CSx55: Distributed Systems

[Distributed Servers]

The lore of Ahmdahl's law

Within each program lurks
    a serial soul and a parallel spirit
one runs alone
    the other splits and swarms

The parallel can be broken
    scattered on separate cores
The serial flies solo, immune to the rush
    needing its jolly time

But as you heap on cores
    the serial swells and comes to the fore
Its shadow the same as before
    but bigger in the grand design

Damping speedup
    dulling efficiency
teaching the law
    what cannot be split, commands it all

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the last class survey

- `ConcurrentHashMap`
  - Is 16 generally a good degree of concurrency?
  - Because it is using lock striping, when a thread is accessing an element, it is only acquiring one lock, correct?
  - What happens if there is a *concurrent modification*?
  - Is a weakly consistent iterator like a deepCopy?
  - But is it thread-safe? Focus seems to be primarily on performance?
- If you have a `Collections.unmodifiable` wrapper, do I need to worry about `ConcurrentModification`?
- Is `MutablePoint` being used in a thread-safe manner because we make copies?

Topics covered in this lecture

- Wrap-up of synchronizers
- Threads in Distributed Servers
- Server design issues

Latches

Latches

- Latch acts as a **gate**
  - Until latch reaches terminal state; *gate is closed* and no threads can pass
  - In the **terminal state**: gate *opens* and allows all threads to pass
- Once the latch reaches terminal state?
  - *Cannot change state* again
  - Remains *open forever*

When to use latches

- Ensure that a computation does not proceed until all resources that it needs are initialized
- Service does not start until other services *that it depends on* have started
- Waiting until all parties in an activity are ready to proceed
  - Multiplayer gaming

CountDownLatch

- Allows one or more threads to **wait for a set of events to occur**
- Latch state has a counter initialized to a *positive number*
  - This is the number of events to wait for
- `countDown()` decrements the counter indicating that an event has occurred
  - `await()` method waits for the counter to reach **0**

Using CountDownLatch

```
public class TestHarness {
   public long timeTasks(int nThreads, final Runnable task)
     throws InterruptedException {
     final CountDownLatch startGate = new CountDownLatch(1);
     final CountDownLatch endGate=new CountDownLatch(nThreads);

     for (int i=0; i < nThreads; i++) {
       Thread t = new Thread() {
         public void run() {
           try {
            startGate.await();
            task.run();
           } finally {
              endGate.countDown();
           }
         }
       };
      t.start();
     }
   long start = System.nanoTime();
   startGate.countDown();
   endGate.await();
   long end = System.nanoTime();
   return end-start;
}
```

Semaphores

- Counting semaphores control the **number of activities** that can:
  - Access a certain resource
  - Perform a given action
- Used to implement resource pools or impose bounds on a collection

Semaphores

- Manage a set of virtual **permits**
  - Initial number passed to the constructor
- Activities *acquire* and *release* permits
- If **no permits** are available?
  - `acquire` *blocks* until one is available
- The `release` method returns a permit to the semaphore

Semaphores are useful for implementing resource pools

- Block if the pool is empty
  - Unblock if the pool is non-empty
- Initialize a semaphore to the **pool size**
- `acquire` a permit before trying to fetch a resource from pool
- `release` the permit after putting the resource back in pool
- `acquire` **blocks** until the pool is non-empty

Binary semaphores

- Semaphore with an **initial count of 1**
- Can be used as a *mutex* with non-reentrant locking semantics
  - Whoever holds the sole permit holds the mutex

Using Semaphores to bound a collection

```
public BoundedHashSet<T> {
   private final Set<T> set;
   private final Semaphore sem;

   public BoundedHashSet(int bound) {
     this.set = Collections.synchronizedSet(new HashSet<T>());
     sem = new Semaphore(bound);
   }

   public boolean add(T o) throws InterruptedException {
     sem.acquire();
     boolean wasAdded = false;
     try {
       wasAdded = set.add(o);
       return wasAdded;
     } finally {
         if (!wasAdded) sem.release();
     }
   }

   public boolean remove(Object o) {
     boolean wasRemoved = set.remove(o);
     if (wasRemoved) sem.release();
     return wasRemoved;
   }
}
```

Barriers

- Barriers are similar to latches in that they **block a group of threads** till an event has occurred

- All threads must come together at **barrier point** *at the same time* to proceed
  - Latches wait for *events*, barriers ***wait for other threads***

Barriers and dinner …

- Family rendezvous protocol

- Everyone meet at Panera @ 6:00 pm;
  - Once you get there, stay there … till everyone shows up
  - Then we'll figure out what we do next

Barriers

- Often used in simulations where work to calculate one step can be done in parallel
  - But all work associated with a given step must complete ***before*** advancing to the next step
- All threads complete step k, before moving on to step $k+1$

CyclicBarrier

- Allows a fixed number of parties to *rendezvous* at a fixed point
- Useful in **parallel iterative algorithms**
  - Break problem into fixed number of independent subproblems
- Creation of a `CyclicBarrier`
  - ```
    Runnable  cyclicBarrierAction = ... ;
    CyclicBarrier cyclicBarrier =
                        new CyclicBarrier(2,  cyclicbarrierAction);
    ```
    ```

Using Cylic Barriers

```
class Solver {
   final int N;     final CyclicBarrier barrier;
   class Worker implements Runnable {
     int myRow;
     Worker(int row) { myRow = row; }
     public void run() {
       while (!done()) {
         processRow(myRow);
         try {
           barrier.await();
          } catch (BrokenBarrierException ex) {
           ...
         }
       }
     }
   }
   public Solver(float[][] matrix) {
     data = matrix;     N = matrix.length;
     barrier = new CyclicBarrier(N, new Runnable() { public void run() {
                                                        mergeRows(...);  } });
     for (int i = 0; i < N; ++i)
       new Thread(new Worker(i)).start(); //DO NOT START THREAD in constructor.
     waitUntilDone();
   }
```

Source: From the Java API

Exchanger

- Another type of barrier
- Two-party barrier
- Parties **exchange data** at the barrier point
- Useful when asymmetric activities are performed
  - Producer-consumer problem
- When 2 threads exchange objects via Exchanger
  - Safe publication of objects to other party

Thread Safety Summary

Thread Safety: Summary [1/4]

- It's all about *mutable*, *shared* **state**
  - The less mutable state there is, the easier it is to ensure thread-safety
- Make fields `final` unless they need to be mutable
- **Immutable** objects are automatically thread-safe
- **Encapsulation** makes it practical to manage complexity

Thread Safety: Summary [2/4]

- Guard each mutable variable with a **lock**
- Guard all variables in an invariant with the ***same lock***
- Hold locks for the *duration* of compound actions

Thread Safety: Summary [3/4]

- Program that accesses mutable variables from multiple threads without synchronization?
  - Broken program
- Include thread-safety in the design process
  - Document if your class is not thread-safe
- Document your synchronization policy

Thread Safety: Summary [4/4]

- Rather than scattering access to shared state throughout your programs and attempting *ad hoc* reasoning about interleaved access
  - Structure program to facilitate reasoning about concurrency
  - Use a set of standard synchronization primitives to control access to shared state

Is this the real life? Is this just fantasy?
Caught in a landslide, no escape from reality
Open your eyes, look up to the skies and see

Bohemian Rhapsody; Freddie Mercury; Queen

PERFORMANCE

Measures of performance

- Service time
- Latency
- Throughput
- Capacity
- Efficiency
- Scalability

How fast?

How much?

Performance and Scalability

- Tuning for performance
  - Do **same** work with **less** effort
  - Caching, choice of algorithms $O(n^2)$ to $O(nlogn)$

How fast?

- Scalability
  - Find ways to parallelize problem
  - Do **more** work with **more** resources

How much?

How fast and How much

- Are separate and can (at times) be at **odds** with each other
- To scale or for better hardware utilization
  - We often end up **increasing** the amount of work for each task
  - Divide tasks into multiple **pipelined** tasks
    - Orchestration overhead

The quest for performance

- What do you mean by **faster**?
- Under what **conditions**?
  - Small or large datasets
  - Perform measurements to substantiate arguments
- **How often** do these conditions arise?
- What are the **hidden costs**?
  - Development/maintenance risks
  - Tradeoffs
  - Ripple effects of decision

Avoid premature optimizations

- First make it **right, then fast**
- **Measure**, don't guess
- Quest for performance is one of the biggest source of **bugs**

Amdahl's Law

How much can we speed things up?

- Harvesting crops
  - The more the number of workers
  - The faster the crop can be harvested
- But some things are fundamentally serial
  - Adding additional workers does not make the crop grow faster

The right tool for the right job: Everything is not a nail

- Make sure that problem is **amenable** to parallel decomposition
- Most programs have a **mix** of parallelizable and serial portions

Amdahl's law describes how much a program can be theoretically sped up

- F : Fraction of components that must be executed serially
- N : Number of available processors

$$Speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

$$Utilization = \frac{Speedup}{N}$$

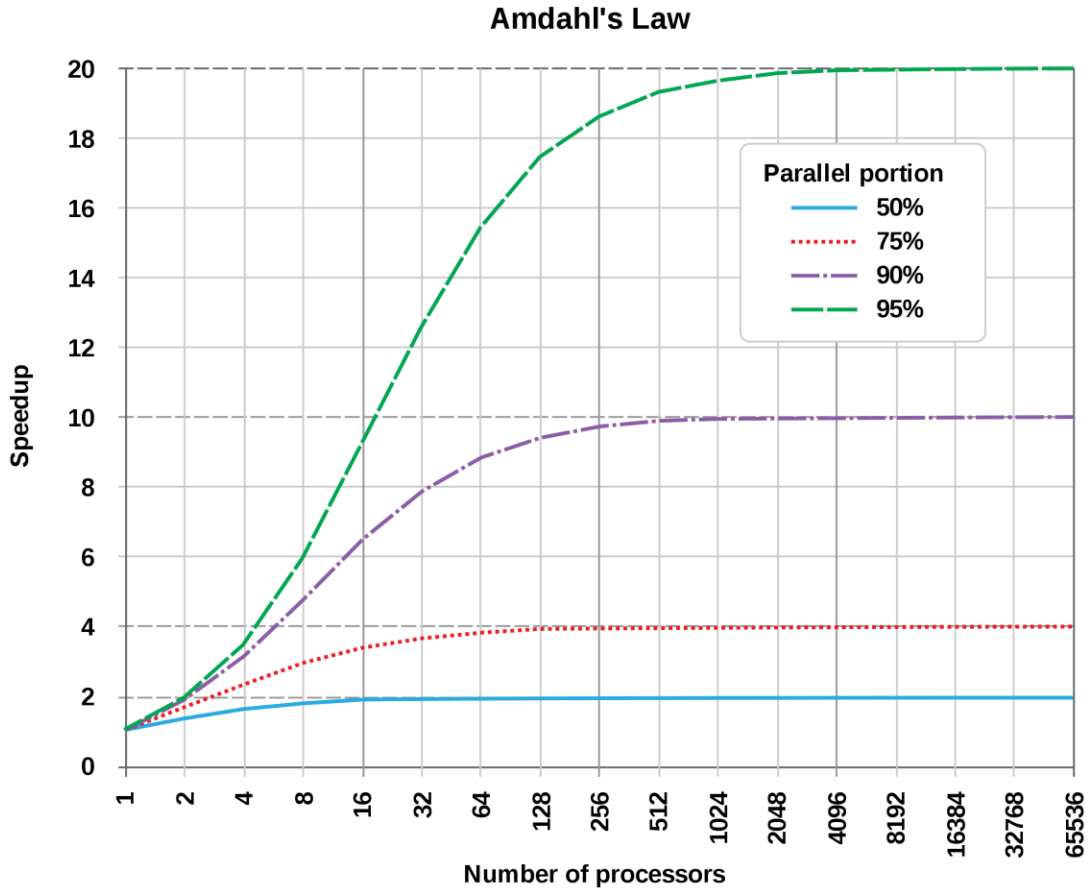
As N approaches infinity; maximum speedup converges to 1/F

- With 50% serial code
  - Maximum speedup is 2
- With 10% serial code
  - Maximum speedup is 10
  - With N= 10
    - Speedup = 5.3 at 53% utilization
  - With N= 100
    - Speedup = 9.2 at 9% utilization

What cannot be parallelized (i.e., the serial component) only grows in importance!

The serial part never shrinks;
add more processors, and it only looms larger.

Speedups for different parallelization portions



Source: http://en.wikipedia.org/wiki/Amdahl's_law

Know what to speed up

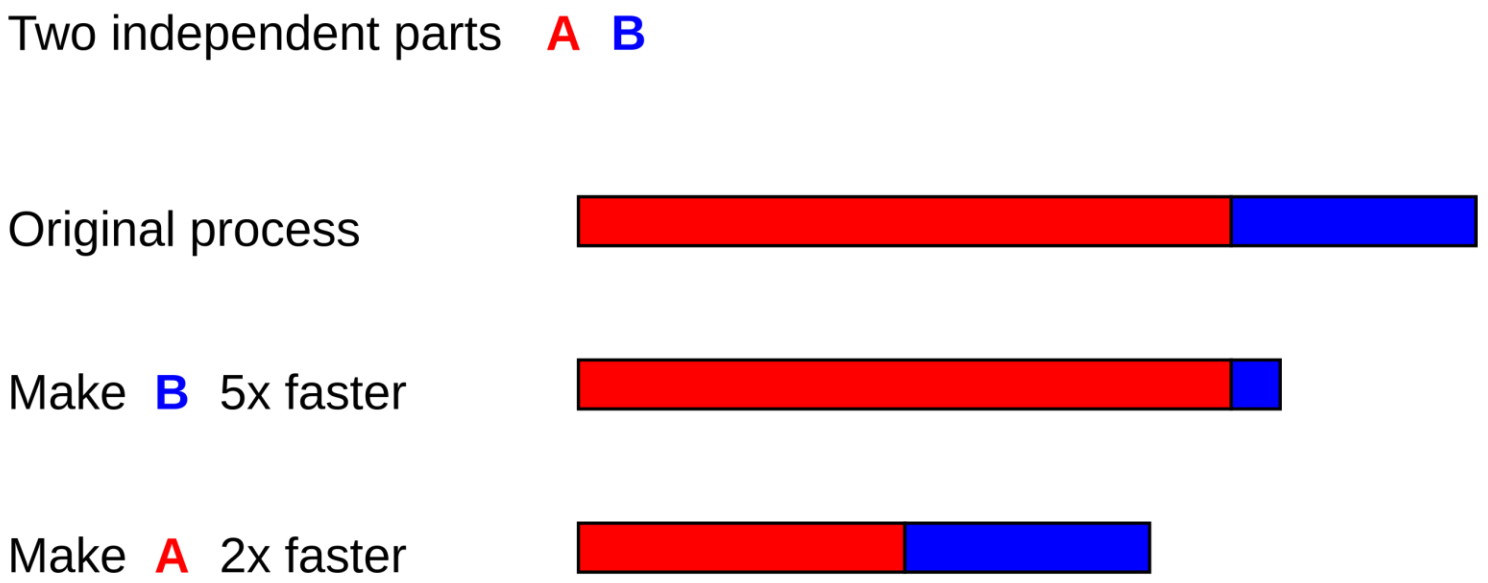


Image from: http://en.wikipedia.org/wiki/Amdahl's_law

Threads in Distributed Systems

Threads in distributed systems: Multithreaded clients

- **Hide** communication latencies
  - Initiate communications
  - Immediately do something else

Interleave

- Web browsers
  - As soon as main HTML page is fetched
    - Display it
  - Activate threads to retrieve other data types

Identical Code

Several connections can be opened simultaneously

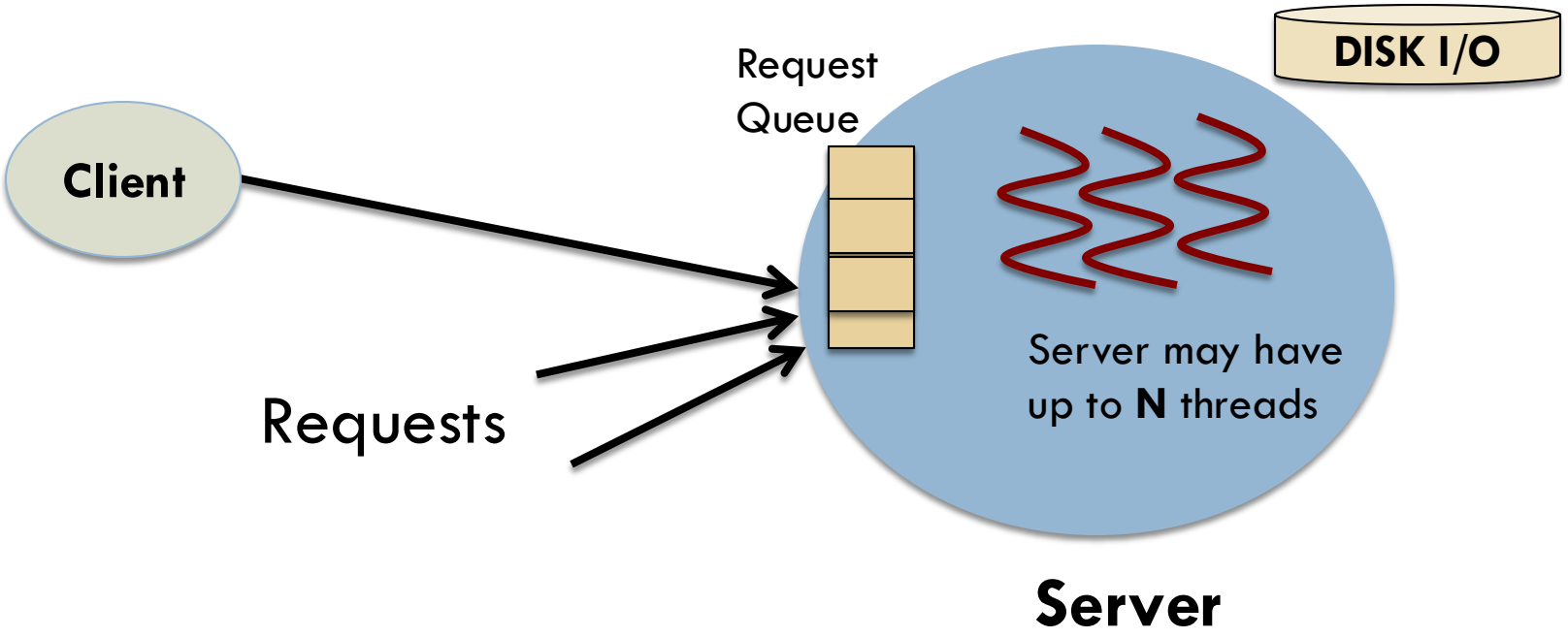
- To the same server
  - If the server is overloaded; things get even slower
- To replicated servers
  - Data transfer in **parallel**
  - Much faster rendering of content

Multithreaded Servers

- Simplifies server code
- Easier to develop servers that exploit parallelism
- E.g.: Handling concurrent connections
  - Each connection managed by a different thread
  - Multiple connections handled by a **pool** of threads

An example of performance improvements with Threads

Client and Server with Threads



Server side processing

- Server has **queue** of requests received from clients
- Server also has a **pool** of one or more threads
  - Each thread repeatedly removes requests & processes it
- Each thread applies the same methods to process the requests
  - Each request takes 2 ms of processing PLUS 8 ms of I/O (when server reads from disk i.e., no caching)

Maximum server throughput with 1 thread

- The turnaround time for handling any request is 2+8 = 10 ms
- The server can handle 100 requests per second
- Any new requests that arrive while the thread is handling a request?
  - These will be queued

Server throughput with 2 threads

- We assume that the threads are independently schedulable
  - One thread can be scheduled while the other is blocked for I/O
- Thread T2 can process a second request when thread T1 is blocked, and vice versa
- This increases throughput … but both threads may be blocked for I/O on the single disk drive
- If all I/O requests are serialized and take 8 ms each?
  - Maximum throughput is 1000/8 = 125 requests/second

Server throughput with disk block **caching**

- Server keeps data that it reads in buffers
- When a server thread tries to retrieve data
  - It first examines the cache and avoids disk accesses if it finds data element there
- If the hit rate is 75%?
  - The mean I/O time per-request reduces to
    $(0.75 \text{ x } 0 + 0.25 \text{ x } 8) = 2$ milliseconds
- Maximum theoretical throughput?
  - Becomes 500 requests per second

But there are costs associated with caching

- Average processor time for a request increases
  - This is because it takes time to search for cached data for every operation
  - Let us assume that this is now 2.5 milliseconds
- The server can now handle 1000/2.5 requests per second i.e., 400

Let's look at caching plus multiple threads

- Each request takes about 2.5 (processing) + 2 (I/O)
  - Total time per request is now 4.5 mSecs when disk accesses are serialized
  - Each thread can do 1000/4.5 requests per second i.e., 222 requests/second
- With two threads?
  - 444 requests/second
- With three threads?
  - 500 requests (bound by the I/O time)

Threading Architectures for Servers

Worker pool architecture

- Server creates a fixed **pool** of worker threads to process requests
  - Pool is initialized when server starts up
- Incoming requests are placed into a queue
  - Workers *retrieve* requests (work units) from the queue and process them

Managing priorities in the worker pool?

- Introduce *multiple* queues
- Worker threads **scan** queues in the order of descending priority

Disadvantages of the worker pool model

- Number of worker threads is fixed
  - So, threads in the pool may be too few to adequately cope with the rate of requests
- Need to account for coordinated accesses to the shared queue

Thread-per-request architecture

- Worker thread is spawned for **each** incoming request
  - Worker thread *destroys itself* after processing request
- Advantages:
  - Threads do not contend for the shared work-queue
  - Throughput is potentially maximized
- Disadvantage
  - Overhead for thread creation and destruction operations

Thread-per-connection architecture

- Associates a thread per connection
- New worker thread created when a client makes a connection
  - Destroyed when client closes the connection
- Client may make many requests over the connection

Thread-per-object architecture

- Associate a thread with each remote object
- A separate thread receives requests and queues them
  - But there is a queue per-object

Thread-per-connection & Thread-per-object

- Advantages
  - Server benefits from lower thread management overheads compared to thread-per-request
- Disadvantages
  - Clients may be delayed when a worker thread has several outstanding requests, but another thread has no work to perform

The contents of this slide-set are based on the following references

- *Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van der Steen. 2nd Edition. Prentice Hall. ISBN: 0132392275/978-0132392273.* [Chapter 6, 2]
- *Distributed Systems: Concepts and Design. George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair. 5th Edition. Addison Wesley. ISBN: 978-0132143011.* [Chapter 7, 14]