Contents

1 Plan 2

2 What and why distributed systems? 2
  2.1 Multiple processes 2
  2.2 Connected by a network 2
  2.3 Want to run algorithms that use all of the processes 2
  2.4 Why complicated? 2
    2.4.1 Assume processes can fail 2
    2.4.2 Assume very general networks 2
    2.4.3 Example of how this makes things complicated 3
    2.4.4 Is it really worth going through all this trouble? Why distributed systems? 3

3 Logistics and work 4
  3.1 Meetings 4
  3.2 Getting Help 4
  3.3 Giving feedback 5
  3.4 Workload 5
    3.4.1 Reading (20%) 5
    3.4.2 Programming Labs (40%) 6
    3.4.3 Final Project (40%) 6
  3.5 Collaboration 8

4 Back to technical content 8
  4.1 Complexity of Distributed Systems 8
    4.1.1 Computational complexity: how much do additional processes improve algorithmic performance 8
    4.1.2 Message complexity: number of messages sent by the algorithm in terms of nodes and input 8
1 Plan

- What and why distributed systems?
- Logistics and work
- More distributed systems
- Announcements

2 What and why distributed systems?

2.1 Multiple processes
Each can execute program logic

2.2 Connected by a network
Used by the processes to communicate with each other.

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

2.4 Why complicated?
2.4.1 Assume processes can fail
Algorithm must work despite failures.

2.4.2 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)

<table>
<thead>
<tr>
<th>Scenario</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>R</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>...</td>
</tr>
<tr>
<td>S2</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>R</td>
<td>R</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>...</td>
</tr>
</tbody>
</table>

Really any case in which the network does not decide to drop 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.

2.4.3 Example of how this makes things complicated

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

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

- G2 → G1: yep let us attack. But message might be lost, in which case G1 won’t attack, so G2 should not attack.

- G1 → 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.

2.4.4 Is it really worth going through all this trouble? Why distributed systems?

1. Machines fail => processes fail. Multiple machines is the only way we can deal with this.

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

2. Programs need to get inputs from different people, living in different buildings, cities, states.
   Internet. Contact tracing. Cellphones

3. Scaling computation to work around resource limits.
   Originally: how to work around temporary limits. Increasingly, the limits are more permanent: end of Dennard scaling and Moore’s law.
   We won’t really talk about scaling in this class. Other classes at NYU (big data, etc.) Seminar on this next semester.

3 Logistics and work

3.1 Meetings

• Lecture Wednesday 5-7pm

• Office Hours Panda: Tuesday, Thursday 2-3pm
  Changgeng: TBD

• Coding Help: Changgeng TBD

• Special just for next week (09/09): No lecture (Legislative day) but going to meet 5-7pm to go over Lab 1 and Elixir. Optional, no need to attend if you don’t want to. Please install Elixir and setup your programming environment before meeting, and try some of the project.

3.2 Getting Help

You can ask questions on Campuswire (should already be invited) or by e-mail.

Generally, technical questions about the reading or lab should go on Campuswire. That way other students can benefit from answers, and you can benefit from the wisdom of the crowd. Administrative questions (including ones about grades, etc.) and course suggestions should be sent over e-mail.
3.3 Giving feedback

This is my second time teaching this course, but the readings and labs have changed significantly. I am always interested in hearing about any concerns or suggestions for you might have about the course, since they are likely to improve both the current and future iterations of the course. Please e-mail me (apanda@cs.nyu.edu) any suggestions you might have.

3.4 Workload

3.4.1 Reading (20%)

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 Shavit; 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:
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.

### 3.4.2 Programming Labs (40%)

This class requires you to complete a set of programming labs. This is in part so it can serve as a capstone course in the department. Traditionally, programming labs in this class have been in Go. This year we are taking a different tack and requiring that the labs be completed in Elixir.

To justify this change a bit: in previous years, this class has traditionally been most people’s first introduction to Go. Thus we are not upping the complexity, the need to learn a new language largely carries over from previous years. That said in my previous attempt at teaching this class using Go, it also appeared that students often did not directly connect the papers to code. My hope is that Elixir will make this link clearer.

You do not need to have more than basic understanding of Elixir for any of the labs, and lab 1 is specifically designed to help you gain this basic understanding. In addition, Changgeng (who is TAing this class) is going to hold sessions to help with code, and we are going to have a lecture next Wednesday focused on teaching you the basics of Elixir. Being able to adjust to new languages is a good skill to have regardless of whether research projects or industrial employment are in your future.

Like every other class will tell you: start early the labs can take a bit of time. You get 5 late days you can use without penalty, otherwise 10% penalty per day late, but you cannot be more than 5 days late on any lab. Please refer to the webpage for details. Start early, get things done on time.

### 3.4.3 Final Project (40%)

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. The poster is due by 5pm ET on 12/09 and must be posted on Campuswire so other students can see and as you questions about it. Ideally you would present also present it during a poster session on 12/09 (during class), but I know that timezones might make this hard. (b) You must also submit a 6 page report (+ as many pages as needed for citations) by noon ET on 12/10.
3.5 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.

4 Back to technical content

4.1 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?

4.1.1 Computational complexity: how much do additional processes improve algorithmic performance

Polylogarithmic time: Problems that take $O((\log n)^c)$ 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.

4.1.2 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 → P1
- P1 does other stuff.
- P1 → P0 algorithm done.

Time is mostly spent waiting, very few messages.

4.1.3 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.

4.2 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):

4.2.1 Safety

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

4.2.2 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
4.2.3 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.

5 Announcements

• Make sure you have access to Campuswire. E-mail apanda@cs.nyu.edu if you do not.

• Fill out the Timezone survey: Link on Campuswire

• No lecture next week (Legislative Day): Instead we will have an optional walk through Lab 1 and Elixir. If attending please: (a) setup Elixir (see Lab 1) and work through the Elixir tutorial; (b) look through the lab code and figure out questions; and (c) be prepared to share your screen, etc.

• Office hours, lectures and recitation are all online this semester. Please e-mail to explicitly set up in-person meetings.