Elevator Project

Practical information and project introduction

Practical information

- 10 Exercises
  - Ex 4 and Ex 5 is part of the project and doesn’t require approval.
  - Ex 9 and Ex 10 is exam preparation (theoretical exercises) and doesn’t require approval.
  - Ex 1, 2, 3, 6, 7, and 8 requires approval to gain access to the exam.
  - You can get approval for an Exercise in the assigned week and the week after.
  - Completing the Exercise should not take the full lab slot, at least half of it should be spent working on the project.
  - You don’t need to complete everything in the exercise to get it approved. If you feel like you’ve learned what you can from it, explain the situation to a student assistant.

Practical information

- The elevator project
  - 25% of your final grade is decided by the elevator project.
  - You will receive 25p from the elevator project.
    - When doing the evaluations we might not use “standard grading scale”, that means the 25p will probably not correspond to the 25% directly.
  - There are 3 (4) partial evaluations.
    - Design review (8p)
    - Code (split in two parts) (9p)
    - Completion test (8p)
  - The code review is both going to teach you to write better code, and teach you to work with other people's code. You will be reviewing other groups code.

Practical information

- Design review. (8p)
  - 15-20min
  - A short presentation
  - Questioning about you elevator design.
  - https://github.com/TTK4145/Project/blob/master/EVALUATION.md#design-review
- Code review. (9p)
  - One part starting in the middle of the project where you get/give feedback on the code from/to your fellow students.
  - The second part in the end of the project evaluating how “good” your code is.
- Completion test. (8p)
  - A practical test of you elevator system.

Practical information

- If you’re going away in week 11 you should deliver Ex 8 together with Ex 7 in week 10.
- You are also allowed to deliver Ex 8 in week 14 but it is not recommended as you will be busy completing the project.
- Your project must be completed on April 9. (Monday week 15) As the Code Review starts on April 10.

Week	Deadlines	Exercises
2		
3		1
4		2
5		3
6	Design Presentation	4
7		5
8		6
9	Code Review (Preliminary)	
10		7
11		8
12		
13		
14		
15	Code Review	
16	FAT	
17		9+10

Practical information

- You must sign up for a lab slot.
- You should find a person you're compatible with to be your partner.
- If you sign up alone you will be matched with some other person that signed up alone.
- A group should consist of 2 persons, exceptions can be made if it's practical to do so.
- The friday session will some weeks be moved to 16:15 - 20:00

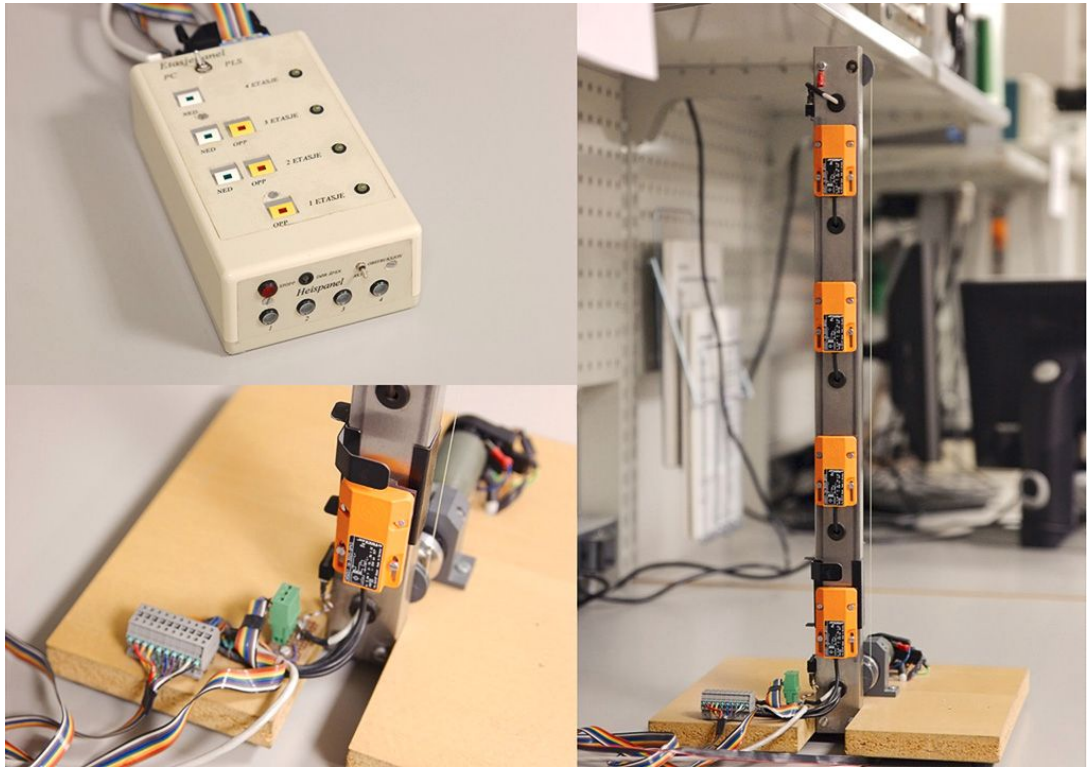
Timeplan for sanntidssalen (G203/G204, Gamle elektro) vår 2018

	Mandag	Tirsdag	Onsdag	Torsdag	Fredag
8-9					TTK4235 Tilpassede datasystemer
9-10					
10-11		TTK4235 Tilpassede datasystemer			
11-12					
12-13			TTK4235 Tilpassede datasystemer		
13-14					
14-15	TTK4145 Sanntidsprogrammering			TTK4235 Tilpassede datasystemer	TTK4145 Sanntidsprogrammering
15-16					
16-17		TTK4145 Sanntidsprogrammering	TTK4145 Sanntidsprogrammering		
17-18					
18-19					
19-20					

Practical information

- Github Classroom
  - We're using Github classroom for all code assignments (including the project).
  - If you haven't got one already, you will need a Github user account.
  - It will provide you with free github repos that you can keep private or open as you wish.
  - Even if you keep your repos private, the student assistants and course staff will have access to your code.
  - You can keep you repo private but grant some of your fellow students read access. This way you can cooperate between groups. (You will also do this for the groups reviewing your code)
  - It's possible to tag student assistants and course staff in the issues on both private and public repos. (Kjetil - @kjetilkjeka, Anders - @klasbo)
  - You are encouraged to use a picture of yourself in your Github profile. Remembering faces is often easier than github usernames.
  - I will give a short intro to Git/Github/Classroom at the end of the lecture.

The Elevator Project



The Elevator Project

- You are going to create a distributed fault tolerant elevator system.
  - It will be tested for three elevators over four floors but must be scalable.
- Distributed means:
  - The elevators will be running on different computers.
  - The network connecting the computers is not necessarily reliable.
- Fault tolerant means:
  - An elevator must arrive within reasonable time.
  - No matter what happens to the elevators, an accepted order can not be lost.
- If we’re not certain an order will be executed, we can refuse to accept it.

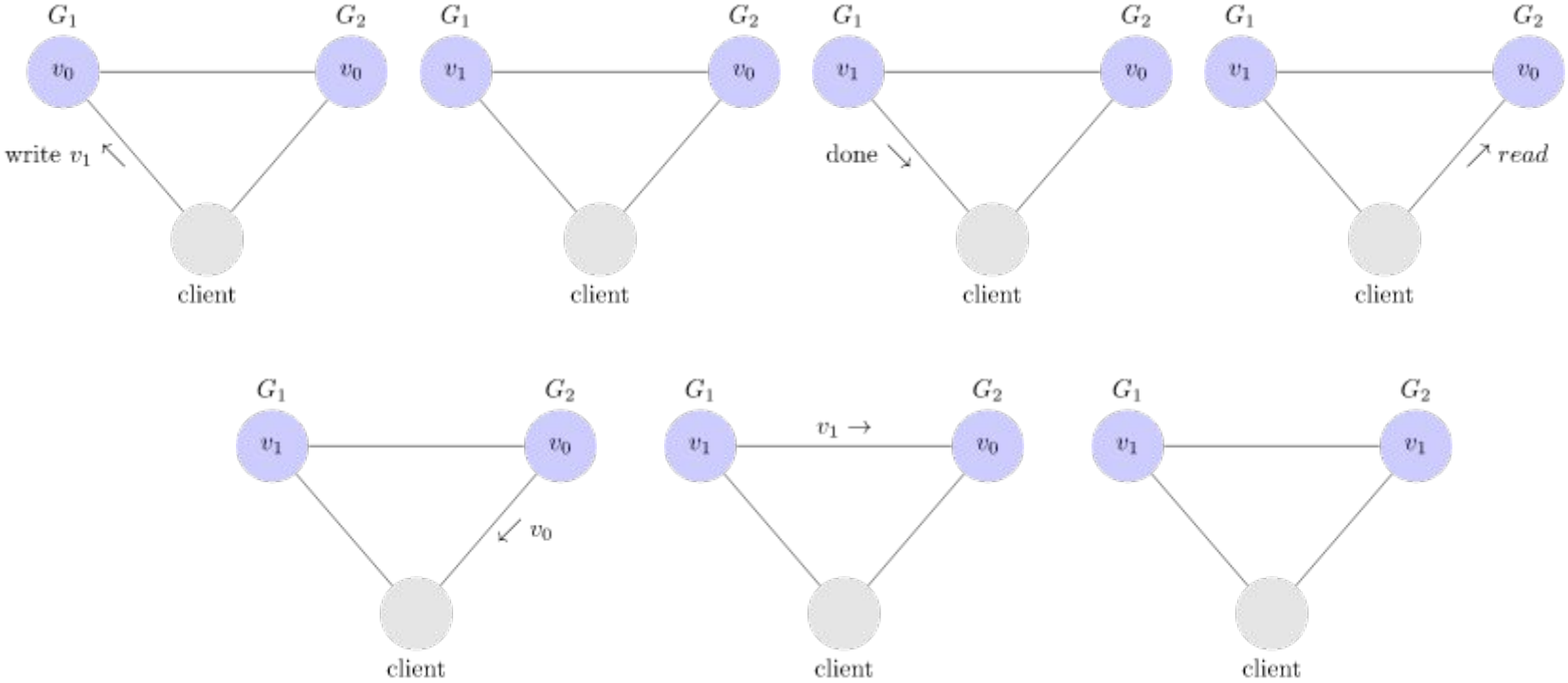
The Elevator Project

- You are already experienced in programming elevators.
- Would a magic module that synchronized all orders between all elevators solve the rest?
  - Yes!
  - We could add all state into the magic module and deterministically calculate how the all elevators should behave.
- Is this project about making such module?
  - No! (but kind of)

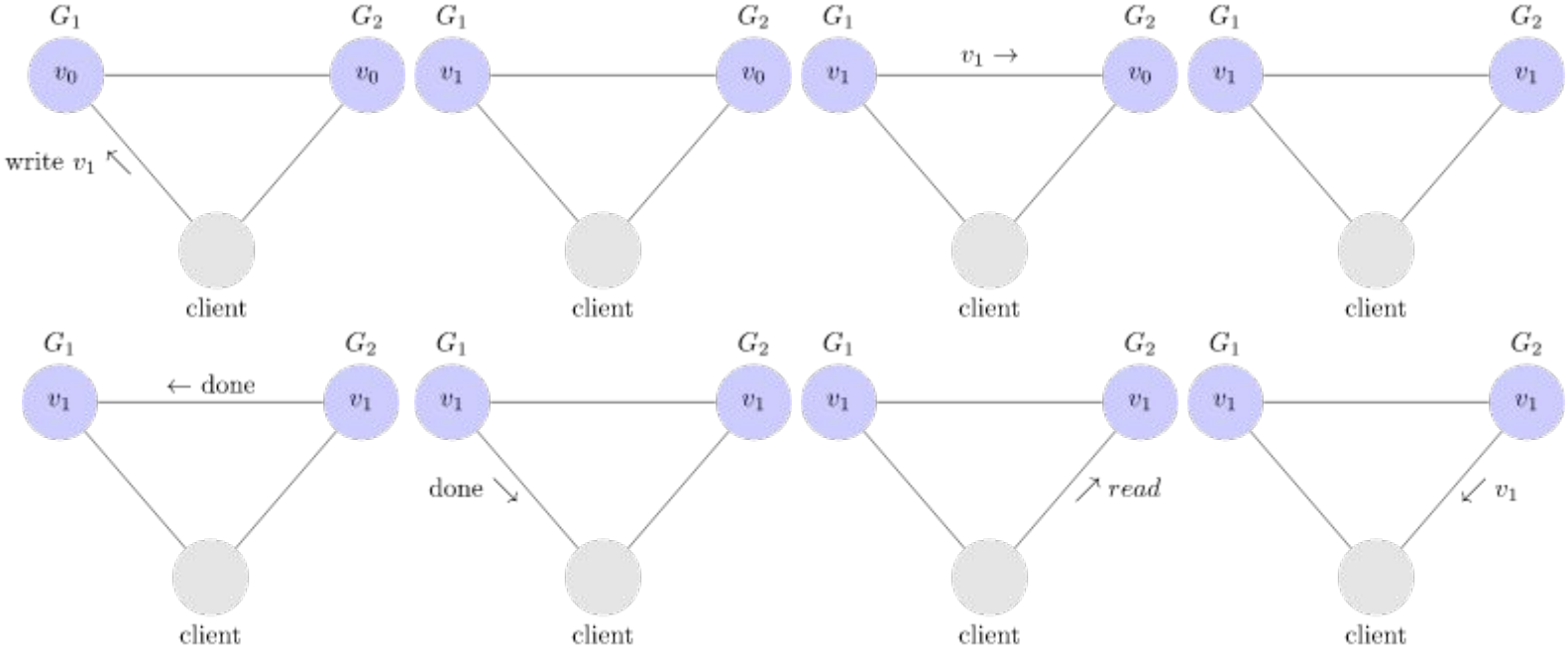
CAP Theorem

- For a distributed system (distributed data store)
  - Consistency:
    - Any read operation that begins after a write operation completes must return that value, or the result of a later write operation.

Inconsistency Example

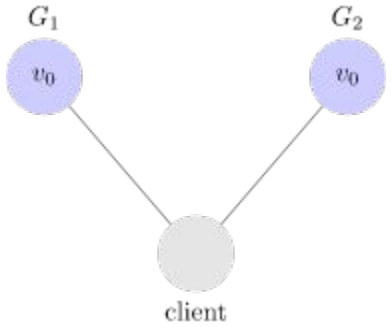


Consistency Example



CAP Theorem

- For a distributed system (distributed data store)
  - Consistency:
    - Any read operation that begins after a write operation completes must return that value, or the result of a later write operation.
  - Availability:
    - Every request received by a non-failing node in the system must result in a response.
  - Partition Tolerance:
    - The network will be allowed to lose arbitrarily many messages sent from one node to another.



CAP Theorem

- The CAP theorem states that you can, in a distributed system (distributed data store), have at most two out of three of the following properties.
  - Consistency
  - Availability
  - Partition tolerance
- You can (and should) have Consistency and Availability until Network Partitioning occurs.

Elevator specification

- Before any network partitioning has occurred your system should be both consistent and available.
- When network partitioning occurs, should we choose availability or consistency?
- We don’t really need consistency. We only need redundancy in the orders we accept. (we forfeit Consistency)
- If it’s possible that a single error will cause the order to disappear, we must refuse to accept the order. (we also forfeit availability in some cases)

Practical information contd.

- To solve the task at hand you're allowed to use:
  - Any programming language
  - Any library
  - Any design
- But there is one constraint:
  - The hardware is only supported in Linux. (but you can develop on Windows with a simulator)

Why any language?

- Because language shapes the way you think.

- Programming is:
    1. An idea of a system (what you want to create) …
    2. … translated …
    3. … into a formal framework (a programming language)

- If #3 is fixed, then #2 is constrained!
- #2 is the hard part. Make it as easy as possible.

But how do i choose language? (the correct answer)

- You don’t (yet), languages shapes the way you think.

- What you do instead is:
    1. You first learn what problem you’re going to solve.
    2. Then you find out how you want to solve the problem.
    3. Then you choose the language that suits your solution best.

- (This was what last slide was about, by the way)

But how do i choose language? (the realistic answer)

- You ask people what language they used last year and copy them if they say they liked it (i have statistics on this).
- Let me save you the trouble: (WARNING: this list is highly opinionated)
  - C/Posix, Python and Go is “officially supported”.
  - We have assistants able to help with Erlang, C++, Rust and D as well.
  - Statistically speaking, people will tell you to choose Go.
  - Kjetil thinks Erlang is better suited than Go (Go is still a good choice), but the learning curve only makes it the right choice if you think it would be fun to learn a different kind of programming language.
  - Anders will probably tell you to pick D over C++.

What former students said about some languages

- Go
  - Best thing about go: “Select, implementing timers with select is orgasm-inducing, amazingly simple and powerful channel stuff once you get the hang of it.
  - Worst thing: “Strictness, hard to develop in due to compiler abortion if an include/variable/function wasn’t used. Makes testing hard. Also a bit “unusual” to document. Not much example code resource on the internet.”
- C++
  - Best thing: “Templates (for meaningful abstractions, not when retardadly used as a turing complete language”
  - Best thing: “You learn a lot.”
  - Worst thing: “You learn too much.”

What former students said about some languages

- Java
  - Best thing: “Documentation/Online resources + Examples”
  - Worst thing: “Only OO, gets annoying after a while and leads to huge amounts of boilerplate. Everything is a class, except sometimes it isn’t. I used to have a problem, now i have a ProblemFactory”
- Erlang
  - Best thing: “The language is made for solving problem similar to what we encountered in the project . High amount of built in functionality for concurrency and fault tolerance”
  - Best thing: “Robust & concise (aka magical) - not much code, not much bugs. Also built-in connection & distribution functionality was a great basis”
  - Worst thing: “Having to learn a lot from scratch, few resources”
  - Worst thing: “Reading the official documentation”

Other languages the project has been done in.

- Rust
  - Aims to make concurrency “fearless” https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html
- D
  - Like C++, except not terrible (read: multi paradigm) (talk to Anders if interested)
- Racket
  - If you decide to do the project in Racket you’re either going to get a very good or a very bad score.
  - https://github.com/BourgondAries/elevators/blob/master/ (They have written a Post Mortem for the interested)

However

- You should probably not listen to what former students tell you about programming languages (or what anyone tells you about anything for that matter)
  - “I would use C again. Or possibly some jalla (I was not able to translate jalla to English) language like matlab of ArnoldC” - C user when asked about which language he would use if he were to start over.

euler1.arnoldc

```
      ENOUGH TALK
    BULLSHIT
      GET YOUR ASS TO MARS mod5
        DO IT NOW modulo n 5
      GET TO THE CHOPPER divisibleBy5
        HERE IS MY INVITATION mod5
        YOU ARE NOT YOU YOU ARE ME 0
      ENOUGH TALK
      BECAUSE I'M GOING TO SAY PLEASE divisibleBy5
        GET TO THE CHOPPER sum
          HERE IS MY INVITATION sum
          GET UP n
        ENOUGH TALK
      YOU HAVE NO RESPECT FOR LOGIC
    YOU HAVE NO RESPECT FOR LOGIC
  CHILL
  TALK TO THE HAND "sum:"
  TALK TO THE HAND sum
  YOU HAVE BEEN TERMINATED

  LISTEN TO ME VERY CAREFULLY modulo
  I NEED YOUR CLOTHES YOUR BOOTS AND YOUR MOTORCYCLE dividend
  I NEED YOUR CLOTHES YOUR BOOTS AND YOUR MOTORCYCLE divisor
  GIVE THESE PEOPLE AIR
    HEY CHRISTMAS TREE quotient
      YOU SET US UP 0
    HEY CHRISTMAS TREE remainder
      YOU SET US UP 0
    HEY CHRISTMAS TREE product
      YOU SET US UP 0
    GET TO THE CHOPPER quotient
      HERE IS MY INVITATION dividend
      HE HAD TO SPLIT divisor
    ENOUGH TALK
    GET TO THE CHOPPER product
      HERE IS MY INVITATION divisor
      YOU'RE FIRED quotient
    ENOUGH TALK
    GET TO THE CHOPPER remainder
      HERE IS MY INVITATION dividend
    GET DOWN product
  ENOUGH TALK
  I'LL BE BACK remainder
  HASTA LA VISTA, BABY

```

git/github/classroom

git (*plural* **gits**)

1. (Britain, slang, pejorative) A contemptible person.
2. (Britain, slang, pejorative) A silly, incompetent, stupid, annoying or childish person (usually a man)

https://en.wiktionary.org/wiki/git

"I'm an egotistical bastard, so I name all my projects after myself. First Linux, now git." - Linus Torvalds

git

- A distributed version control system.
  - Information model that is difficult to understand.
  - Crazy syntax (`git checkout <branch_name>` is totally different to `git checkout <file_name>`)
  - No abstractions, you need to keep track of every detail.
  - Allows rewriting of history quite liberally.
  - And is absolutely totally awesome!
- Fear not! You will hopefully not need any “advanced features”
  - If you still mess things up, ask a student assistant.
  - If the student assistants can’t help, ask Kjetil.

Practical demo

- Even though git is distributed, we're going to pretend it's centralized.
  - Github Classroom is going to create a repo for you on the TTK4145 organization for every Exercise and your project.
  - This repo will be the one true centralized repo.
  - Classroom demo time
- If you always do "pair programming" you can use a centralized workflow.
  - https://www.atlassian.com/git/tutorials/comparing-workflows#centralized-workflow
  - `git clone` -> `git add` -> `git commit` -> `git push`
  - Demo time (Ex1)
- If you work on different parts at the same time, use feature branching.
  - https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
  - Push to a separate branch and make a PR
  - You might get merge conflicts

Resources

- The Elevator Project:
  - https://github.com/TTK4145/Project/
- Illustrations CAP Theorem:
  - https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/
- 10 things i hate about git:
  - https://stevebennett.me/2012/02/24/10-things-i-hate-about-git/