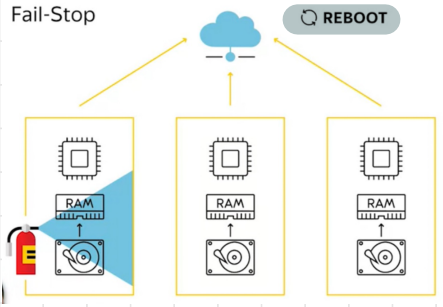
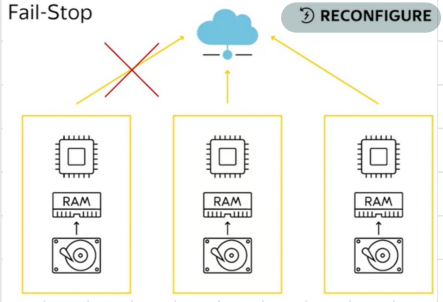
# Hadoop MapReduce:

## How to build reliable system from unreliable components.

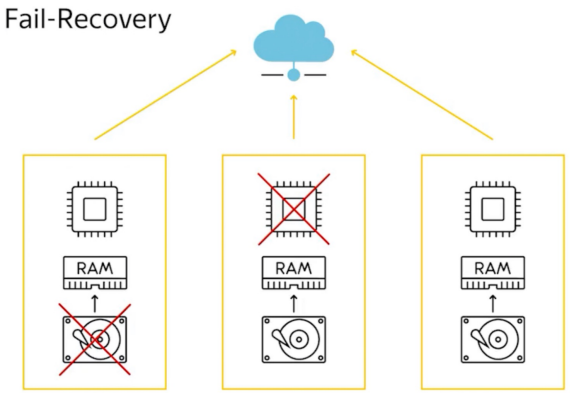
- Distributed systems are often built from unreliable components, cluster nodes can break any time because of power supply, disk damages, overheated CPUs, and so on.
- Types of Node Failure.

  1, Fail - Stop
  - means that if machines get out of service during a computation, then you have to have an external impact to bring system back to a working state.
    - Reboot: fix the node and reboot the whole system or part of.
    - Reconfigure: retire the broken machine and reconfigure the distributed system, not robust to node crashes.



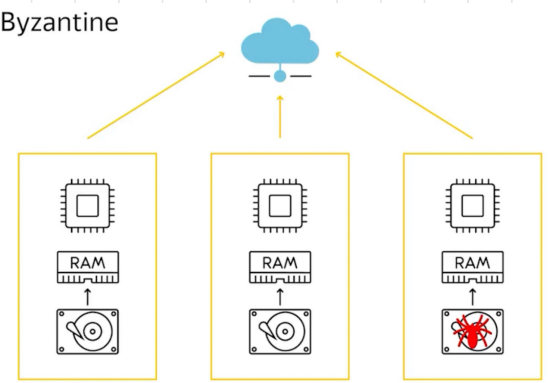



2, **Fail Recovery**.




- during computations, nodes can arbitrary crash and return back to servers.
- **doesn't influence correctness and success of computation**. no external impact necessary to reconfiguring the system.
- If hard drive was damaged, system administrator can physically change the hard drive. After reconnection, this node will be automatically picked up by a distributed system.

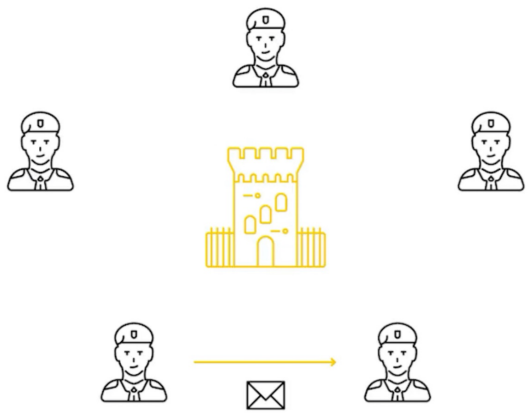
3, **Byzantine failure**.



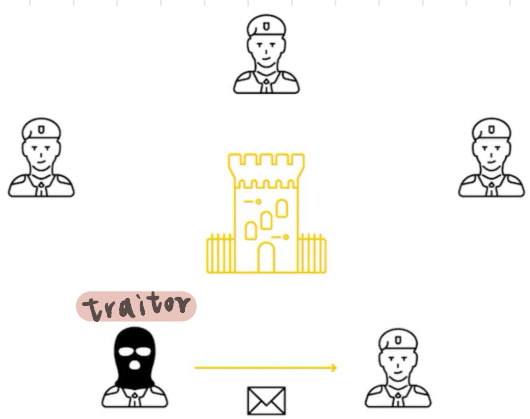

- A distributed system is robust Byzantine failure if it can work

despite some of nodes behaving out of protocal.

- **Byzantine Generals' Problem**
  - group of generals of the Byzantine army camped with their troops around an enemy city, communicating only by a messenger, the generals must agree upon a common battle plan.



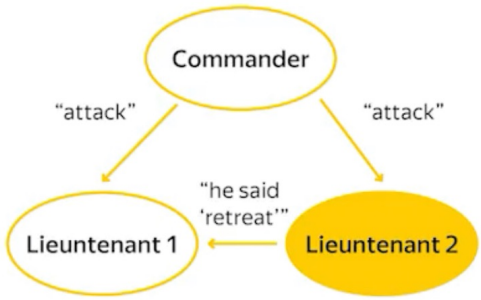
However, one or more of them may be a traitor who trying to confuse the others.



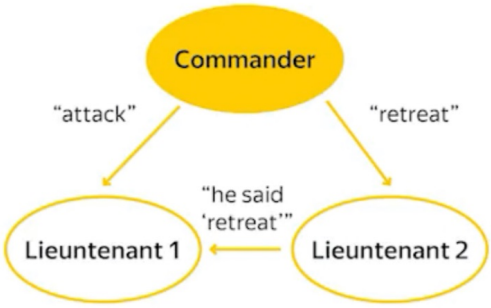

It is shown that using only unsigned messages, you have to have more than 2/3 of loyal generals.

In the case of one commanding general, two lieuntenant:

1. If second lieuntenant is traitor.

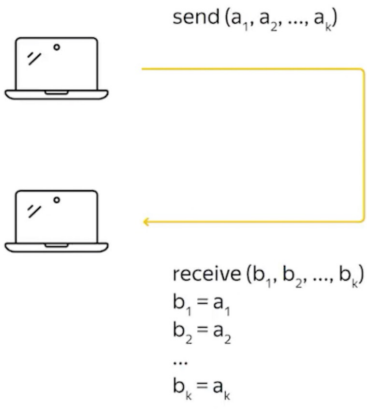



2. If commander is traitor.



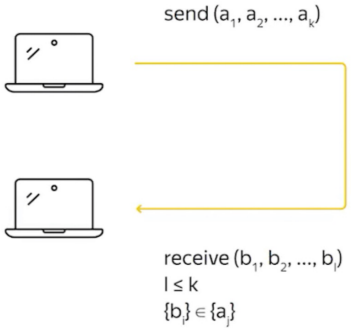

- Types of Link Failures

1. **Perfect Link**:

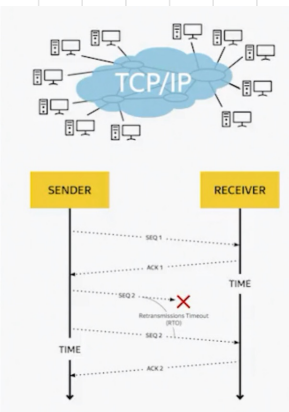



all the sent messages must be delivered and received without any modification in the same order.

## 2. Fair-Loss Link:

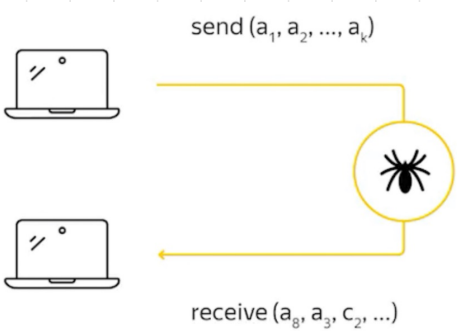



some part of the messages can be lost, but the probability of message loss doesn't depend on contents of a message.




Packet loss is a very common problem for network connection. For instance, the well-known TCP/IP protocol tries to solve this problem by re-transmitting messages if they were not received

## 3. Byzantine Link




Some messages can be filtered according to some rule, and some messages could be created out of nowhere. Nothing comes out of nowhere according to the law of conversation of energy.

- **The Two Generals Paradox.**
  - started in 1975 and then popularized by Jim Gray in 1978.
  - the first computer communication problem to be proved **unsolvable**
  - There are two generals on a campaign. They have an objective, if they simultaneously march the objective, they win. They can communicate only via messages. Unfortunately, the world is occupied by the city's defenders and there is a chance that any given method sent through the wall will be captured. The problem is to find a protocol that allows the generals to march together even. There is a simple proof that **no fixed plans protocol exists**.
    1. The first general send a message "attack at 5a.m." how does he know that the second general will receive?
    2. The second general should send an acknowledgement message comfirmed. But, how does he know that the first general will receive the acknowledgement message?

    ⇒ **continue infinitely!!**

1.


2.

- Types of clock synchronization problem.

  1, **clock skew**: time can be different on different machines.

  2, **clock drift**: different clock rate,

  - Any clock synchronization mechanism is subject to some precision,

    ⇒ **logical clock** were invented.

    - help to track happened before events, and therefore, order events to build reliable protocols.
    - logical clocks were named after his inventor, Leslie Lamport: Lamport logical clocks.

- **Synchronous System**.

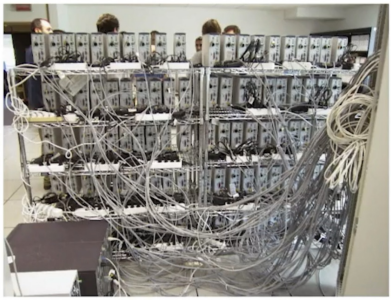
  - every messages between nodes is delivered within limited time.
  - clock drift is limited
  - each instruction exection is also limited.

- Different Distributed Systems.

  1, Fail-stop + Perfect Link + Synchronous

  - usually refevred as a **parallel computation model** and widely adopted by **supercomputers** where many process connected by a local high speed computer bus.

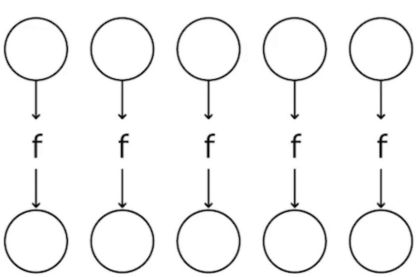

2. Fail Recovery + Fair-Loss Link + Asynchronous.




3. Byzantine-Failure + Byzantine Link + Asynchronous

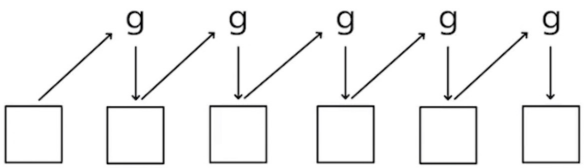
- It is a model adopted for the systems where computational components spread across the globe of unreliable and untrusted network connections.
- The common representative of this model is grid computing.

- MapReduce
  - Invented by Jeffery Dean and Sanjay Ghenawat, presented on Symposium and Operating Systems Design and Implementation in 2004.
  - Map: apply the same function to each element of your collection.

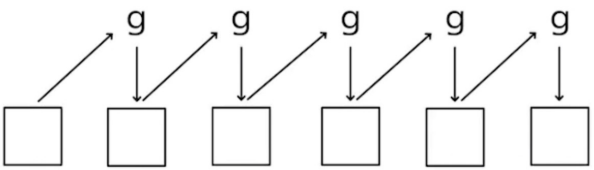



```
>>> map(lambda x: x*x, [1,2,3,4])
[1, 4, 9, 16]
```

- **Reduce** (Fold / Aggregate)




```
>>> reduce(operator.sum, [1, 4, 9, 16])
>>> reduce(operator.sum, [5, 9, 16])
>>> reduce(operator.sum, [14, 16])
30
```

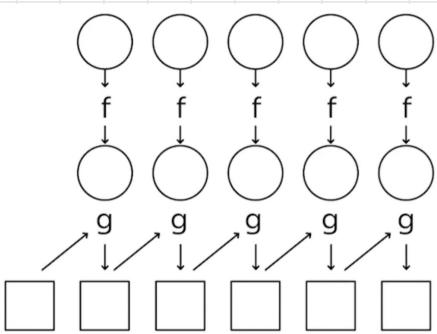



```
>>> average = lambda x, y: (x + y) / 2.
>>> reduce(average, [1, 2, 3])
2.25
>>> reduce(average, [3, 2, 1])
>>> reduce(average, [2.5, 1])
1.75
```

different result.

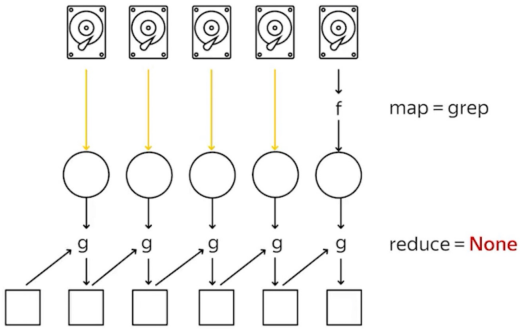
- Reduce operator cause a sequence of elements by applying the following procedure iteratively.
- reducing function sum is not associative.
- mean function is not associative: changing the order of the atom effects the result.

- **MapReduce**: combine Map and Reduce together.



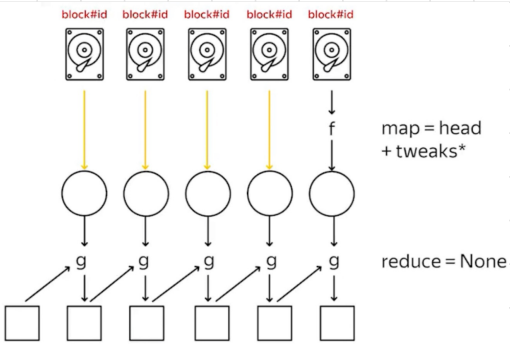

```
>>> reduce(operator.add, map(lambda x: x*x,
[1, 2, 3, 4]))
30
```

- Distributed Shell: grep

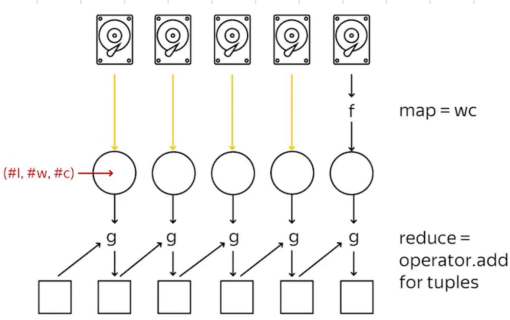



(in MapReduce application, you don't always need map or reduce function)

- Distributed Shell: head




- Distributed Shell: wc



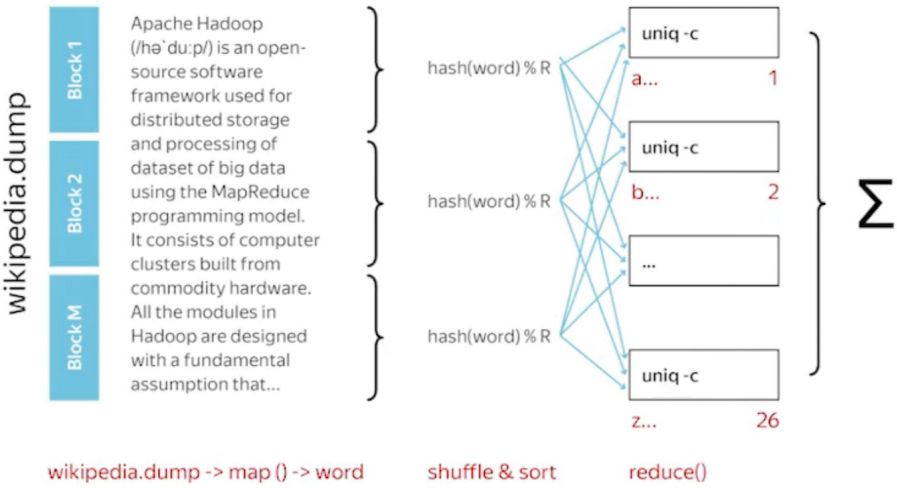

- Distributed Shell: Word Count

```
distributed: cat * | tr ' ' '\n'  | sort | uniq -c
map=sort
reduce=sort (doesn't feat in Memory / Disk)
```

⇒ introduces shuffle and sort.

Map → Shuffle & Sort → Reduce

- text is split into words.
- words are distributed to a reduce phase in the way that reduce function can be executed independently on different machines.




external sorting

- MapReduce Formal Model.
  - All input and output of map and reduce function should be a key value pair.

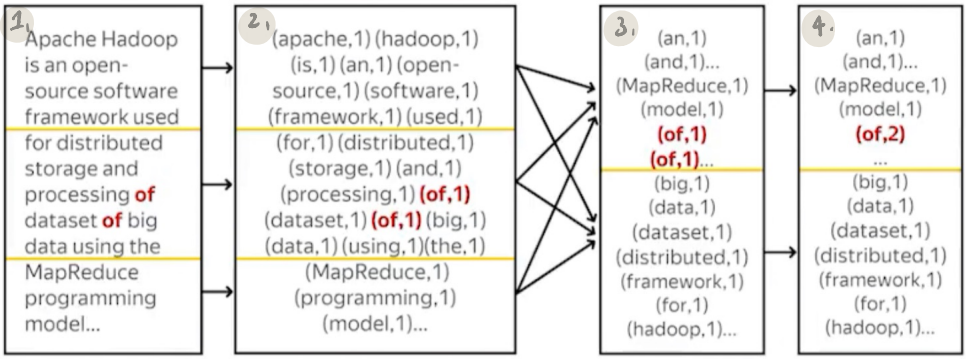
    map: (key, value) → (key, value)
    reduce: (key, value) → (key, value)

  - WordCount example

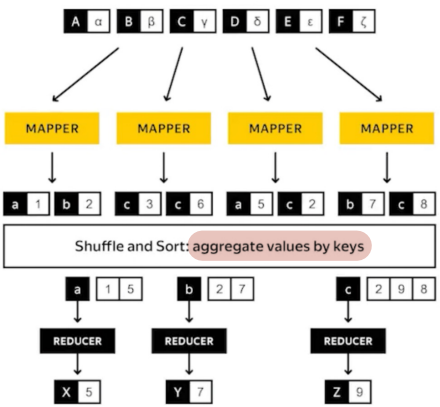
```
$ cat -n wikipedia.dump | tr ' ' '\n'|
sort | uniq -c
```

› cat **-n** wikipedia.dump: [(line_no, line), ...] 1,
› tr ' ' '\n': (-, line) —> [ (word, 1), ... ] 2,
› sort: Shuffle & Sort 3,
› uniq -c: (word, [1, ...]) —> (word, count) 4.

1, read data and get pairs with line number, line content.
2, map phase: ignore line number and split lines into words.
3, shuffle and sort phase: spread the words by the hashes.
4. reduce phase: sum up to get the answer.




Text (Big Data) Map Shuffle & Sort Reduce




› read: [(k_in, v_in), ...]

› map: (k_in, v_in) —> [(k_interm, v_interm), ...] ← intermediate key-value pairs.

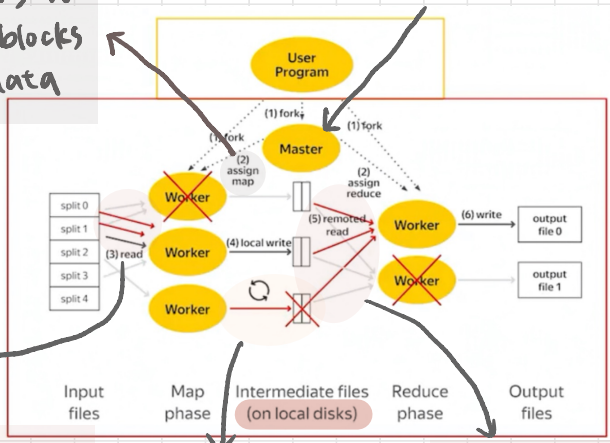
› Shuffle & Sort: sort and group by k_interm ← aggregated by intermediate keys.

› reduce: (k_interm, [(v_interm, ...)] ) —> [(k_out, v_out), ...] ← transformed by reducers, can be stored on local disks of a distributed system.

- MapReduce framework: **Fault Tolerance Model**

launch mappers to process input blocks or splits of data

control the execution



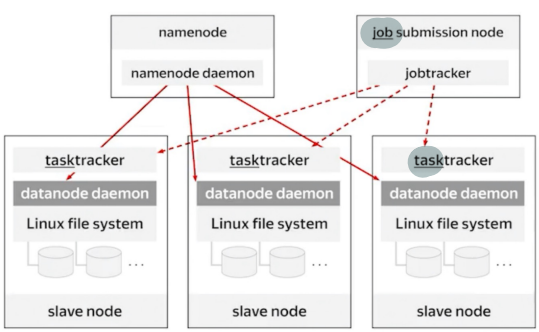

assign another worker to execute mapper against these data.

re-execute mapper on another alive worker if this data is lost

Shuffle and sort data for this particular reducer to another worker.

  - **All this complexity is hidden in MapReduce framwork** to make your life way more easier.
  - You only need to provide deterministic map and reduce functions.

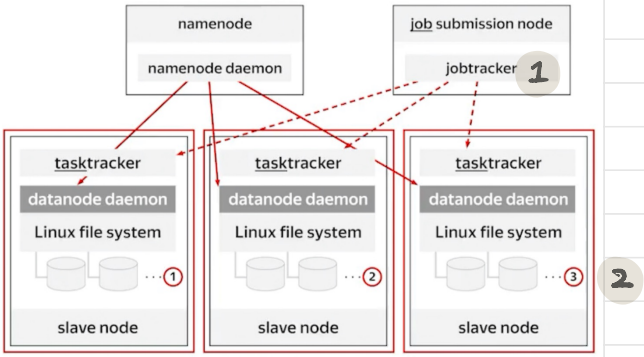
- MapReduce concepts




**jobs**: One MapReduce application is a job, is a big chunck of work.

**task**: A map or reduce function applied to some chuck of data.

- Hadoop MapReduce v1.




1. There was one global JobTracker to direct execution of MapReduce jobs. It is usually located on one high-cost and high-performance node with HDFS namenode.
2. TaskTrackers are located once per every node where you store data or where datanode daemon is working. TaskTracker spawns workers from mapper or reducer.
3. In this scenario, JobTracker is a single point of failure. That is why to reduce the load on JobTracker, some functionality of future version was delegated to other cluster nodes.