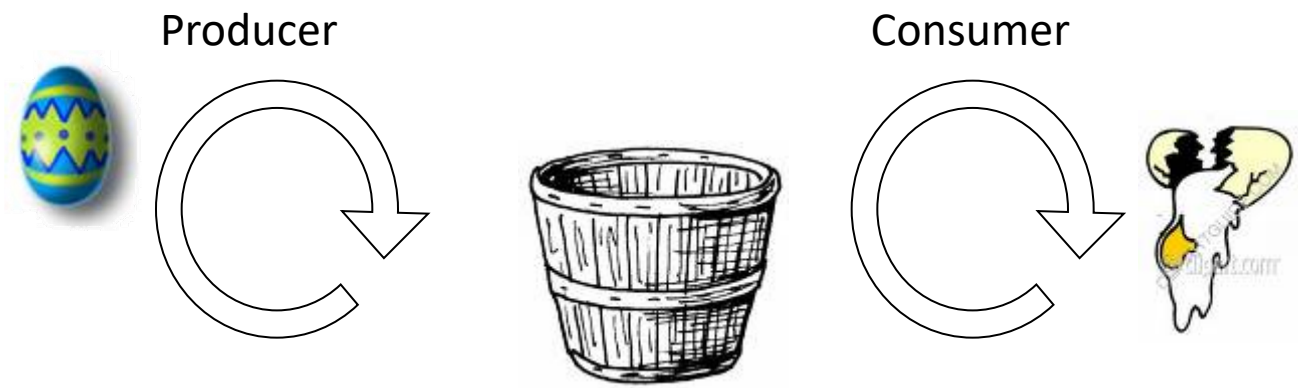
# Producer-Consumer in Java VI

(synchronized = only one thread can execute method at a time)

```
class TestSystem {
    …
    class Basket {
        int content;
        …
        public synchronized void takeEgg () {
            while (content!=1) wait();
            content--;
            notify();
        }
    }
}
```

# Producer-Consumer Problem

(notify = wakes up waiting threads, one will succeed to execute)

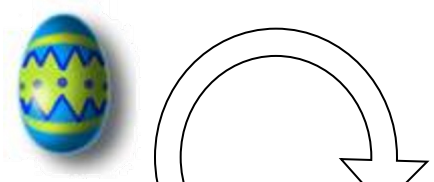



```
public synchronized void putEgg () {
        while (content!=0) wait();
        content++;
        notify();
}
```

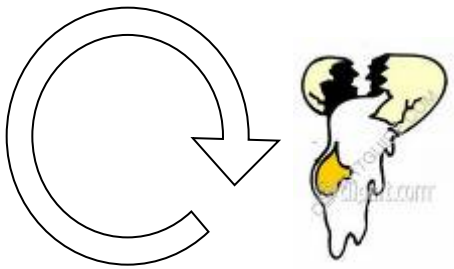
```
public synchronized void takeEgg () {
        while (content!=1) wait();
        content--;
        notify();
}
```

# Producer-Consumer Problem

Producer



Consumer



```
public synchronized void putEgg () {
        while (content!=0) wait();
        content++;
        notify();
}
```

```
public synchronized void takeEgg () {
        while (content!=1) wait();
        content--;
        notify();
}
```

Monitor in Java: One active thread in method per instance!

# Summary: Threads

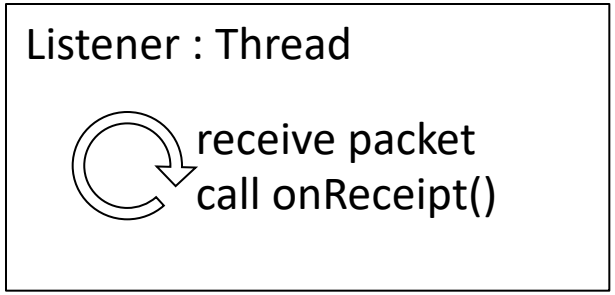
- Concurrent Execution
  - Non-deterministic Execution
- Java
  - Inherit from Thread class
  - Implement Runnable interface
- Synchronization
  - wait() & notifyAll() / notify()

# “Event-based Programming”

```
public void run() {
    DatagramPacket packet;


    try {
        while(true) {
            packet = new DatagramPacket(new byte[PACKETSIZE], PACKETSIZE);
                                socket.receive(packet);
                                onReceipt(packet);
        }
    } catch (Exception e)  {e.printStackTrace();}
}
```

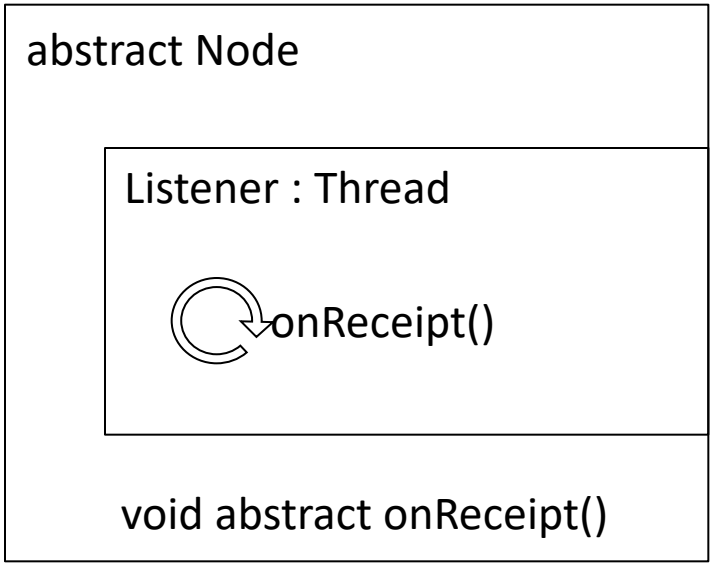
# “Event-based Programming”

(Callback -> onReceipt method – event happens, call onReceipt)

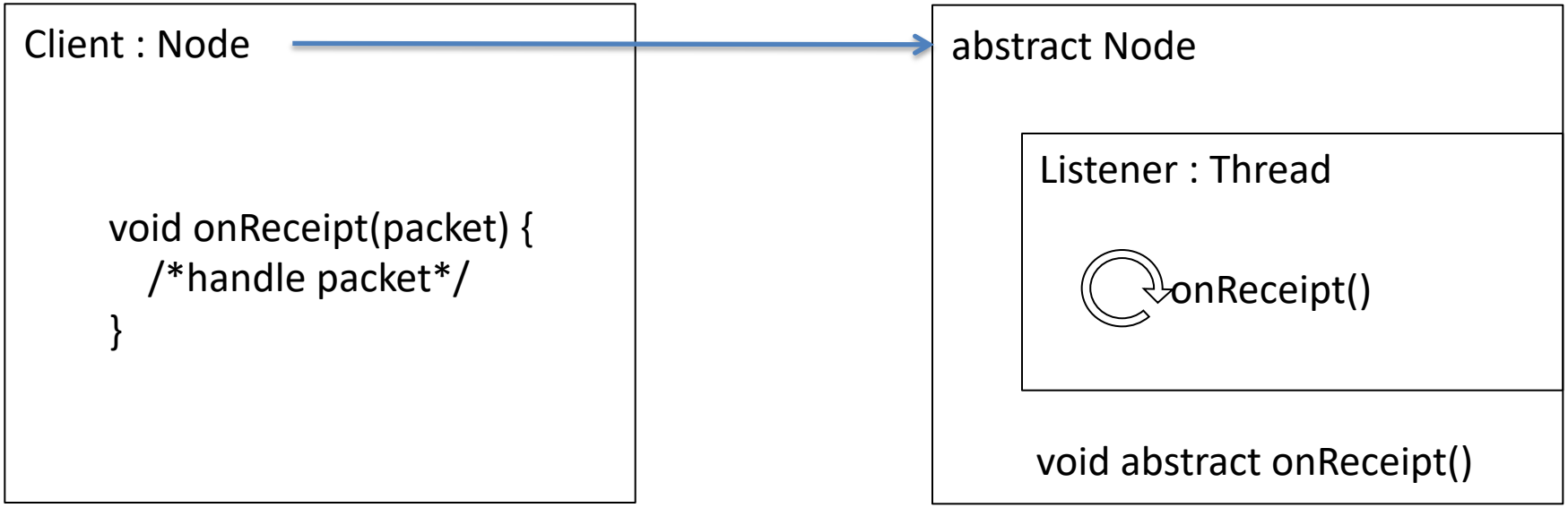


# “Event-based Programming”

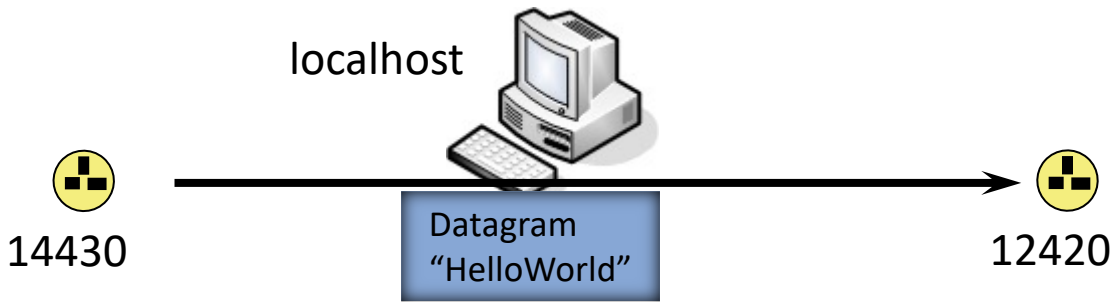
(Abstract class that forces all Node classes to have an onReceipt method)



# “Event-based Programming”



# “Event-based Programming”




Listener

Server

```
void onReceipt(packet) {
}
```

# “Event-based Programming”

(latch as quick-fix to only start receiving packets when receiver is ready)

```
public void go() {latch.countDown();}
public void run() {
    DatagramPacket packet;

    try {
        latch.await();
        while(true) {
            packet = new DatagramPacket(new byte[PACKETSIZE], PACKETSIZE);
                                socket.receive(packet);
                                onReceipt(packet);
    }
    } catch (Exception e)
            {if (!(e instanceof SocketException)) e.printStackTrace();}
}
```

# Summary: Socket Communication

- Sockets
  - Localhost
- Threading
  - Process vs Threads
  - Synchronization
  - Monitors in Java (synchronized keyword)
- Event-based Programming

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin