Lecture 1 (Sep 2)

Aurojit Panda

1 Plan for today

1.1 Motivating this class

1.1.1 Why distributed systems?

1.1.2 Why this class has you do what it does?

1. Why read papers.
2. Why talk about proofs, safety and liveness?
3. Why use Elixir?

1.2 Logistics

1.2.1 Reading and participation (5%)

1.2.2 Labs (40%)

1.2.3 Midterm (15%)

1.2.4 Final Exam (25%)

1.2.5 Final project (15%)

1.3 Getting started with distributed systems

2 Motivation

2.1 Why distributed systems

Prompted by three basic questions:

- How can we build reliable systems from unreliable components?

Let us start with the first: computers can fail. Hardware will eventually fail, software has bugs, etc. Can improve reliability somewhat by increasing cost and limiting complexity, but thus far we have not found a way to guarantee reliability. Things look worse when we consider computers in space ships, nuclear power plants or other extreme environments.

At the same time we use computers for a lot of safety critical uses.

NASA SIFT (Software Implemented Fault Tolerance) 1978: Fault tolerance for airline systems, adopted for all sorts of different things.

See SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control

- How can we build systems that require more compute than is available?

  Two limits to compute: (a) how much processing power is available; (b) IO how quickly can the processor read input data and write output data. Both have limits.

  This issue has always been important, someone always wants to compute something that needs more resources than currently available. However, they have become even more crucial as we arrive at the end of Dennard scaling, and soon Moore's law.

- How can we build programs and systems that allow people in very different geographic regions to collaborate and interact with each other?

  Saw the importance and benefits of this last year, but has been something we have wanted for a long time. Among other things allows us to reduce expenditure on equipment, enables scientific collaboration, etc.

Distributed systems is how we have addressed these questions. However, as opposed to many other abstractions (e.g., virtual memory) where we can hide complications and concerns from most programmers, this is not true for distributed systems. Getting them correct is challenging and usually requires understanding the algorithms, reasoning about their behavior (and performance) under failures, and reasoning about what happens when we compose these algorithms.

2.2 Why the various components of this class, and how to approach them?

2.2.1 Why read papers?

Stolen in part from advice given by Lindsey Kuper (`http://composition.al/CSE290Q-2019-09/course-overview.html`), who chose the advice from Blum, Keshav and Mitzenmacher.

This class requires that you read research papers. During a previous iteration some students provided feedback indicating that they would prefer a text book. I did not follow this feedback and want to explain why. There are primarily two reasons I think papers make more sense for this course:

First, and foremost similar to primary sources in other areas, a paper reveals the model used by the authors who developed the algorithms and ideas we are discussing. Much of the approach in distributed systems has depended on the model selected by the authors. For example, there are at least three different approaches that have been used thus far to explain some of the impossibility results in distributed systems: the approach taken by Lamport, FLP, and others where one looks at processes and indistinguishability of traces; approaches based on combinatorial topology developed by Herlihy and Shavits; and approaches based on knowledge developed by Halpern and Moses. We won't look at most of these approaches, but reading papers directly exposes you to the models and how authors have developed them. Textbooks tend to develop a model and reuse it, robbing you of the opportunity to experience all of them.

Second, there are not particular textbooks that closely match the sequence of papers we work through here. Also as a graduate student you need to be able to learn material and understand what is going on without needing text explicitly designed for classroom teaching.

That said, reading papers is a skill that you need to develop and that takes some effort. Several people have written about how to read papers. Much of this advice talks about how you might want to read papers out of order, and revisit portions as you gain a better understanding. Here is some advice that I think is useful:

- S. Keshav `http://ccr.sigcomm.org/online/files/p83-keshavA.pdf`
- Michael Mitzenmacher `https://www.eecs.harvard.edu/~michaelm/postscripts/ReadPaper.pdf`
- Manuel Blum: `http://www.cs.cmu.edu/~mblum/research/pdf/grad.html`

For the class we will help a bit with the reading: we will post a short summary of each paper and questions you should consider. This will happen about a week before the reading is due to give you some time. The summary is only meant to provide an overview of the problem addressed, and will not cover most of the paper. It is important that you read through the paper.

2.2.2 Lectures focus on desired correctness properties, invariants and why they hold?

One way to present this material would be to describe to you the various systems we study, list where they are used, and then move on. Unfortunately, that is not the approach followed by this class.

Instead the lectures focus on the properties these systems maintain, under what circumstances (e.g., what types of failures) they are maintained, and how this is accomplished. In the past some students have asked why focus on this since it is unlikely that most of you will have to go write proofs of correctness or even design new distributed algorithms.

There is a reason for this focus: while you might not be writing proofs or developing new protocols, it is extremely likely that over the course of your professional career you will have to compose multiple distributed systems together. The problem now is that you need to be able to reason about the performance, correctness and failure properties of the resulting system. This requires using many of the same reasoning tools we focus on in lectures.

Beyond this companies are starting to build provably correct software, so it might come in handy independent of this.

2.2.3 Labs

You are going to be implementing some of the systems we study in Elixir, an untyped functional language.

A common concern last year was around why we used Elixir and whether it was in fact a language used in practice?

To answer the question about practicality: Elixir is used in practice by companies such as Square Enix (make games), Facebook (where it is a part of what runs WhatsApp) and others. Builds on Erlang and has generally been adopted by many companies.

As for why: the main reason is that because Elixir was designed around the idea of actors the code you write for your assignment closely resembles the listings in the papers you are reading.

2.3 Logistics

- Reading and participation (5%)
- Labs (40%)
- Midterm (15%)
- Final Exam (25%)
- Final project (15%)

This class ends with a final project, however learning from previous experience we are going to split the class into two groups for this portion:

Research oriented: Some of you are PhD students who are probably working on research with your advisors. Others among you might not yet be PhD students but might either be already involved in research, or want to be involved. In this case you should see if you can relate your existing research to the themes of this class and use that as your final project. Generally, I will try to be very accommodating and so the link can be very weak. If you are in doubt set up some time with Panda, we are more likely than not to accept your existing research project. The main requirement here is that the research you are working on must be such that it is eventually meant to be submitted and published at a reasonable venue. You will need to submit a project proposal ($<$ 3 pages + unlimited citations) which should provide (a) a motivation for your work; (b) a survey of prior work; (c) a summary of your approach; and (d) how you plan to evaluate your approach. These are sections you will need for submitting your eventual paper anyways.

Not research oriented The second group is those who are not already involved in research and are not looking to start a project. For this case we will provide a list of open ended project proposals (by the 1st week of October). These proposals will build on the programming labs. For example a couple of thoughts I have been toying with are (a) use vector clocks (developed in Lab 2) to implement a logging layer that you can incorporate with other labs, and then having you write programs that analyze these logs; (b) extend and use the fuzzing logic that the labs build on to implement byzantine failures and use that for simple BFT protocols. These are not necessarily going to be on the list, and will definitely not be the only thing. The main properties of this list is that the suggestions are going to be open-ended and less well specified than the labs. The idea is to give you the opportunity to design and build your own system. If you take this option you will still need to submit a project proposal (again $<$ 3 pages) which must say (a) which of the options you are picking; (b) what your approach is going to be to address the problem; and (c) how are you going to evaluate your approach?

Expectations Regardless of the option you pick, towards the end of the class you need to submit two things: (a) a poster, for which we will provide a template, explaining what you did and what you found, and (b) a written report.

2.4 Collaboration

Short statement: you should work together, but you should not cheat. What this means is that you must have written and understood everything you turn in. Other people, webpage, etc. can help you improve understanding but shouldn't do your work for you.

Another short statement: Don't cheat: it causes problems for us, which in turn leads to problems for you.

You are encouraged to discuss the papers with each other. Going to try and set up ways in which you can figure out who else is in the class and find people to do this with: Campuswire is one option, but going to try finding others. Will also have a breakout room during Office Hours that you can use. That said **all code and written material** you submit must be yours, and you must understand what you are handing in. Please also cite anyone you talk to, any websites you consult, etc. Please see website for more detailed comments.

3 Getting Started

3.1 Multiple processes

Each can execute program logic

3.2 Connected by a network

Used by the processes to communicate with each other.

3.3 Want to run algorithms that use all of the processes.

3.4 Why complicated?

3.4.1 Assume processes can fail

Algorithm must work despite failures.

3.4.2 Assume very general networks:

- Delay messages
- Drop messages
- But do it fairly: drops are independent of the contents of the message.

More formally, the network is fair if a message sent by a process P to another process Q infinitely-often is delivered to Q infinitely-often. This allows for many pathological cases, but limits badness. Let us see what is in scope and what is not. Consider a case where P sends Q a message at each time step as shown below (R means message sent by P at time t was received, D means dropped)

Scenario	0	1	2	3	4	...	100	101	102	103	...
S1	R	D	R	D	R		D	R	D	R	
S2	D	D	D	D	D		R	D	R	D	

Really any case in which the network does not decide to dropp all messages with content M sent by P to Q, but allows all messages with content M' sent by P to Q is fine with this definition.

3.4.3 Example of how this makes things complicated

Two generals problem: Two generals, both need to agree on whether to attack or not.

- G1 $\rightarrow$ G2: let us attack. But message might be lost, so G1 should not attack.

- G2 $\rightarrow$ G1: yep let us attack. But message might be lost, in which case G1 won't attack, so G2 should not attack.

- G1 $\rightarrow$ G2: ok, I heard back from you. But again message might be lost ...

Seems awfully annoying: how do we get past this?

"Things are not as bad as the worse case": will formalize this notion on Oct 7.

3.4.4 Complexity of Distributed Systems

When thinking about algorithms we generally analyze asymptotic complexity as a way to understand how "long" the algorithm takes as a function of input lengths. In some cases this might also reflect on whether it is practical to use such algorithms as we increase the size of inputs, etc.

What are some equivalent notions in distributed systems?

1. Computational complexity: how much do additional processes improve algorithmic performance Polylogarithmic time: Problems that take $O((logn)^c)$ time with $O(n^k)$ processors for constants c and k. Decision problems in this class belong to NC (Nick's class), and are "efficiently solvable" on parallel computers.

   But in many cases we are not solving problems faster, just aiming to survive failures.

2. Message complexity: number of messages sent by the algorithm in terms of nodes and input. Tells us how much network capacity is required, but not necessarily how long things will take. Why?

   - P0 $\rightarrow$ P1
   - P1 does other stuff.
   - P1 $\rightarrow$ P0 algorithm done.

   Time is mostly spent waiting, very few messages.

3. Blocking vs non-blocking How much coordination is required between processes. Tries to capture a notion of whether to wait or not:

   Obstruction-freedom (Herlihy, Luchangco, Moir '03): If a process needs to eventually execute in isolation (i.e., no other process is alive) then it completes in bounded number of steps.

   Lock-freedom: At least one process makes progress eventually.

   Wait-freedom: All processes can make progress, regardless of what other processes do.

   All of those are binary, so a metric is to measure the number of times processes need to coordinate with each other.

3.4.5 Correctness of distributed systems

Much of our focus this semester is going to be on looking at whether a distributed system is correct. What exactly does correctness entail here? Want to think about two types of properties (also one of the readings in 2 weeks):

1. Safety Something bad does not happen. If violated once the system is wrong, no matter what happens after that. Example: mutual exclusion.

2. Liveness Something good will eventually happen (but eventually can be a really long time). Violations here mean that the system has arrived at a point where the property can never be met. Example: deadlock

3. Stable vs Unstable properties Addresses the question of when can we peek at the distributed system to decide that something bad happened.

   Stable: once violated the property remains violated. Unstable: violation is not visible after some time.