Review: Resource Metric

Camera

Sends images

Image Service

Processes images
Review: Resource Metric

• Say, capacity is measured in terms of processor cycles
Review: Resource Metric

- Say, capacity is measured in terms of processor cycles
- Workload might be processing a single image
Review: Resource Metric

- Say, capacity is measured in terms of processor cycles
- Workload might be processing a single image
- Utilization could 10% -> 90% of processor is unused
Review: Resource Metric

- Say, capacity is measured in terms of processor cycles
- Workload might be processing a single image
- Utilization could 10% -> 90% of processor is unused
- Overhead could go to the OS, perhaps 5% of the CPU going to OS bookkeeping
Review: Resource Metric

- Say, capacity is measured in terms of processor cycles
- Workload might be processing a single image
- Utilization could 10% -> 90% of processor is unused
- Overhead could go to the OS, perhaps 5% of the CPU going to OS bookkeeping
- Useful work would then be 5%
Review: Latency

Camera

Sends images

Image Service
Phase 1
Phase 2

Processes images
Review: Latency

- In client/server model, latency is simply: time between client sending request and receiving response
Review: Latency

- In client/server model, latency is simply: time between client sending request and receiving response
- What contributes to latency?
Review: Latency

• In client/server model, latency is simply: time between client sending request and receiving response

• What contributes to latency?
  • Latency sending the message
Review: Latency

• In client/server model, latency is simply: time between client sending request and receiving response

• What contributes to latency?
  • Latency sending the message
  • Latency processing the message

Camera

Sends images

Image Service

Phase 1

Phase 2

Processes images
Review: Latency

- In client/server model, latency is simply: time between client sending request and receiving response.

- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
Review: Latency

- In client/server model, latency is simply: time between client sending request and receiving response
- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
- Adding pipelined components -> latency is cumulative

---

Diagram:
- Camera sends images to Image Service
- Image Service processes images

Phase 1 and Phase 2
Review: Latency

- In client/server model, latency is simply: time between client sending request and receiving response.
- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
- Adding pipelined components -> latency is cumulative.
Review: Latency

- In client/server model, latency is simply: time between client sending request and receiving response
- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
- Adding pipelined components -> latency is cumulative

![Diagram showing latency components]

Camera → Sends images 10ns → Image Service

Phase 1: 10ns

Phase 2: 5ns

Total latency: 30ns
Review: Throughput

- Measure of the rate of useful work done for a given workload
- Example:
  - Throughput is camera frames processed/second
- When adding multiple pipelined components -> throughput is the minimum value

![Diagram]

Camera -> 1000 fps Sends images -> Image Service

Phase 1: 10fps
Phase 2: 29fps

Total throughput: 10fps
Review: Designing for Performance

- Measure system to find which aspect of performance is lacking (throughput or latency)
- Measure each component to identify bottleneck
- Identify if fixing that bottleneck will realistically improve system performance
- Measure improvement
- Repeat
Review: Streams

• Java 8 introduced the concept of **Streams**
• A stream is a sequence of objects
• Streams have functions that you can perform on them, which are (mostly) **non-interfering** and **stateless**
  • Non-interfering: Does not modify the actual stream
  • Stateless: Each time the function is called on the same data, get same result
• Example:
  
  ```java
  IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
  .filter(x -> isPrime(x)) //Retain only values that pass some expensive isPrime function
  .forEach(System.out::println); //For each value returned by filter, print it
  ```
Review: Streams
Review: Streams

```java
IntStream.range(1, 1000000)
    .filter(x -> isPrime(x))
    .forEach(System.out::println);
```

//For each value returned by filter, print it
Review: Streams

• Why use the stream interface instead of

```java
System.out.println(x);
```
Review: Streams

- Why use the stream interface instead of

- Who wants to write the parallel version of this?

```java
IntStream.range(1, 1000000).filter(x -> isPrime(x)).parallel().forEach(System.out::println); // For each value returned by filter, print it
```
Review: Streams

• Why use the stream interface instead of

• Who wants to write the parallel version of this?

• The magic works as long as isPrime is stateless!
Announcements

• Reminder: HW2 is out
• Today: Networking Basics
• Reading: Saltzer Ch 7.1, 7.2
Streams under the hood

- Just adding more parallel() doesn't always make it faster! (see: law of leaky abstractions)
- There is some overhead to how a parallel operation occurs
- Internally, Java keeps a pool of worker threads (rather than make new threads for each parallel task)
- Streams use a special kind of pool, called a ForkJoinPool
Fork/Join Programming

• Special kind of task - `fork()` defines how to create subtasks, `join()` defines how to combine the results
• Similar to map/reduce, but not distributed
• For streams:
  • **Fork** a task into subtasks for many threads to work on
  • **Join** the results together
Fork/Join Programming

• Obligatory array sum example

```java
class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 5000;

    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }

    protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }

    static long sumArray(int[] array) {
        return ForkJoinPool.commonPool().invoke(new Sum(array, 0, array.length));
    }
}
```
(Fork/Join Demo)
Promise

• What if we want to run some task, and do stuff while we are waiting for it to be done?
• You COULD do it with a complicated combination of `synchronized`, `wait`, and `notify`
• You can use the **Promise** abstraction instead
  • Called a **CompletableFuture** in Java 8

```java
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Result of the asynchronous computation";
});
```

```java
// Block and get the result of the Future
String result = future.get();
System.out.println(result);
```
Promise Use-Cases

• Any case where you need to have multiple things happen in the background, but care about the result, and care about them happening in some order

• Asynchronous I/O
  • Read data from a web service
  • Then process it
  • Then save it to a file
Chaining Promises

Promise to get some data
Chaining Promises

Promise to get some data

then

Promise to make some changes to that data
Chaining Promises

1. Promise to get some data
2. then
3. Promise to make some changes to that data
4. then
5. Report on those changes to the user
Chaining Promises

- Promise to get some data
- Promise to make some changes to that data
- Report on those changes to the user

If there's an error...

Report on the error
Chaining Promises

1. Promise to get some data
2. Promise to make some changes to that data
3. Report on those changes to the user
   
   then

4. Promise to make some other changes to that data

5. Report on the error
   
   then
   
   If there's an error...

6. Report on those changes to the user
   
   then
   
   If there's an error...
Chaining Promises

Promise to get some data

Promise to make some other changes to that data

Promise to make some changes to that data

Report on those changes to the user

Report on the error

If there's an error…

then

then

then

thenCombine

If there's an error…
Promises

• Catch errors by providing a callback function for **exceptionally** (called when an exception occurs in any of those threads)

• API: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html
(Promises Demo)
Networks as Abstractions

• A network consists of communication links
• Networks have several “interesting” properties we will look at
  • Latency
  • Failure modes
• What is the abstraction?
Networks as Abstractions
Networks as Abstractions

- Stuff goes in, stuff goes out?
Networks as Abstractions

- Stuff goes in, stuff goes out?
- Not a perfect abstraction, because:
  - Speed of light (1 foot/nanosecond)
  - Communication links exist in uncontrolled/hostile environments
  - Communication links may be bandwidth limited (tough to reach even 100MB/sec)
Networks as Abstractions

- Stuff goes in, stuff goes out?
- Not a perfect abstraction, because:
  - Speed of light (1 foot/nanosecond)
  - Communication links exist in uncontrolled/hostile environments
  - Communication links may be bandwidth limited (tough to reach even 100MB/sec)
- In contrast to a single computer, where:
  - Distances are measured in mm, not feet
  - Physical concerns can be addressed all at once
  - Bandwidth is plentiful (easily GB/sec)
Networks are Shared

• With processes, we considered how one CPU could be shared between multiple programs running at once
• With networks, communication links are probably shared even more widely
Networks are Shared

• With processes, we considered how one CPU could be shared between multiple programs running at once
• With networks, communication links are probably shared even more widely
Networks are Shared

• With processes, we considered how one CPU could be shared between multiple programs running at once
• With networks, communication links are probably shared even more widely

Everyone talks to everyone on their own link
Not scalable
Networks are Shared

• With processes, we considered how one CPU could be shared between multiple programs running at once

• With networks, communication links are probably shared even more widely

Everyone talks to everyone on their own link
Not scalable
Network as Abstractions

• What do we send, what gets received?
• At the lowest level, we call what gets sent frames
• Each frame is limited in size
  • Ethernet: max 1522 bytes
• Frame is packed with source/destination info into a packet
• Network knows what to do with packets to get them to their destination
Networks as Abstractions

LAN1

Host A

client

protocol software

LAN1 adapter

Router

Host B

server

protocol software

LAN2 adapter

LAN2

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

LAN1

Host A
client

protocol
software

LAN1
adapter

(1) data

(3) frame packet

Router

LAN2

Host B
server

protocol
software

LAN2
adapter

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

LAN1

(1) data

(3) frame packet

Host A

client

protocol software

LAN1 adapter

(4) frame packet

LAN2

Host B

server

protocol software

LAN2 adapter

Router

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

1. **Host A**
   - Client
   - Protocol software
   - LAN1 adapter

2. **Host B**
   - Server
   - Protocol software
   - LAN2 adapter

**LAN1**

1. Data
2. Frame
3. Frame
4. Frame
5. Frame

**LAN2**

**Router**

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

LAN1

1. data
2. protocol software
3. frame packet
4. frame packet

LAN2

1. data
2. protocol software
3. frame packet
4. frame packet

Host A

client

Host B

server

Router

PH: Internet packet header
FH: LAN frame header

GMU CS 475 Spring 2018
Networks as Abstractions

LAN1

(1) data
(3) frame packet
(4) frame packet

Host A
client
protocol software
LAN1 adapter

Router

LAN2

Host B
server
protocol software
LAN2 adapter

(5) frame packet
(6) frame packet
(7) frame packet

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

LAN1

(1) data

(3) frame packet packet

(3) frame packet

(4) frame packet

Host A

client

protocol software

LAN1 adapter

LAN2

Host B

server

protocol software

LAN2 adapter

Router

(5) frame packet

(6) frame packet

(7) frame packet

(8) frame packet

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

LAN1

Host A

client

protocol software

LAN1 adapter

(1) data

(3) frame packet

(3) frame packet

(3) frame packet

(4) frame packet

Router

LAN2

Host B

server

protocol software

LAN2 adapter

(8) frame packet

(7) frame packet

(6) frame packet

(5) frame packet

PH: Internet packet header
FH: LAN frame header
Networks as Abstractions

LAN1

(1) data

(3) frame packet

(3) frame packet

(3) frame packet

(4) frame packet

Host A

client

protocol software

LAN1 adapter

Router

LAN2

(5) frame packet

(8) frame packet

(7) frame packet

(6) frame packet

Host B

server

protocol software

LAN2 adapter

etc

PH: Internet packet header
FH: LAN frame header
Packet Switching Delays
Packet Switching Delays

• As these packets flow through a network and are routed, we might see delays due to:
Packet Switching Delays

• As these packets flow through a network and are routed, we might see delays due to:
  • Propagation (traveling across the link, speed of light, etc)
Packet Switching Delays

• As these packets flow through a network and are routed, we might see delays due to:
  • Propagation (traveling across the link, speed of light, etc)
  • Transmission delay (big packets take longer to transmit)
Packet Switching Delays

As these packets flow through a network and are routed, we might see delays due to:

- Propagation (traveling across the link, speed of light, etc)
- Transmission delay (big packets take longer to transmit)
- Processing delay (once switch sees packet, might be slow to process)
Packet Switching Delays

- As these packets flow through a network and are routed, we might see delays due to:
  - Propagation (traveling across the link, speed of light, etc)
  - Transmission delay (big packets take longer to transmit)
  - Processing delay (once switch sees packet, might be slow to process)
  - Queuing delay (link might be busy)
Packet Loss
Packet Loss

• Some packets could be delayed, others might never reach their target, due to:
  • Buffers overflowing (e.g. on switch)
• Networks are usually considered best-effort
  • Aka third-class mail
  • We’ll try to get your packet there, but if it doesn’t, sorry.
• Solved by requiring recipient to send a confirmation message was received
  • If no confirmation received, assume didn’t get sent
• What happens to duplicates?
  • Each message includes a unique ID, can be discarded if duplicate received
Resending Packets

send request, set timer

A

request 1

B

time

set timer

reset timer

Fig © Saltzer & Kaashoek
Resending Packets

send request, set timer

receive response, reset timer

A

request 1

response 1

B

time

Fig © Saltzer & Kaashoek
Resending Packets

send request, set timer

receive response, reset timer

send request, set timer

request 1

response 1

request 2

X

X

overloaded forwarder discards request packet.

Fig © Saltzer & Kaashoek
Resending Packets

- send request, set timer
- receive response, reset timer
- send request, set timer
- timer expires, resend request, set new timer
- receive response, reset timer

A

- request 1
- response 1
- request 2
- X
- request 2'
- response 2'

B

- time
- overloaded forwarder discards request packet.

Fig © Saltzer & Kaashoek
Resending Packets

send request, set timer

request 3

overloaded forwarder discards response 3

Fig © Saltzer & Kaashoek
Resending Packets

- **send request**, set timer
- **timer expires**, resend request, set new timer
- **receive response**, reset timer
- **request 3**
- **request 3’**
- **response 3’**
- Overloaded forwarder discards response 3
- Duplicate arrives at B
- B sends response 3’
Resending Packets

send request, set timer → request 4

A

B

duplicate arrives at B, resend request 4

receive response, reset timer

duplicate gets delayed

set timer

receive response 4

Fig © Saltzer & Kaashoek
Resending Packets

- Send request, set timer
- Timer expires, resend
- Receive response, reset timer
- Request 4
- Response 4
- Request 4’
- Packet containing response gets delayed

Fig © Saltzer & Kaashoek
Resending Packets

- Send request, set timer
- Timer expires, resend
- Receive response, reset timer
- Receive duplicate response

A

- Request 4
- Response 4
- Request 4’
- Response 4’

X

B

- Packet containing response gets delayed
- Duplicate arrives at B
- B sends response 4’
Resending Packets

• That ID is **really** important to put on the packets!
• Note: it works, but can result in **lots** of duplicate packets sent back and forth
• Also, note: no guarantee that packets are delivered in order!
Networks as Abstractions

• Obviously, we don’t think or care about packets
• We think and care about sending data!
• We want abstractions, like RPC (Remote Procedure Calls)
• Abstractions hide the complexity of what’s below them
• Next class: all RPC
Networks as Abstractions

- Obviously, we don’t think or care about packets
- We think and care about sending data!
- We want abstractions, like RPC (Remote Procedure Calls)
- Abstractions hide the complexity of what’s below them
- Next class: all RPC
Networks as Abstractions

- Obviously, we don’t think or care about packets
- We think and care about sending data!
- We want abstractions, like RPC (Remote Procedure Calls)
- Abstractions hide the complexity of what’s below them
- Next class: all RPC

```java
addPerson(“Prof Bell”, “ENGR 4422”);
```
Networks as Abstractions

- Obviously, we don’t think or care about packets
- We think and care about sending data!
- We want abstractions, like RPC (Remote Procedure Calls)
- Abstractions hide the complexity of what’s below them
- Next class: all RPC

Client

addPerson("Prof Bell","ENGR 4422");

Server
Networks as Abstractions

- Obviously, we don’t think or care about packets
- We think and care about sending data!
- We want abstractions, like RPC (Remote Procedure Calls)
- Abstractions hide the complexity of what’s below them
- Next class: all RPC

```java
addPerson("Prof Bell", "ENGR 4422");
```
Networks as Abstractions

- Obviously, we don’t think or care about packets
- We think and care about sending data!
- We want abstractions, like RPC (Remote Procedure Calls)
- Abstractions hide the complexity of what’s below them
- Next class: all RPC
Networks as Abstractions

- Obviously, we don’t think or care about packets
- We think and care about sending data!
- We want abstractions, like RPC (Remote Procedure Calls)
- Abstractions hide the complexity of what’s below them
- Next class: all RPC

```java
Client
addPerson("Prof Bell", "ENGR 4422");

Server
addPerson("Prof Bell", "ENGR 4422");
```
3 Layer Abstraction

• The typical network abstraction model has 7 layers
  • Take CS 455 if you want to know more about these
• We’ll think about 3 abstraction layers, and really focus on the top one
3 Layer Abstraction

• The typical network abstraction model has 7 layers
  • Take CS 455 if you want to know more about these
  • We’ll think about 3 abstraction layers, and really focus on the top one

Link layer

Physical links: care about how to deliver packets
3 Layer Abstraction

• The typical network abstraction model has 7 layers
  • Take CS 455 if you want to know more about these
• We’ll think about 3 abstraction layers, and really focus on the top one

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link layer</td>
<td>Physical links: care about how to deliver packets</td>
</tr>
<tr>
<td>Network layer</td>
<td>Figures out where to send packets</td>
</tr>
</tbody>
</table>
# 3 Layer Abstraction

- The typical network abstraction model has 7 layers
  - Take CS 455 if you want to know more about these
- We’ll think about 3 abstraction layers, and really focus on the top one

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>End-to-End layer</strong></td>
<td>Handles packet loss, etc. Translates from application-data to packets, implements a protocol</td>
</tr>
<tr>
<td><strong>Network layer</strong></td>
<td>Figures out where to send packets</td>
</tr>
<tr>
<td><strong>Link layer</strong></td>
<td>Physical links: care about how to deliver packets</td>
</tr>
</tbody>
</table>
Transport Protocols

• Anything in the end-to-end layer is likely built on top of some lower level protocol (more abstractions)
• TCP, or UDP
• Data integrity (checksumming)
• Ordering control
• Flow control (not worrying about congestion)
Transport Protocols

• Anything in the end-to-end layer is likely built on top of some lower level protocol (more abstractions)
• TCP, or UDP
• Data integrity (checksumming)
• Ordering control
• Flow control
  (not worrying about congestion)
Transport Protocols

- Anything in the end-to-end layer is likely built on top of some lower level protocol (more abstractions)
- TCP, or UDP
- Data integrity (checksumming)
- Ordering control
- Flow control (not worrying about congestion)
Transport Protocols

- Anything in the end-to-end layer is likely built on top of some lower level protocol (more abstractions)
- TCP, or UDP
- Data integrity (checksumming)
- Ordering control
- Flow control (not worrying about congestion)
Reminder: Leaky Abstractions

- From this lecture, you should have found out that networks:
  - Can vary in
    - Data rates
    - Propagation, transmission, queuing and processing delays
  - Traverse hostile environments and may corrupt data or stop working
  - Even best-effort networks have:
    - Variable delays, transmission rates, can discard packets, duplicate packets, have a maximum packet length, can reorder packets
  - Even if using TCP, this can still show up!
    - Messages might still arrive late
Sockets as an Abstraction

• Simplest way to build our end-to-end layer is using a **socket**, which gives us an interface to TCP or UDP

• Socket looks **just** like reading/writing to a file (e.g. file descriptor in C, InputStream in Java)

• Sockets are identified by:
  • IP address - identifies the device on the network
  • Port number - identifies the application on the device
Preview for Next Class

Process 1

Hey, are you there?

Process 2

Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Preview for Next Class

Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Hey, are you there?

Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Hey, are you there?

Hey, are you there?

Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Preview for Next Class

Spoiler alert: You cannot tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Preview for Next Class

Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Preview for Next Class

Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!
Preview for Next Class

 Spoiler alert: You can not tell the difference in a distributed system between a computer failing and network being arbitrarily slow!

...well, I can still talk to these guys so I guess internet is ok
Socrative

• Reminder - class name is CS475