Lecture 1

Paper to read: Map reduce
https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Infrastructure for Applications

"We'd love to have abstractions"

Discussing about necessary infra needed for applications:

- Storage: (Accessing data abstracted away from application)
- Communication: (MapReduce is a specific example)
- Compute: (Networking, Reliability etc.)

The overall goal is to make distributed systems in such a way that they provide clean abstractions, so applications don't have to worry about underlying nature of implementation (k8s is good example).

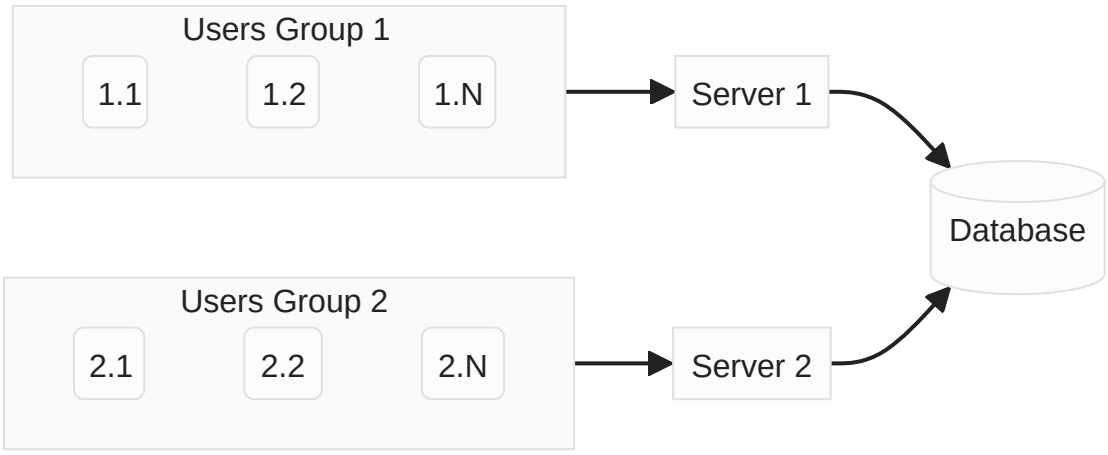
Common specific implementations include:

- RPCs
- Threads
- Concurrency (things about mutex, etc.)

Performance

"We are in general looking for scalabale performance"

- Buy computers not programmers :)



This kinda of scalability is basically theoretically.

So in this case adding more servers will not work since the bottleneck will become to Database.

Fault tolerance

"A system with a 1000s of computers there will always be a possibility that something will go wrong, that means one of the computers completely turning off"

Scale brings a lot of issues, since failures are common.

The overall idea is to hide/mask the failures that can occur in systems.

Main concepts under fault tolerance:

- Availability: make sure that application is atleast available, replicasets are answer to some extent given not all of them are failing
- Recover-ability: in general, making sure that systems recover in case of any anomaly
  - Replication: the biggest issue here is that state of replicas might change leading to inconsistency
- Consistency:
  - Consider a simple service:
    - Get(k) -> v
    - Put(k, v)
    - Now when there are more than one copy of this service, you can have multiple copies of state.
    - Consider this
      - Initially all the RSs had Get(1) -> 20, we need to update to 21
      - Put(1, 21) (rs1)
      - Now we need to update rs2, but it fails to update (rs2)
      - Get(1), now rs2 will return stale copy

        - To avoid such conditions we need to put some conditions (eg: eventual consistent systems)
        - Strong/Weak consistency strategies
    - Distributing service across regions to avoid natural disasters
        - But this leads to increased latency usually

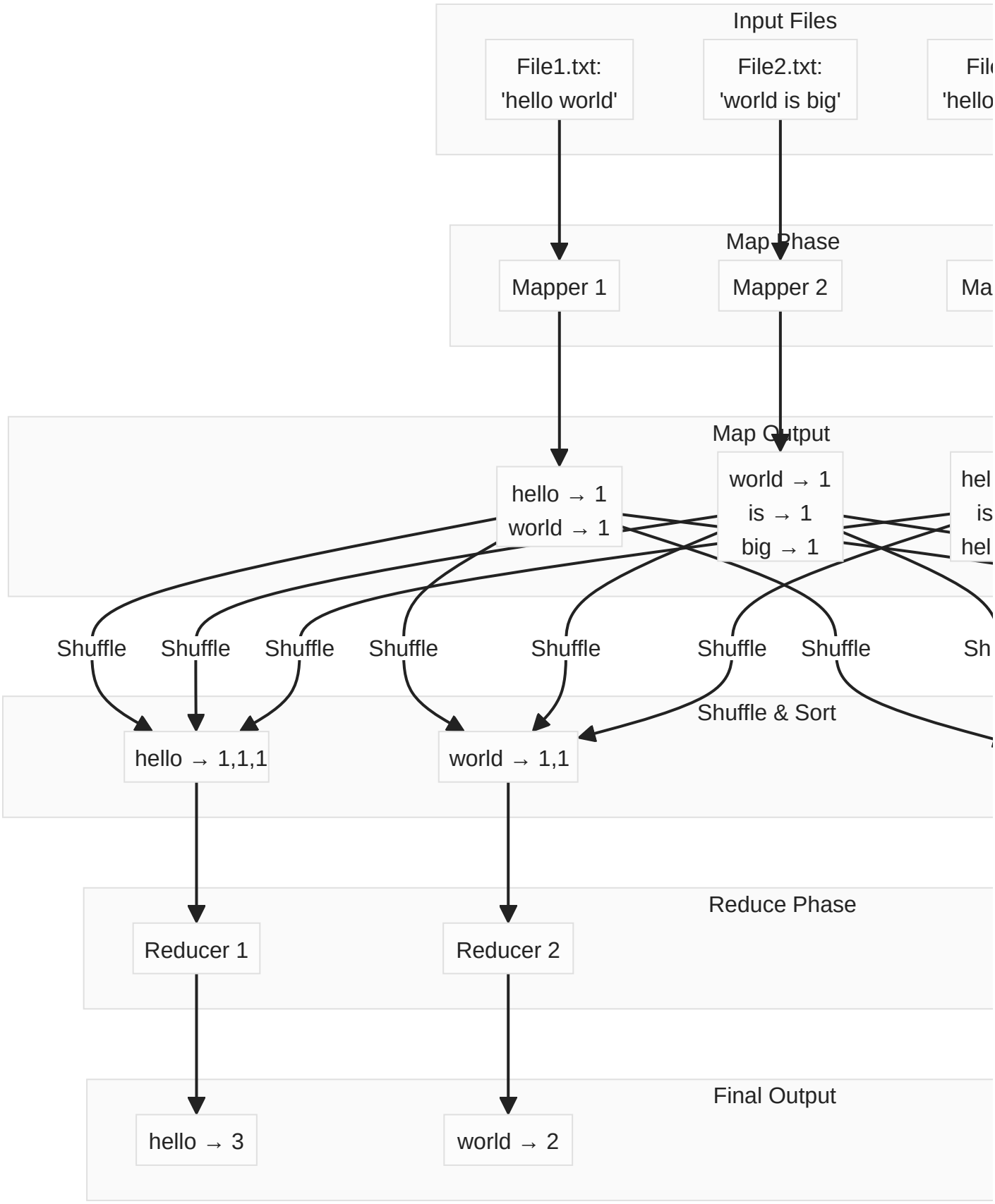
Case Study: MapReduce

Designed by Google

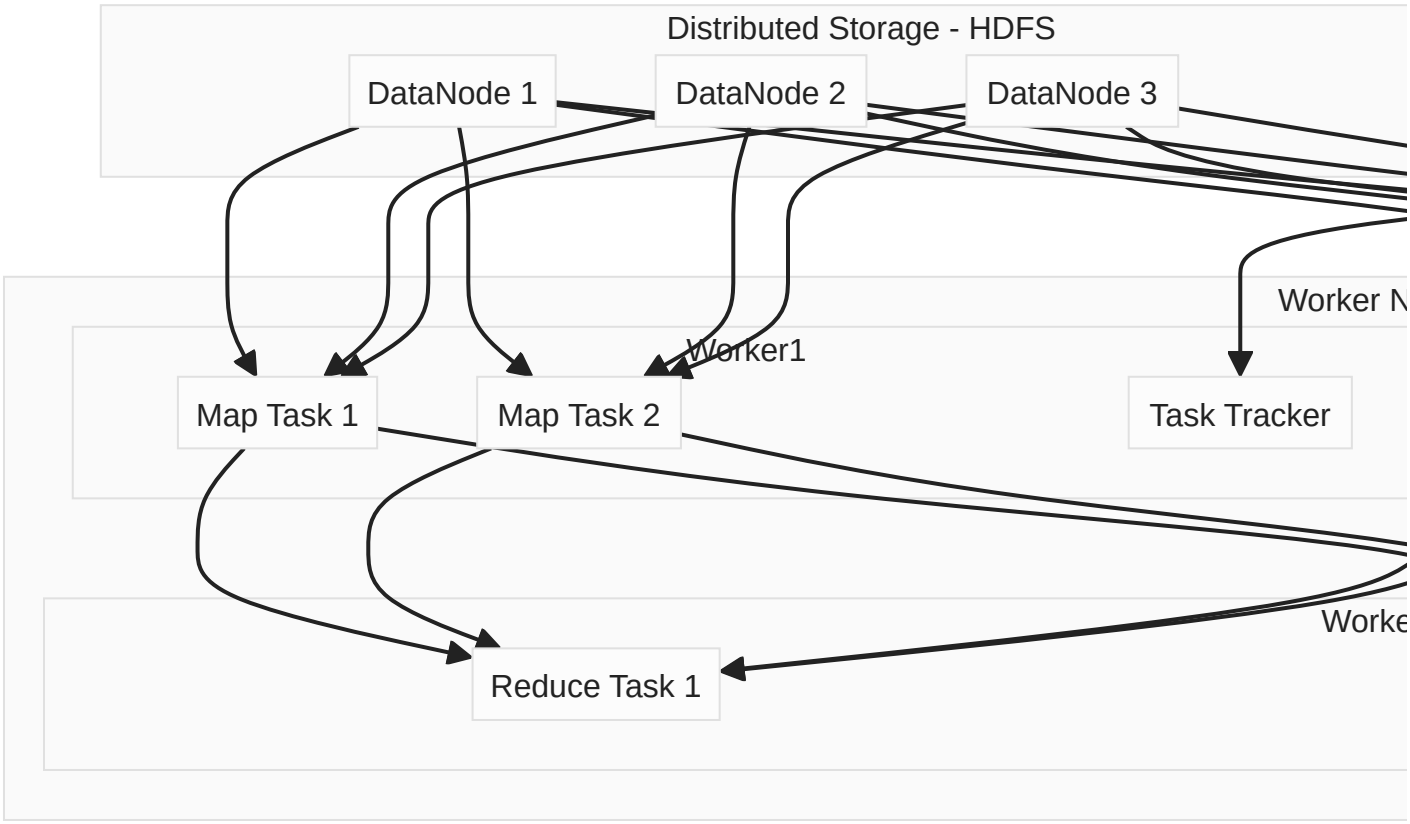
- Running huge computations at PetaByte scale (like indexing the whole fucking web lmao)
- Let engineers build and run applications without worrying about infrastructure (abstracting away using MapReduce)

```
Input: ArbitraryFileType

// Map function takes InputA and produces a key-val pair
// Each key-val pair is then processed by a Reduce function
// Map-Reduce together form a Task
```



An actual system that does map-reduce using Hadoop (distributing workers)


Distributed Storage - HDFS
DataNode 1
DataNode 2
DataNode 3
Worker N
Worker1
Map Task 1
Map Task 2
Task Tracker
Worke
Reduce Task 1