# COMS 4113 Homeworks

Fall 2026

*The homework series, and HW1 in depth*

# Agenda



- Deadlines & how HWs factor into your grade
- Difficulty
  - Tips
  - Testing your assignments
- Overview of homeworks
  - Topics
- Grading specifics
  - HW2 - HW5
- HW1 - in depth
- Grading for HW1
- Questions

# Deadlines & Your Grade



## How they count

**8 homeworks in total**
(excludes hw0)

- Best **6 of 7** used for hw1 through hw4b ... **60%**
- **hw5 is a must** ... **10%**

- Total = **70%** of your final grade

| Coursework | Release date | Submission deadline | Time provided |
|---|---|---|---|
| **hw0** | Tue 09/08 | **Sun 09/13** | *5 days(!)* |
| **hw1** | Sun 09/13 | **Tue 09/22** | 1 week |
| **hw2a** | Sun 09/20 | **Tue 10/06** | 2 weeks |
| **hw2b** | Sun 10/04 | **Tue 10/13** | 1 week |
| **hw3a** | Sun 10/11 | **Tue 10/27** | 2 weeks |
| **hw3b** | Sun 10/25 | **Tue 11/10** | 2 weeks |
| **hw4a** | Sun 11/08 | **Tue 11/24** | 2 weeks |
| **hw4b** | Sun 11/22 | **Tue 12/08** | 2 weeks |
| **hw5** | Sun 12/06 | **Tue 12/22** | 2 weeks |

# Difficulty



HW1 < HW2 < HW3 < HW4 ≈ HW5 and **part a < part b**

## Tips across all homework assignments

- Read the papers and understand the protocol *before* coding
- Print your results frequently when debugging a distributed system
- Start your part B before the deadline of part A
- When designing your solution for part A, keep part B in mind
- Read ahead!
- Get familiar with what the tests expect for each assignment

# Testing your assignments



- Run the unit tests **at least 100 times**, ideally on more than one machine
- The result is **not deterministic** - especially for HW2-HW4 - because of goroutine and thread scheduling
- Passing **once** does **not** mean your code is correct, and it may still fail on our grading machine
- We will **not** add hidden tests cases. Everything you have is all we will run, but many many times
- You may add print statements to the unit tests - just remove them before submitting

A test that passes is one execution that happened to work. **A concurrency bug you did not hit is still in there.** That's why you should run tests many times, perhaps tens of times.

# Overview of homeworks



| Homework | Project | Related topics |
|---|---|---|
| HW1 | MapReduce | MapReduce, RPC |
| HW2 | Primary/Backup Key/Value Datastore | Fault tolerance |
| HW3 | Paxos & Key/Value Datastore | Consensus, Paxos, availability |
| HW4 | Sharded Key/Value Datastore | Scalability, Paxos, atomic commitment |
| HW5 | Model Checking the Paxos Protocol | Testing & model checking |

# Themes across all HWs



### Test-driven: the tests are provided

You are not guessing at the specification. The tests ARE the specification, and you can read them before you write a line.

### Most API skeletons are provided

- You fill them out
- You can add more logic and functions
- But for all given functions KEEP the function signatures the same

### Design, design, design

The coding is rarely the hard part. Deciding what state lives where, and who is allowed to change it, is.

### Data structures

- Some are provided
- Some you modify
- Some you write yourself
- Which need a lock, and what invariant it protects, recurs

# Grading



- Unit tests are used to grade your assignments
- Unit tests in the same homework each have the **same weight**
- Each unit test is run **many times**
- Every time a unit test fails, the score of that unit test is multiplied by **0.9**

| #fails | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|
| score | 100% | 90% | 81% | 73% | 66% | 59% | 53% | 48% | 43% | 7% |

The decay is *gradual*, but it never stops. Half your credit is gone by six failures.

# HW1 - MapReduce



Getting used to **RPC** and **Go**, on a simple Map/Reduce implementation.

### Part 1 - Map & Reduce

Write the map and reduce functions themselves. One process, no distribution, no failures. This part teaches you the shape of the computation.

### Part 2 - Distribute it

A master hands map and reduce jobs out to workers over RPC. Now there is a network, and the interesting questions start.

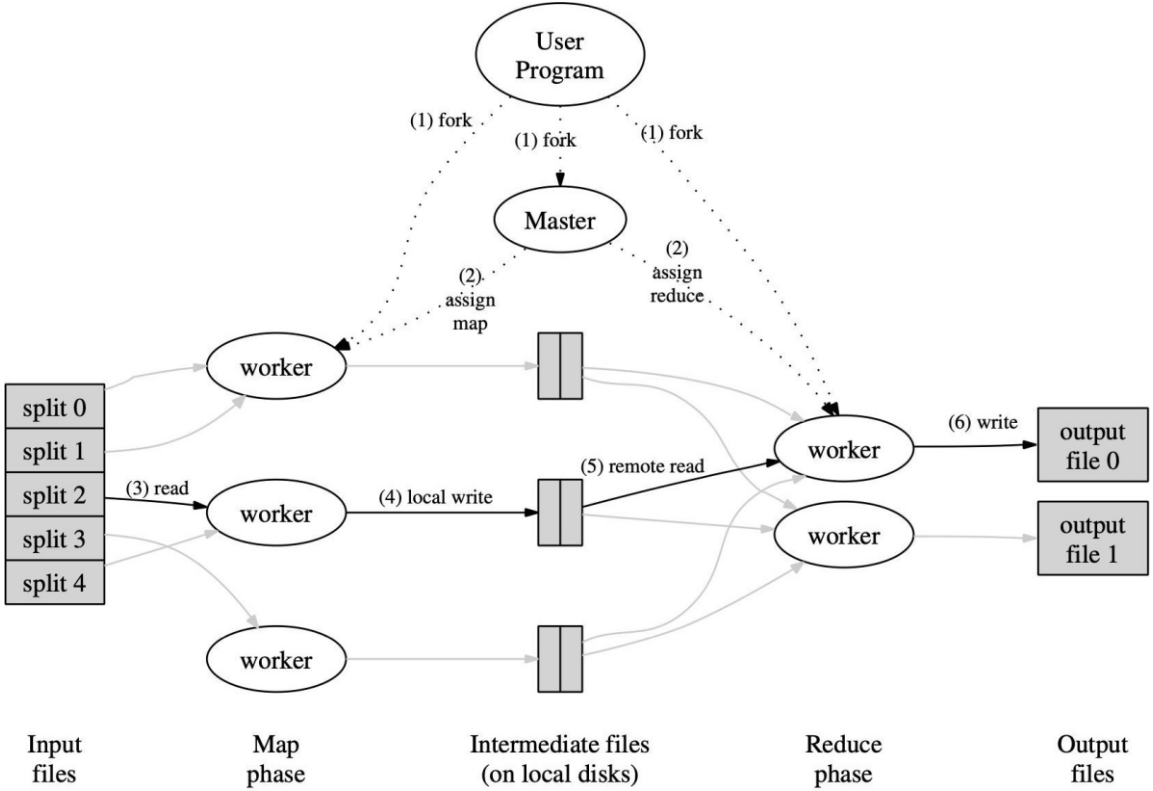
### Part 3 - Handle failures

A worker can die mid-job. The work still has to finish. This is the first fault tolerance you write in this course.

**Why MapReduce first?** Map and reduce tasks are deterministic and side-effect-free, so recovering from a dead worker is just *run it again somewhere else* - no state to reconcile, no agreement needed. Every later assignment loses that luxury.

# HW1 - MapReduce





# HW1 - WordCount, Step by Step


**Split the input:** `input-1.txt`, `input-2.txt`, ..., `input-m.txt`

## Map phase

*"..., a dog, a cat ..."* =>

```
{"a": ["1", "1"], "dog": ["1"], "cat": ["1"], ...}
```

**Intermediate files** - one per (map task, reduce partition) pair

| | |
|---|---|
| `int-file-1-1.txt` | `{"a": ["1", "1"], "cat": ["1"], ...}` |
| `int-file-1-2.txt` | `{"dog": ["1"], ...}` |
| `...` | |
| `int-file-m-r.txt` | |

With **m** splits and **r** partitions there are **m × r** intermediate files.

# HW1 - WordCount, Continued



**Reduce phase** - all values for one key, from every map task

```
{"a": ["1", "1"], "cat": ["1"], ...},
{"a": ["1", "1", "1"], "apple": ["1", "1"]}    =>
```

```
{"a": "5"}, {"apple": "2"}, {"cat": "1"}, ...
```

**Output:** `output-1.txt`, `output-2.txt`, ..., `output-r.txt`

Two things to notice. **r output files, not one** - the output is partitioned the same way the intermediate data was. And the counts add across *every* map task: "a" comes out as **5** because reduce sees every partial list for that key, from all m of them.

# HW1 - The Three Parts

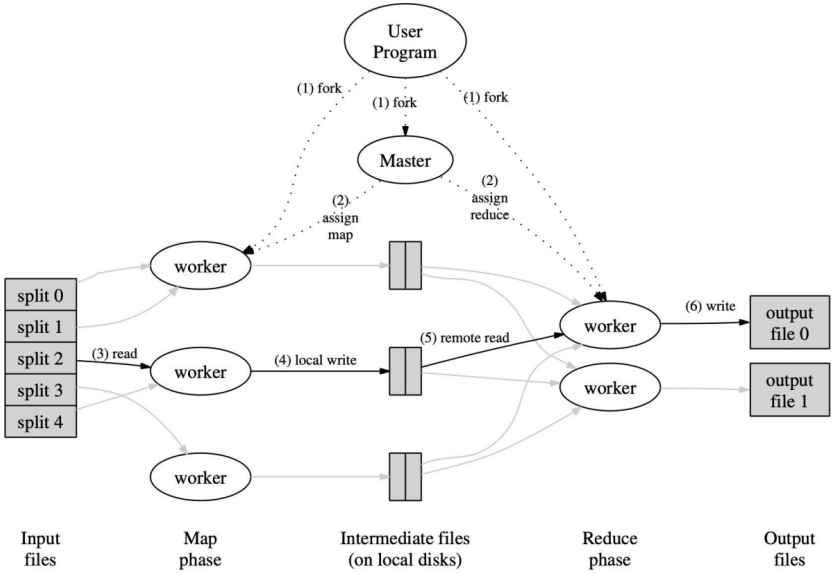


## Part 1

- Straightforward implementation
  - A few ways to run this (sequential, etc.)
  - The comments provide some instructions
- Map each word; count the mappings

## Part 2 / 3

- Map & Reduce are written for you
- You distribute the workload
  - Write a master that coordinates Map & Reduce jobs between workers
- Account for worker failures



# Grading for HW1



### Test 1 - 20 points (Part 1)

- `test-wc.sh`, under `src/main`
- **Pass or fail.** If it passes you get all 20; if not, this part scores 0.

### Test 2 - 100 points (Part 2/3)

- Follows the grading scheme from the earlier slide
- 50 runs per test, x0.9 per failure

The final grade is **120 points, scaled to 100.**

For example, if you get:

| 20 points in **Test 1** | 70 points in **Test 2** | (20 + 70) × 5/6 = **75** |
|---|---|---|

**Test 1 is all or nothing.** `test-wc.sh` either passes or you lose all 20 points - a sixth of the assignment - on the easiest part of the work. *Run it before you submit.*

Thanks to all prior TAs

# Questions?