Lecture 1: Introduction

Introduction

What is the core of Distributed Systems?
A way to think about Distributed System is: A set of cooperating computers that are communicating with each other over network to get some coherent task.

We can find Distributed Systems in a lot of computer architect ure out there, such as big websites, peer to peer file sharing, big data computation (mapReduce), etc. Although very powerful, Distributed Systems are not simple to implement, so if you can possibly solve your problem in a single computer, you should do it that way.

There are some reasons why people may drive to Distributed Systems:

They need high performance - **Parallelism**; Lots of CPU, a lot of memory running in parallel;

To be able to tolerate faults - **Fault tolerance**; Having two or more computers doing the same task, one can cover another one in case of failures;

Physical reasons - **Physical** - some systems are inheretly phisically distributed (banks in different cities or countries, to manage transactions);

Achieve security goal - **Security** - to run suspecious computation on different computers, that connects with each other with some kind of narrow internet protocol, promoting **isolation**;

What makes distributed systems hard, is that with concurrent programming, there's a lot of possible consequences, like complex interactions, timing dependence. Also, besides multiple computers, you have the network component, where you can have unexpected failure patterns (partial failures, network broken or unreliable, etc).

So **concurrency** and **partial failures** are some challenges of Distributed Systems.

Infrastructure for Applications

Applications uses different types of infrastructure based on their own needs.

Most of systems provide some kinds of architecture based on **Storage**, **Communication** and **Computation**, possible to have all the three together.

The perfect scenario is if we could create such an abstraction level, where the interface between the application and the infrastructure looks and act just like a non-distributed system, but that are actually

high performant, fault tolerant distributed systems underneath - but that's something not easy to achieve.
To be able to achieve that abstraction, there are some general topics we need to understand and its important tools:

- **Implementation (how to build the system)**
    - Remote Procedure Call (RPC)
    - Threads (Harness multicore computing, structuring concurrent computation, simplifying it)
    - Concurrency control (locks)
- **Performance**
    - Scalability (2x omputers or resources gets me 2x resources/ throughput)
- **Fault Tolerance**
    - Availability (Under certain set of failures, system continue to provide service)
    - Recoverability (If something goes wrong and the service stop working, the system is able to get back to normal operation after repair)
    - Non-volatile storage
    - Replication
- **Consistency**
    - You need (not always, but for strong consistent systems) to be able to consistently perform operations across different servers or computers. For example, if you update a value in a database table, you need that this value is updated on all of the table replicas spread over your system.

MapReduce

> MapReduce is a programming model and an associated implementation for processing and generating large
> data sets. Users specify a map function that processes a
> key/value pair to generate a set of intermediate key/value
> pairs, and a reduce function that merges all intermediate
> values associated with the same intermediate key.
> Taken from: Jeffrey Dean and Sanjay Ghemawat's paper.

How does that work?

1. Input file
2. Map
Processes data into key value pairs
A function called mapper routes a series of key-value pairs inside the map stage, and serially processes very key-value pair separately. This creates zero or more output key-value pairs.

3. Shuffler/combiner

Used to sort, group and shuffle the output coming from the mapper function. This is where it moves the output from the mapper to the reducer.

4. Further data sorting and organizing

5. Reducer:

Aggregates and computes a set of result and produces a final output. It produces zero or more output key-value pairs.

6. Track

MapReduce keeps track of its task by creating a unique key

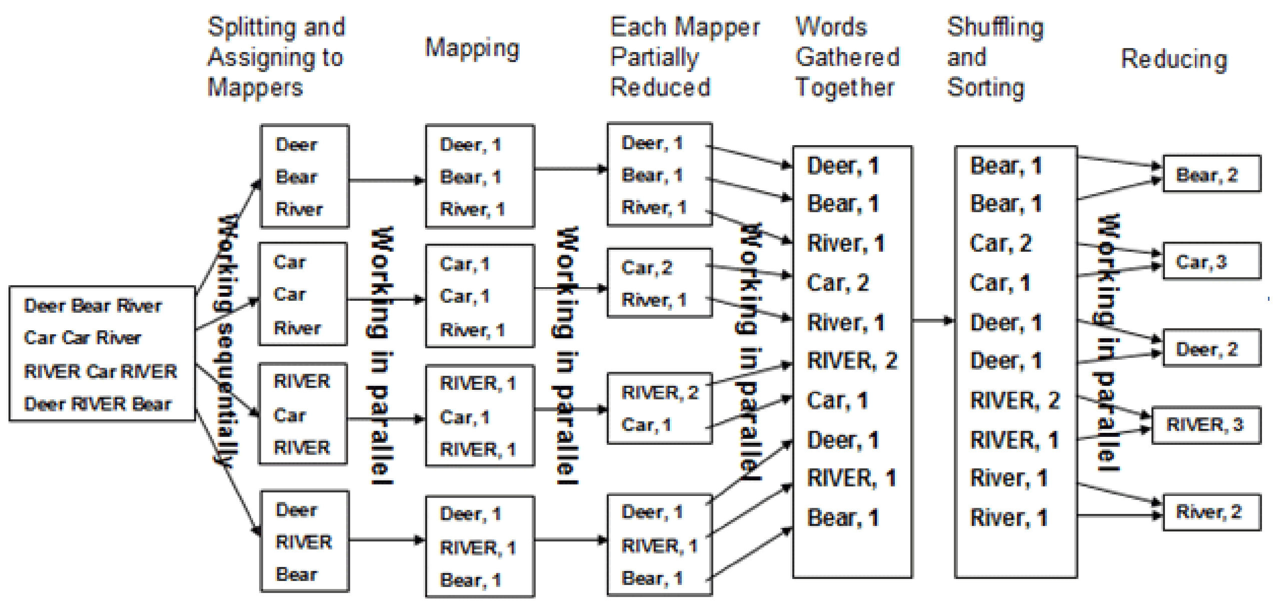
For example, considering a word count example:

Input	Maps	key value	Reduce	Reduce Output
Input 1	->	a,1 b,1	key a	a,2
Input 2	->	b,1	key b	b,2
Input 3	->	a1, c1	key c	c,1

```
Map(k,v) // key value
    Split v into words
    for each word w
        emit(w, "1")

Reduce(k, v) // key, vector of all results from map
    emit(len(v))
```

Word Count with four Mappers/Splits



In real life, there are implementations with multi-stages and iterative MapReduce jobs, like pageRank algorithm.