# Lecture 5 - Project Management + Distributed systems

## Sprints

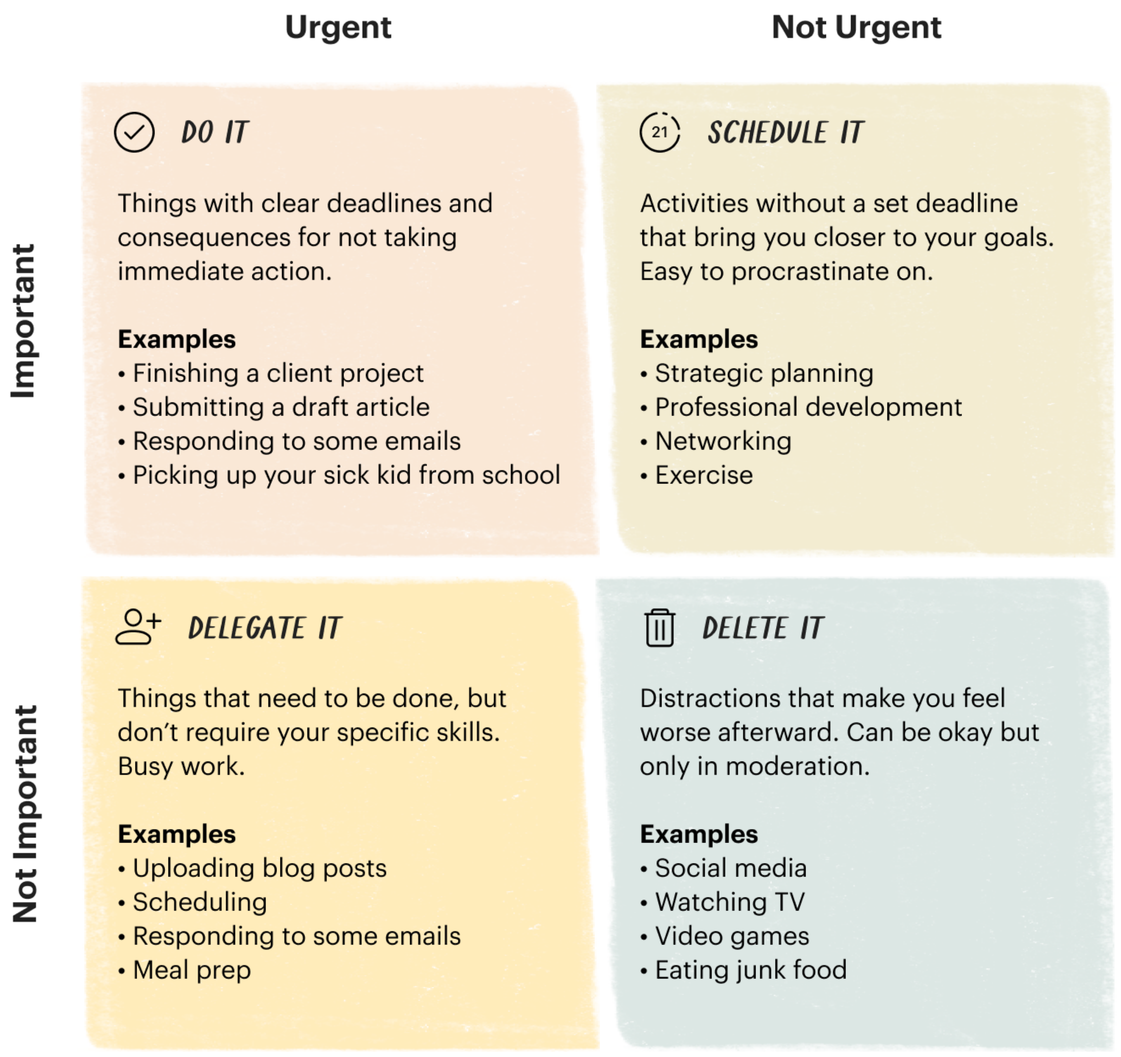
- How was the first sprint? How did you like working in a large group?
- Recall from the first lecture when we talked about agile vs waterfall. Where various different projects depend on the resolution you are looking at, it could be interpreted as an agile project or a waterfall project.
- For example, in any project, there is typically an overall planning phase where requirements are written for both product and engineering. We also typically have some idea of relative dependencies and target completion dates from the beginning.
- However, if you zoom in one level deeper and you will find a more “agile” day to day style of development, where things are structured into sprints and everyone makes constant micro adjustments while keeping the overall goals in mind.
- Once you zoom even deeper, typically each sprint is bookended on one side by planning and other side by sprint retrospectives. With further planning of dependencies etc through a waterfall process.
- Throughout this entire process, there is also a constant feedback loop between what’s happening on the “ground” against the initial plan for scoping and timeline. Where the team should be responsive to blockers and time overruns through judicious use of managing scope and shifting internal timelines around.
- Typically, it’s **not possible** to say to various stakeholders at the beginning saying “don’t enforce a structure and timelines, we are an agile team and we’ll get it done when it’s done”.
- Now we get to how a sprint actually works.
- **Planning**: What work can get done in this sprint and how will the chosen work get done?
- **Standup**: Regular checkins about how the work is progressing, goal is to not report status, but surface any blockers and questions
- **Retrospective/Review**: identify areas of improvement for the next sprint
- Common terms:
    - **Spike**: if no one knows the scope of a particular task, or the team is having issues estimating a certain task. A spike is created to investigate the task more so proper estimation can be created
    - **Velocity**: how much work is done by each person. Typically story points are used, and you can measure how many points per sprint an engineer works on.

## Critical Path

- Longest path of sequential activity from start to end of a user story. Therefore, the length of a critical path determines how long a user story is. Any delays in the critical path activities delays the project.
- **Step 1**: break down a user story into steps/activities
- **Step 2**: For each activity, which activity need to happen before this activity can be started
- **Step 3**: draw a dependency graph
- **Step 4**: longest path in the dependency graph is the critical path

## Prioritization

- How does one prioritize? Any ideas?
- Urgency VS importance
- One modification I would give to urgent vs important is the “urgent but not important”, oftentimes it’s not important “right now”. However, if you wait long enough, they will become important.
- In other cases, namely certain bugs you may get from time to time. Sometimes the bugs are fairly ephemeral and if you wait long enough, they will become not urgent and not important.



Urgent | Not Urgent

Important

**DO IT**

Things with clear deadlines and consequences for not taking immediate action.

**Examples**
- Finishing a client project
- Submitting a draft article
- Responding to some emails
- Picking up your sick kid from school

**SCHEDULE IT**

Activities without a set deadline that bring you closer to your goals. Easy to procrastinate on.

**Examples**
- Strategic planning
- Professional development
- Networking
- Exercise

Not Important

**DELEGATE IT**

Things that need to be done, but don't require your specific skills. Busy work.

**Examples**
- Uploading blog posts
- Scheduling
- Responding to some emails
- Meal prep

**DELETE IT**

Distractions that make you feel worse afterward. Can be okay but only in moderation.

**Examples**
- Social media
- Watching TV
- Video games
- Eating junk food

## Estimation:

- One important skill every software engineer must develop is a good sense of how long certain things take.
- However, it's a tough skill to learn and develop. Typically the knowledge you need is split into a few categories: Known Knowns, Known Unknowns, Unknow Unknowns.
- Most projects of an interesting size are filled with the two types of unknowns. If you have not encountered them yet, it most likely means you haven't thought deeply about it.
- One of the tactics i've developed is when faced with projects with unknowns. Start by breaking down the task into smaller pieces. As you break tasks down, you will start to find areas where you don't know, and you can isolate and research it through a spike task.

- Oftentimes, it's helpful to break down a task into 1-3 day granularity. Since most people have a pretty good grasp on how much work can be done in a day or a few days, it's useful tool to figure out a more accurate estimate.

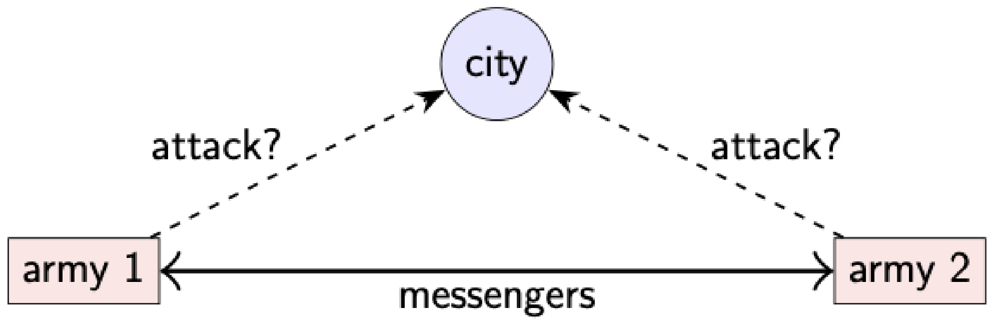
## Distributed Systems

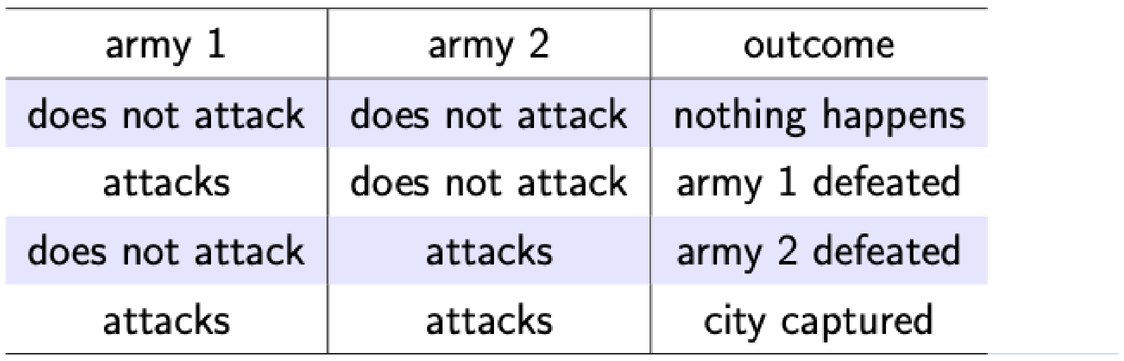
- What's the difference between Concurrent programs vs distributed systems?
    - Shared-memory concurrency. Most of the concurrency you learn in the context of a single system is how to deal with conflicts on the same memory space
- "... a system in which the failure of a computer you didn't even know existed can render your own computer unusable" – Leslie Lamport

### Why?

- Inherently distributed, communication between multiple different devices, peer to peer networks etc
- Better **reliability**: individual nodes can fail but the whole system stays up
- **Performance**: better data locality, getting information from a closer data center than one far away
- **Bigger problems**: the amount of data processed cannot fit on one machine, even a very big one

## Two generals problem

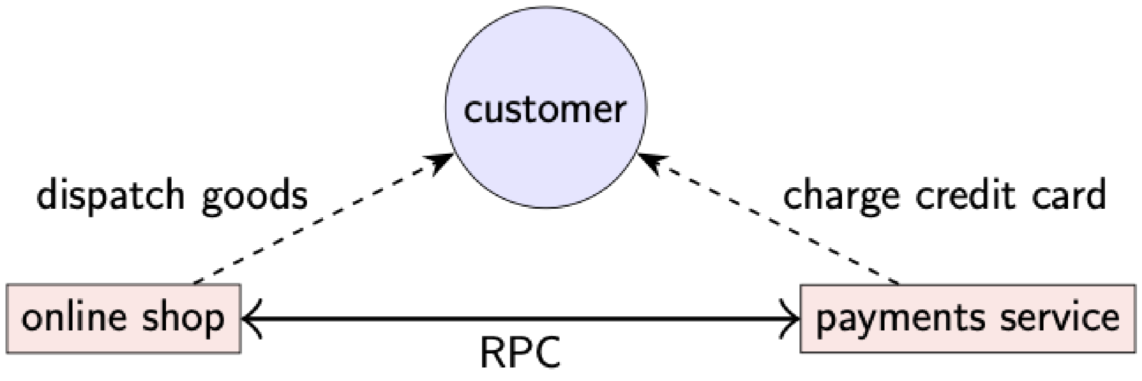




| army 1 | army 2 | outcome |
|---|---|---|
| does not attack | does not attack | nothing happens |
| attacks | does not attack | army 1 defeated |
| does not attack | attacks | army 2 defeated |
| attacks | attacks | city captured |

**Desired:** army 1 attacks *if and only if* army 2 attacks

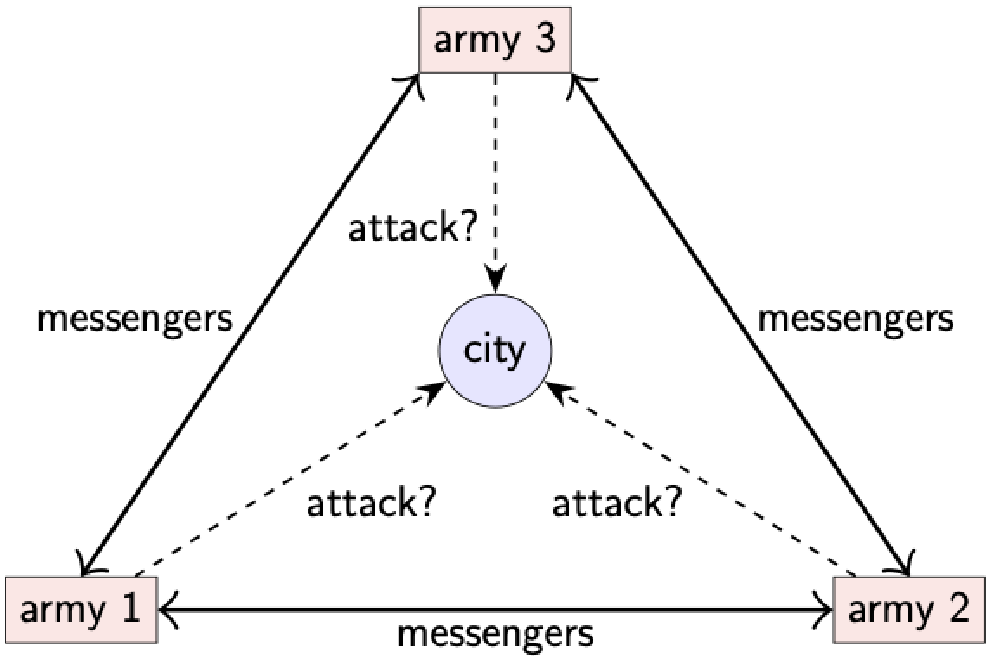
- Key distributed system definition is that you can only send messages between nodes in the system
- Here there are two armies, ideal case is neither attacks or both attacks. If only one army attacks, then the army that attacks will have been defeated.
- How would you coordinate the armies?
- If army 1 sends a message to army 2, and is intercepted by the city, vs if army 1 successfully sends a message to army 2, and army 2 sends a response back, and that gets intercepted. To army 1, the two cases are identical.
- General 1 always attacks, even if no response is received?
    - Send lots of messengers to increase probability that one will get through
    - If all are captured, general 2 does not know about the attack, so general 1 loses
- General 1 only attacks if positive response from general 2 is received?
    - Now general 1 is safe
    - But general 2 knows that general 1 will only attack if general 2's response gets through
    - Now general 2 is in the same situation as general 1 in option 1
- No common knowledge: the only way of knowing something is to communicate it



| online shop | payments service | outcome |
|---|---|---|
| does not dispatch | does not charge | nothing happens |
| dispatches | does not charge | shop loses money |
| does not dispatch | charges | customer complaint |
| dispatches | charges | everyone happy |

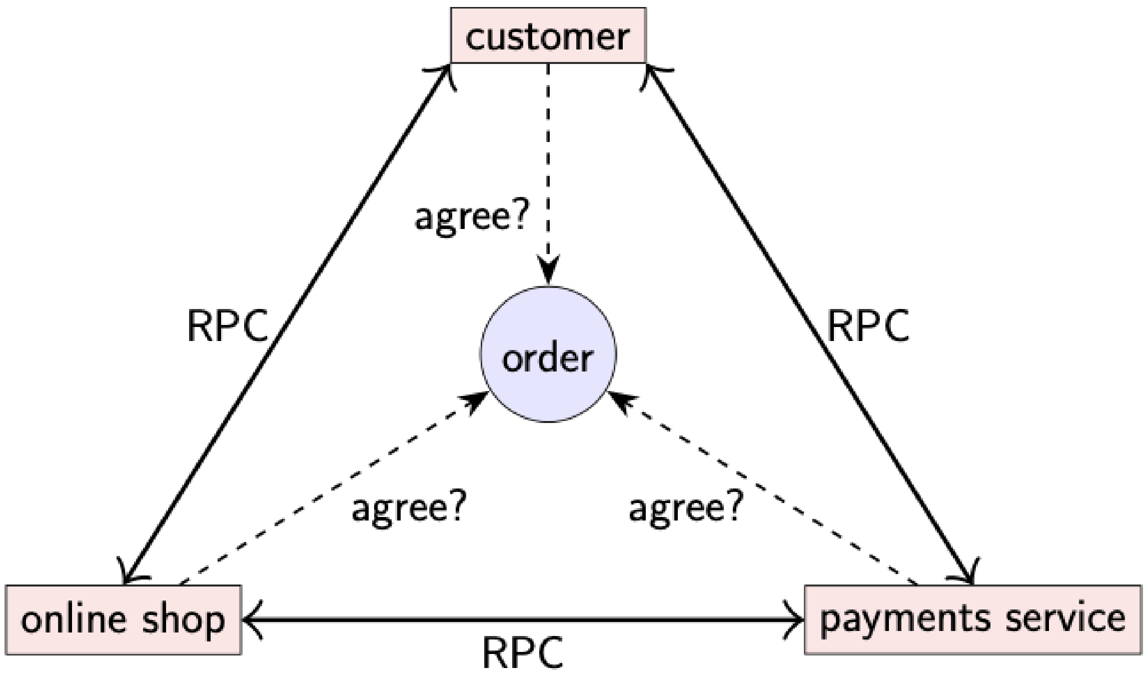
**Desired:** online shop dispatches *if and only if* payment made

## Byzantine Generals problem



**Problem:** some of the generals might be traitors

- Slightly different, we assume messages always arrive correctly, but models some generals might be traitors. And may lie about various messages being sent.
- For example, army 1 might tell different things to army 2 and army 3. However, if you are army 3's general, you have no idea whether general 1 is lying or general 2 is lying. It can be proved that byzantine general's problem can be solved if strictly fewer than ⅓ of generals are malicious.



Who can trust whom?