

CS7680

**Special Topics in Computer Systems:**

# Programming Models for Distributed Computing

Fall 2016

Heather Miller

[heather@ccs.neu.edu](mailto:heather@ccs.neu.edu)

Office: WVH224 (temp) & WVH302D

Office hours: by appointment

Course webpage:

<http://heather.miller.am/teaching/cs7680>

# This course is:

A research seminar course. That means we will focus on:

- ▶ (primarily) on reading, analyzing, discussing research articles
- ▶ informally presenting and explaining scientific contributions,
- ▶ and working together to write up our insights for a broad technical audience

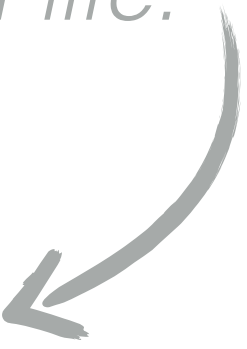
# Prerequisites

- ▶ A basic undergraduate CS curriculum
- ▶ Some familiarity with introductory PL concepts (or a willingness to learn)

More practitioner-focused, no hardcore PL theory required.

*PhD-level course. Open to upper-level undergraduates or MS students with permission.*

*skill that will help you  
in many walks of life:*



**At the end of this course,**

you should be not only be proficient at reading and digesting research papers, but at dissecting them, and clearly explaining the key insights and implications within them.

**As a side effect of this course,**

you'll learn a lot about the different sorts of distributed systems that are out there, when different systems are appropriate, and you'll be recognized writer :-)

# Outline:

**What this course is about**

Course structure/logistics

Programming Models

+

Distributed Systems

# Programming Models

Bridge the gap between an underlying runtime/architecture and the supporting levels of software available

Typically focused on achieving increased developer productivity

Typically provide guarantees to a programmer, and/or restrictions (hopefully helpful ones)

# Programming Models

 A moving target!

**Bridge the gap between an underlying runtime/architecture and the supporting levels of software available**

Typically focused on achieving increased developer productivity

Typically provide guarantees to a programmer, and/or restrictions  
(hopefully helpful ones)



# Programming Models

 A moving target!

**Bridge the gap between an underlying runtime/architecture and the supporting levels of software available**

Typically focused on achieving increased performance

**Sometimes: an abstraction over the underlying system/runtime, sometimes not.**

Typically provide guarantees to a programmer, and/or restrictions (hopefully helpful ones)

*Reading: A View of Cloud Computing (2010), see website*

# Programming Models

Bridge the gap between an underlying runtime/architecture and the supporting levels of software available

**Typically focused on achieving increased developer productivity**

Typically provide guarantees to a programmer, and/or restrictions (hopefully helpful ones)

**There's a bit of a human element to programming models.  
There's also a logical one.**

*That is, one that amenable to dynamic/static checking and/or verification.*

# Programming Models

Bridge the gap between an underlying runtime/architecture and the supporting levels of software available

Typically focused on achieving increased developer productivity

**Typically provide guarantees to a programmer, and/or restrictions (hopefully helpful ones)**

A system that makes weaker guarantees has more freedom of action, and hence potentially greater performance - but it is also potentially hard to reason about.

# Distributed Systems

**In a perfect world**, with unlimited resources, we wouldn't need distributed systems. We would we could just specify whatever resources we would need, and a machine with everything we need would always be available.

**Since we live in an imperfect world**, we have to figure out the right place on some sort of **cost-benefit curve** to place our system.

Most of what you've learned in undergrad will help you figure this out if your problem can largely fit on one machine, and upgrading your hardware as your problem grows usually works.

**However, if your problem grows, in some way, and upgrading your hardware on a single node isn't possible, you'll find yourself next in the world of distributed systems.**

# Distributed Systems

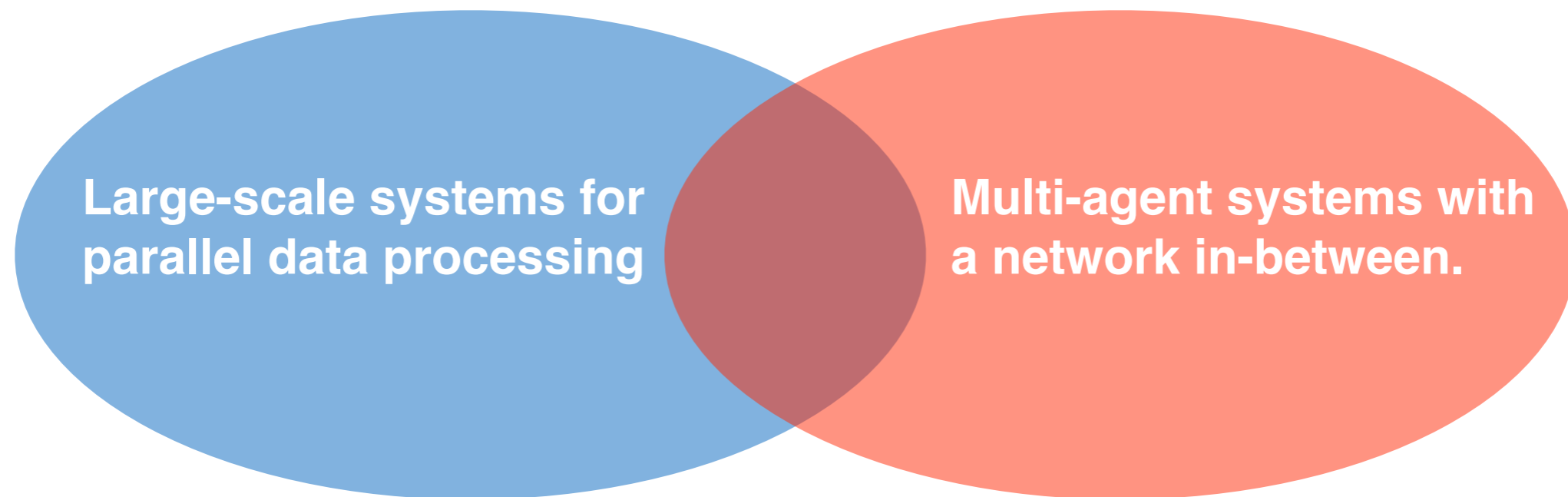
**However, if your problem grows, in some way, and upgrading your hardware on a single node isn't possible, you'll find yourself next in the world of distributed systems.**

## **Different scenarios that may require you to go distributed:**

- ▶ You need many independently-operating clients (games)
- ▶ You have too much work to do given the time/space you have to do it (big data)

# Distributed Systems

*in all cases,  
tens, hundreds,  
even thousands  
of nodes*



Large-scale systems for  
parallel data processing

Multi-agent systems with  
a network in-between.

**Think:** massive datasets that we  
want to develop insights from.

Doesn't have to be "big data" can  
just be "many heterogeneous clients"

**Think:** popular multiplayer games.  
Lots of frequently changing data that  
everyone wants access to.

Things that change when  
distribution happens:

*Everything.*

What makes *distribution* more different or more difficult to reason about?



(Jargon)

Things we now have to consider  
in the distributed case:

Scalability

Performance

Latency

Availability

Fault tolerance

*Reading: Introduction of Distributed Systems for Fun and Profit*

# Things we now have to consider in the distributed case:

## **Scalability**

*Performance*

*Latency*

*Availability*

*Fault tolerance*

is the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

# Things we now have to consider in the distributed case:

*Scalability*

**Performance**

*Latency*

*Availability*

*Fault tolerance*

is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.

# Things we now have to consider in the distributed case:

*Scalability*

*Performance*

**Latency**

*Availability*

*Fault tolerance*

is the time between when something happened and the time it has an impact or becomes visible.

# Things we now have to consider in the distributed case:

*Scalability*

*Performance*

*Latency*

**Availability**

*Fault tolerance*

the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable.

# Things we now have to consider in the distributed case:

*Scalability*

*Performance*

*Latency*

**Availability**

*Fault tolerance*

Availability =  
$$\text{uptime} / (\text{uptime} + \text{downtime})$$

*Availability % How much downtime is  
allowed per year?*

90% ("one nine") More than a month

99% ("two nines") Less than 4 days

99.9% ("three nines") Less than 9 hours

99.99% ("four nines") Less than an hour

99.999% ("five nines") ~ 5 minutes

99.9999% ("six nines") ~ 31 seconds

# Things we now have to consider in the distributed case:

*Scalability*

*Performance*

*Latency*

*Availability*

**Fault tolerance**

ability of a system to  
behave in a well-defined  
manner once faults occur

Things can (and do) go wrong.

In 1994, Peter Deutsch, a fellow at Sun, drafted a list of assumptions that architects and designers of distributed systems are likely to make, which prove wrong in the long run—resulting in all sorts of troubles.

## **The Fallacies of Distributed Computing:**

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

*Reading: Fallacies of Distributed Computing Explained, see website*



Why did I just bring all of these  
terms up?

Why are they relevant?

Why did I just bring all of these terms up?

Why are they relevant?

**All of this sneaks into programming models to varying degrees of intensity.**

Sometimes a model doesn't consider any of these, leaving the programmer to imagine all of the ways their system can go wrong, and to plan for it.

Other systems have varying degrees of solutions to these concerns already built in, freeing the programmer up from having to worry about them, like fault tolerance.

Why did I just bring all of these terms up?

Why are they relevant?

**All of this sneaks into programming models to varying degrees of intensity.**

Sometimes a model doesn't consider any of these, leaving the programmer to imagine all of their system's

Other concerns already built in, freeing the programmer up from having to worry about them, like fault tolerance.

**HINT: think about these terms when reading papers and writing your writeups!**

“ Back to programming models...

**This is where abstractions and models come into play.**

**Abstractions** make things more manageable by removing real-world aspects that are not relevant to solving a problem.

**Models** describe the key properties of a distributed system in a precise manner.

A good abstraction makes working with a system easier to understand, while capturing the factors that are relevant for a particular purpose.

A main recurring question throughout the rest of this course:

**How have models and systems out there been designed in view of all of these potential distribution-specific issues?**

# The sort of things we'll look at:

## **Large-scale parallel processing (batch)**

- ▶ Spark, MapReduce/Hadoop, DryadLINQ

## **Languages designed for distribution**

- ▶ Emerald, Argus, Linda, Orca, E

## **Inter-process communication**

- ▶ RPC & all of its benefits and flaws

## **Consistency & Coordination**

- ▶ CRDTs, and languages that take consistency into consideration

## **Languages extended for distribution**

- ▶ CloudHaskell, AliceML, Termite Scheme, ML5

## **Message Passing**

- ▶ The Actor Model, Erlang, Scala

## **Asynchronous Programming, Futures & Promises**

- ▶ Promises, MultiLisp, Oz, F# Async/Await, Finagle

## **Large-scale parallel processing (streaming)**

- ▶ Naiad, Twitter Heron

# Outline:

What this course is about

**Course structure/logistics**

This course is  
A research seminar course.

There are two main components.

- ▶ Weekly readings/writeups
- ▶ Final project



# Grading

You will be evaluated on:

- ▶ Your weekly research paper summaries (20%)
- ▶ Your semi-weekly paper presentations (15%)
- ▶ Participation in discussion (10%)
- ▶ One-time minuting of the group discussion (1hr)(5%)
- ▶ Your final project (50%)

Schedule

# Structure of the course

Every 1-2 weeks will be dedicated to a specific topic or programming model.

Each topic is covered by a selection of papers.

Each student will be responsible for a specific paper.

# Structure of class sessions

## **First half of the class:**

Each paper will have a 15-20 minute slot for a whiteboard presentation given by 1-3 students.

## **Second half of the class:**

Dedicated to a group discussion aimed at understanding the differences between each approach presented.

# Weekly responsibilities

- ▶ **Weekly reading** (1 paper, assigned)
- ▶ **Detailed summary/analysis** of your assigned paper. (~1-2pgs) **to be completed on your own!**
- ▶ **Whiteboard presentation** (group or solo) based on your writeup.

# Final Project

**A book of articles that we'll publish online.**

I expect it to generate a lot of interest in the open source community!

So, please keep this in mind throughout the course as you read/analyze/write! Your work may be used as by developers as reference material for years to come, so be thorough!

# Final Project

## **More specifically...**

A collection of extensive survey articles representing the history and current state of the art of a number of important topics at the confluence of distributed systems and programming languages.

# Final Project

Articles (or chapters) will correspond roughly to the weekly topics we cover together in the course.

**Said another way, every week, we will be, together as a class, making big steps towards the final project.**

Students may collaborate with one another on these articles, however, each student will take the responsibility as **lead** on one specific article/chapter.



# Final Project: Experimental Evaluations

While the focus of the final project is a polished writeup that we will work on together, **experimental evaluations/implementations are welcome to be included as well.**

If you wish to include some kind of implementation or experimental evaluation on your topic, please discuss and clear your ideas with me by the appropriate deadline.

# Project Organization/Timeline

- ▶ **September 29th** (or before): topics assigned/finalized
- ▶ **October 13th**: plans for final project experimental evaluations finalized

**1-on-1s:** You will be expected to briefly meet with me roughly every 3 weeks to discuss your progress. I expect you to begin devoting time to reading/structuring/sketching ideas early on.

# Weekly writeups

Summaries/analyses should be completed alone. However, after submission, writeups will be posted for the class to see/reference.

## **Your summaries should include the following:**

- ▶ a one or two sentence summary of the paper.
- ▶ a deeper, more extensive outline of the main points of the paper, including for example assumptions made, arguments presented, data analyzed, and conclusions drawn.
- ▶ any limitations or extensions you see for the ideas in the paper.
- ▶ your opinion of the paper; primarily, the quality of the ideas and its potential impact.

# Weekly writeups

Your writeup is due as a pull request to our class repo on **Thursdays between 5pm-6pm.**

*This is to ensure that everyone writes their own writeup without borrowing from someone else.*

**Repo:**

<https://github.com/heathermiller/cs7680>

# Whiteboard presentations

Typically in groups of 3, students work together to give an informal whiteboard presentation based on their weekly writeup.

We'll devote 15 minutes at the start of every class session to meet in groups to plan whiteboard presentations.

*It's generally a good idea to jot down an outline of your explanation, equations, or important points you'd like to make on a blank sheet of paper before coming to class and to carry this with you to your presentation so you have a reference sheet of thoughts you may want to write on the board while explaining.*

# Group discussion: minutes

It's expected that group discussions will provide lots of ideas and discussion points that will be very useful to the final writeup.

I will make audio recordings of the discussion section of the course, and each week 1 student will be in charge of the week's recording.

If you're minuting for the current session, **you are not required to submit a writeup or do a presentation at the next session** (though you still ought to read your assigned paper). Your minutes are due as a PR to the class repo by the start of the next class.

# Things to do right now:

- ▶ **Send me your github username so I can add you to the course repo.**
- ▶ **Select a research paper to read for next week.**
- ▶ **Sign up to minute**

# Note for next week...

Exceptionally for next week:

**Writeups are due at noon on September 15!**

*(So I can give everyone feedback on the first submitted writeups of the semester.)*