CS162
Operating Systems and Systems Programming
Lecture 23

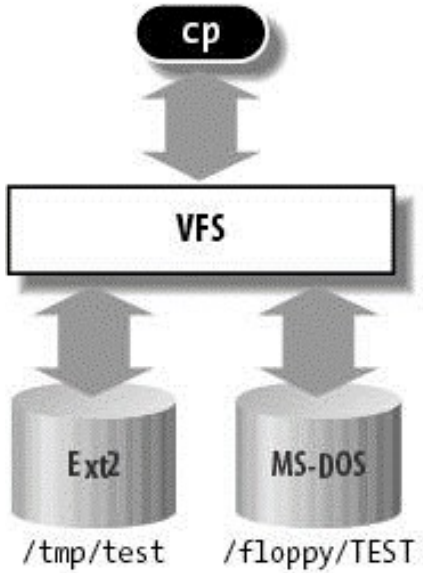
Internet & Data Processing Systems

Professor Natacha Crooks

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: Virtual Filesystem Switch




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

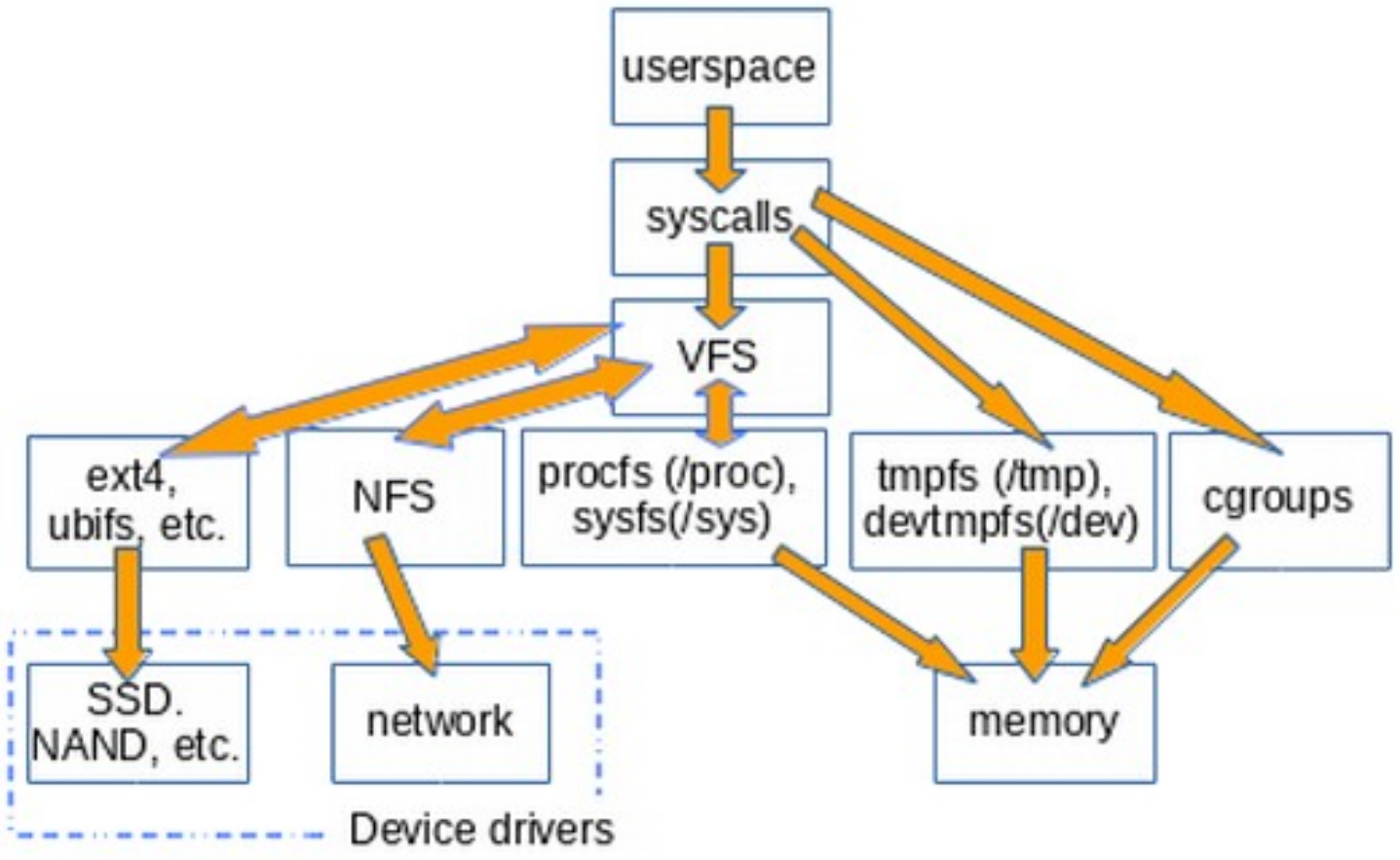
Virtual abstraction of file system

- Provides virtual superblocks, inodes, files, etc
- Compatible with a variety of local and remote file systems

VFS allows the same system call interface (the API) to be used for different types of file systems

- The API is to the VFS interface, rather than any specific type of file system

Example Linux mounting tree



Recall: Stateless Protocol

Stateless Protocol: A protocol in which all information required to service a request is included with the request

Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)

Recall: Network File System (NFS)

It's an open world!

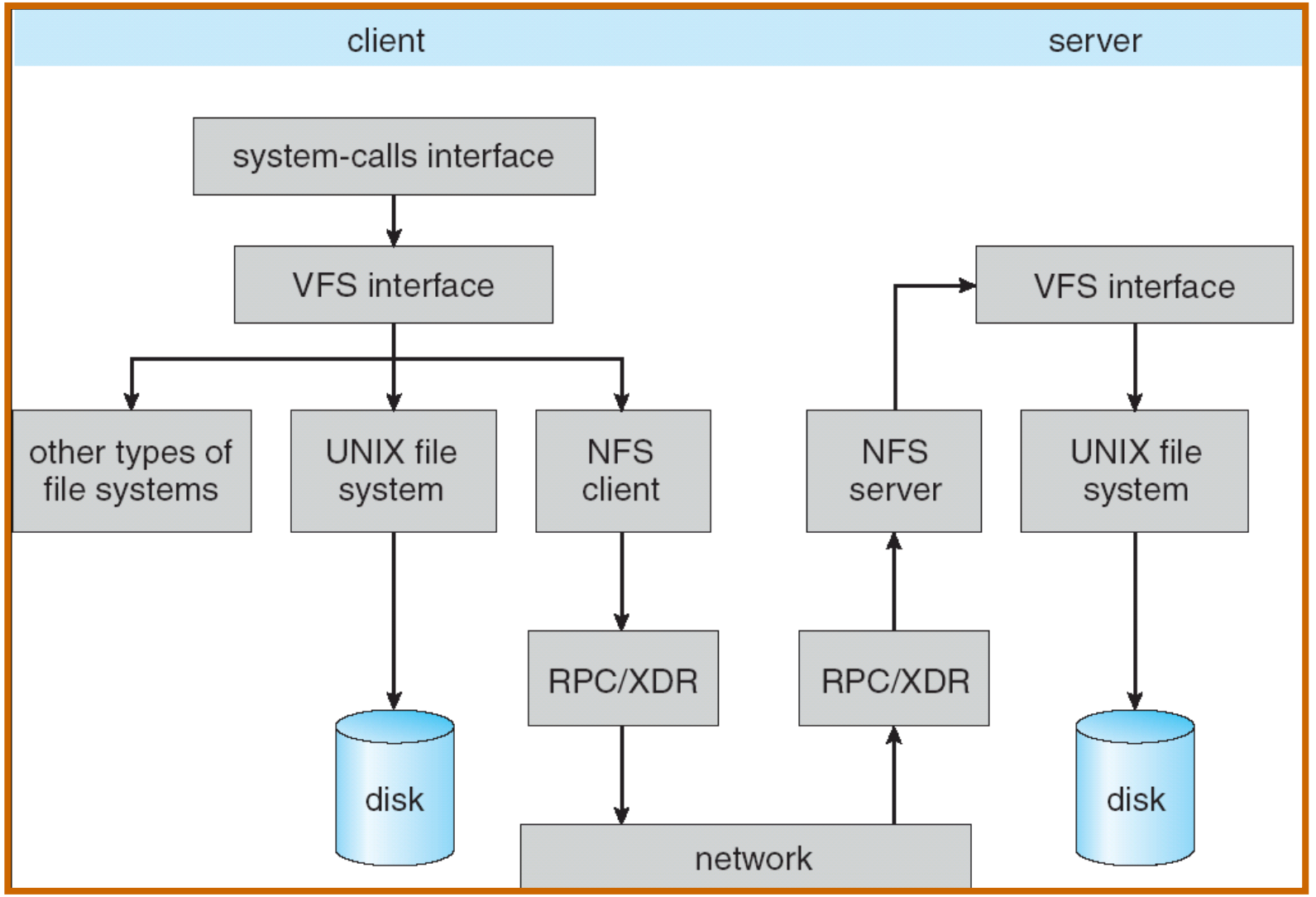
Three Layers for NFS system

UNIX file-system interface: open, read, write, close calls + file descriptors

VFS layer: distinguishes local from remote files
- » Calls the NFS protocol procedures for remote requests

NFS service layer: bottom layer of the architecture
- » Implements the NFS protocol

Recall: NFS Architecture



Topic roadmap

Distributed File Systems

Peer-To-Peer System:
The Internet

Distributed Data Processing

Coordination
(Atomic Commit and Consensus)

Case study: The Internet

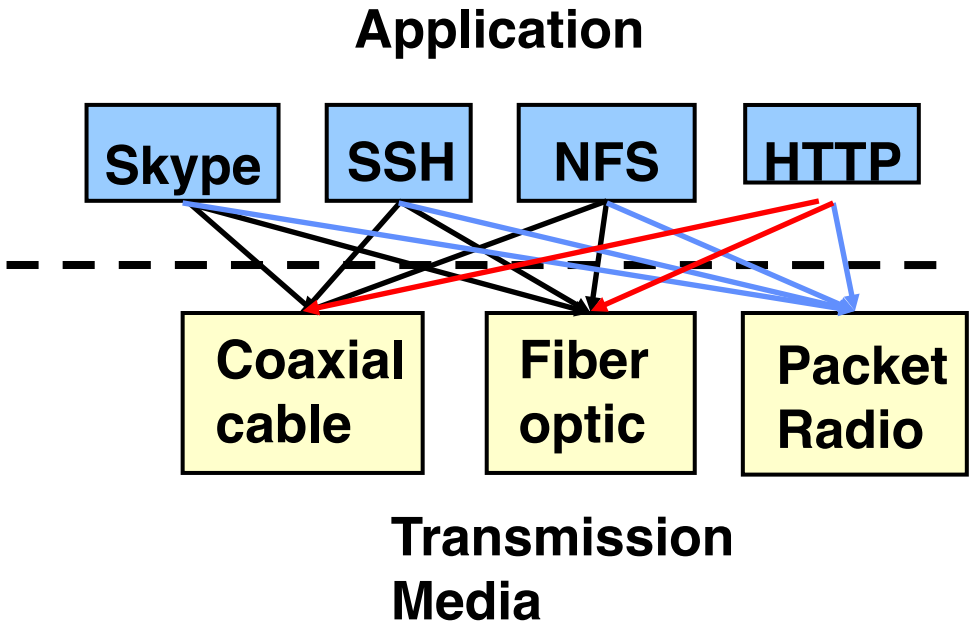
The Internet is the largest distributed system that exists!

Many different applications
- Email, web, P2P, etc.

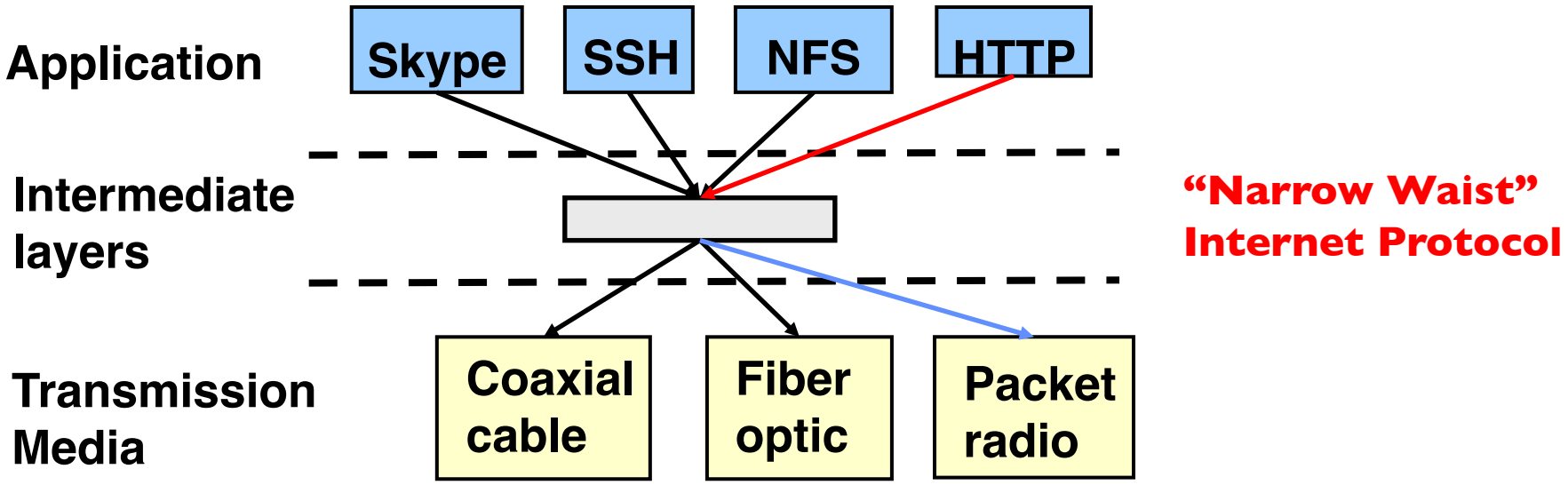
Many different operating systems and devices

Many different network styles and technologies
- Wireless, wired, optical

How do we organize this mess
- Layering & end-to-end principle



The Internet: Layers, Layers, Layers

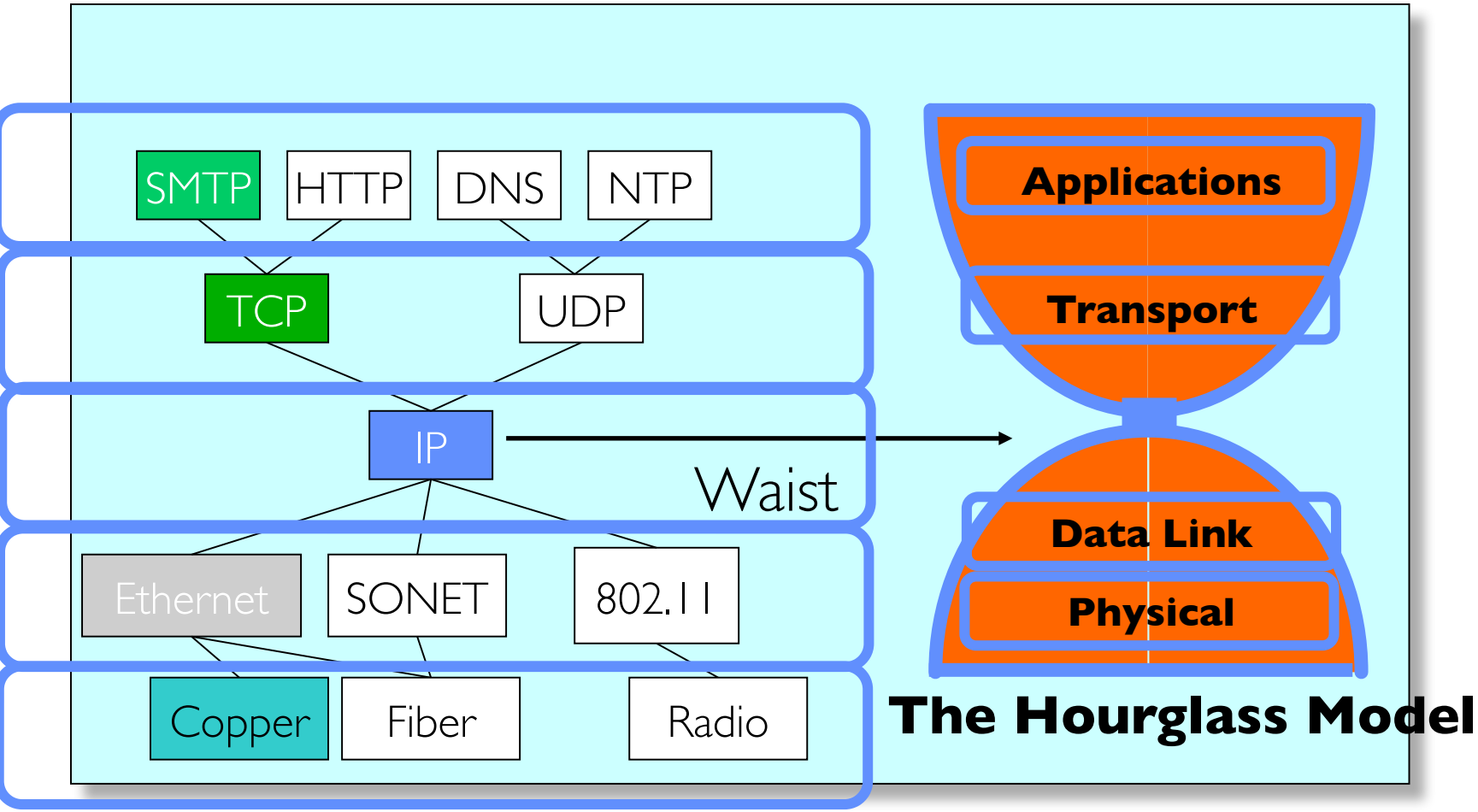


Introduce intermediate layers that provide set of abstractions for various network functionality & technologies

- A new app/media implemented only once

Goal: Reliable communication channels on which to build distributed applications

The Internet: The *hourglass*



"Narrow waist" facilitates interoperability

Layers "abstract" away hardware so that upper layers are agnostic to lower layers

=> Sound familiar?

The Internet: Implications of Hourglass

Single Internet-layer module (**IP**)**:**

Allows arbitrary networks to interoperate

- Any network technology that supports IP can exchange packets

Allows applications to function on all networks

- Applications that can run on IP can use any network

Supports simultaneous innovations above and below IP

- But changing IP itself, i.e., **IPv6**, very involved

The Internet: Drawbacks of Layering

Layer N may duplicate layer N-1 functionality

- E.g., error recovery to retransmit lost data

Layers may need same information

- E.g., timestamps, maximum transmission unit size

Layering can hurt performance

- E.g., hiding details about what is really going on

Some layers are not always cleanly separated

- Inter-layer dependencies for performance reasons
- Some dependencies in standards (header checksums)

End-To-End Argument

Hugely influential paper:

- "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)

"Sacred Text" of the Internet

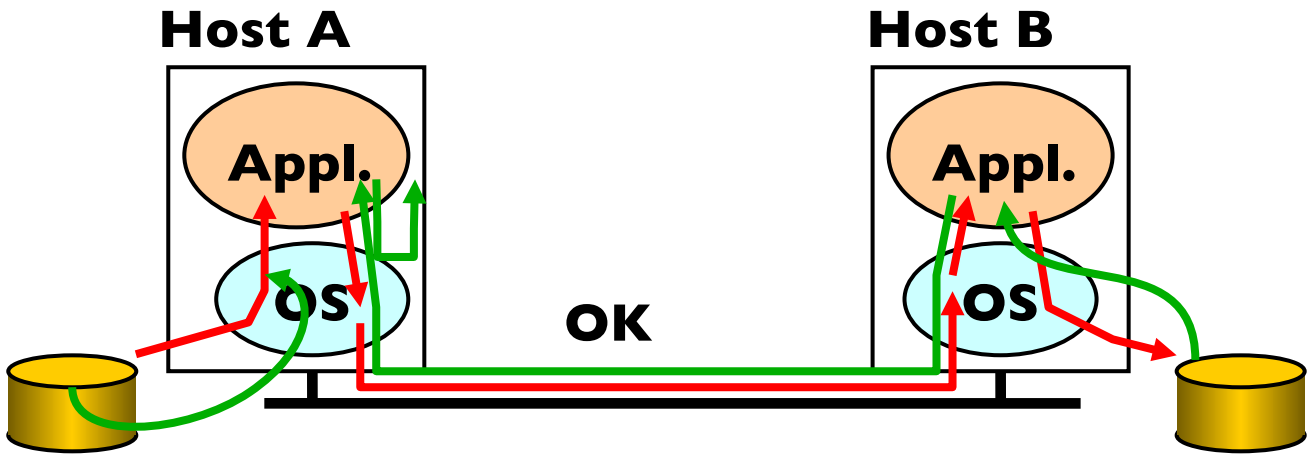
- Endless disputes about what it means
- Everyone cites it as supporting their position

Simple Message: Some types of network functionality can only be correctly implemented end-to-end

- Reliability, security, etc.

Hosts cannot rely on the network help to meet requirement, so must implement it themselves

Example: Reliable File Transfer



Solution 1: make each step reliable, and then concatenate them

Solution 2: end-to-end check and try again if necessary

Discussion

Solution 1 is incomplete

- What happens if memory is corrupted?
- Receiver has to do the check anyway!

Solution 2 is complete

- Full functionality can be entirely implemented at application layer with no need for reliability from lower layers

Is there any need to implement reliability at lower layers?

- Well, it could be more efficient

End-to-End Principle

Implementing complex functionality in the network:

- Doesn't always reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality

However, implementing in network can enhance performance in some cases

  - e.g., very lossy link

Conservative Interpretation of E2E

Don’t implement a function at the lower levels of the system unless it can be completely implemented at this level

Or: Unless you can relieve the burden from hosts, don’t bother

Moderate Interpretation

Think twice before implementing functionality in the network

If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement

But do so only if it does not impose burden on applications that do not require that functionality

This is the interpretation we are using

Case Study: Distributed Data Processing

Motivation

How can I compute the number of different words in a set of files?

Option 1: Iterate over the files one by one

Option 2: Spawn one thread per file, merge at the end

Option 3: Spawn one thread per file-chunk, merge at the end

Option 4: Spawn one thread per file-chunk on many machines, merge at the end

Distributed Word Count

Pros

Can scale almost infinitely. Not bound by processing power of a single machine. Many cheap machines usually cheaper than one big one

Cons

Building distributed application is really hard.
Must deal with networking/RPC
Must tolerate partial failures

Must deal with distributed scheduling.

Distributed Data Processing Goal

Come up with a model for breaking large computations into smaller tasks, then build a framework that distributes those tasks to workers in a cluster

Emphasis on simplicity! Non-experts should be able to use the framework.

Introduce MapReduce (Hadoop open-source version)

Motivation / History

MapReduce developed by Google; paper published in 2004

Google had large amounts of raw data:

- Crawled web pages
- Server logs
- Search data

Needed to be able to analyze that data to construct search indices, analyze website popularity, etc.

Motivation / History

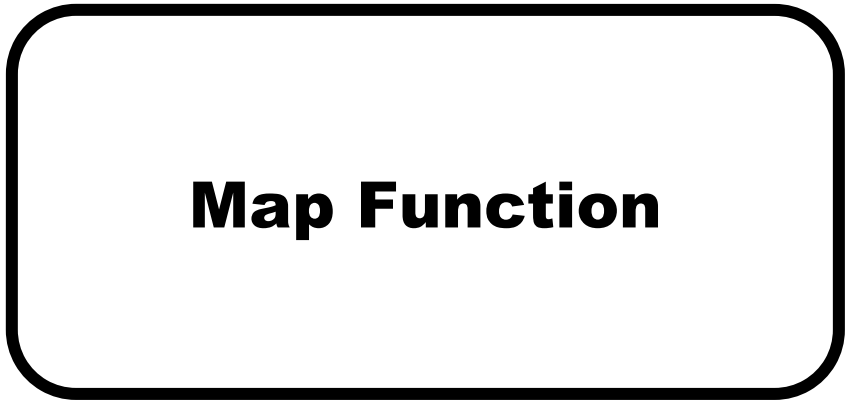
Large amounts of clusters of commodity machines

- Commodity: an “off-the-shelf” machine, ie. hardware not custom-built for Google
- Wanted to distribute workload to all these machines

Many “one-off” solutions for parallelizing workload

- Hard to maintain
- Hard to get right
- Time-consuming to implement

Map/Reduce Programming Model

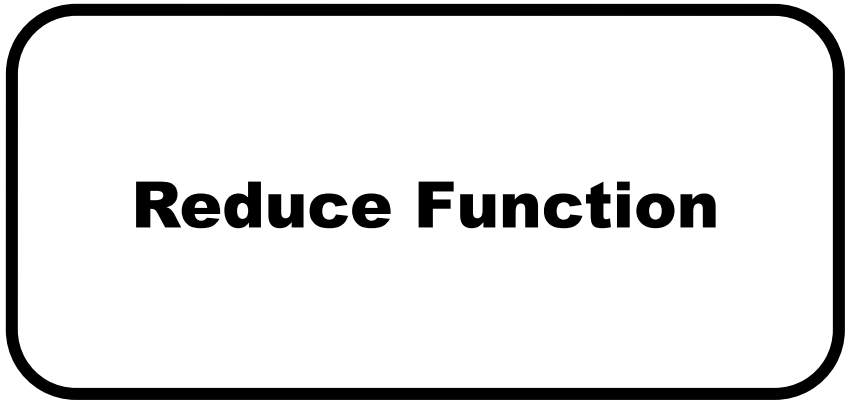


map: (k1,v1) → list(k2,v2)

Takes an input key-value pair
Outputs a list of key-value pairs

```
Map:  (k1,v1) -> (k1 + 1, v1 + 1)

Map:  (k1,v1) -> (v1, k1)
```

Map/Reduce Programming Model



reduce: (k2,list(v2)) → list(v2)

Takes in a key and a list of all values corresponding to that key
Produces a list of output values

```
reduce: (k2, list(v2) -> [sum(list(v2))]

reduce: (k2, list(v2) -> [fold(0,+,(list(v2))]
```

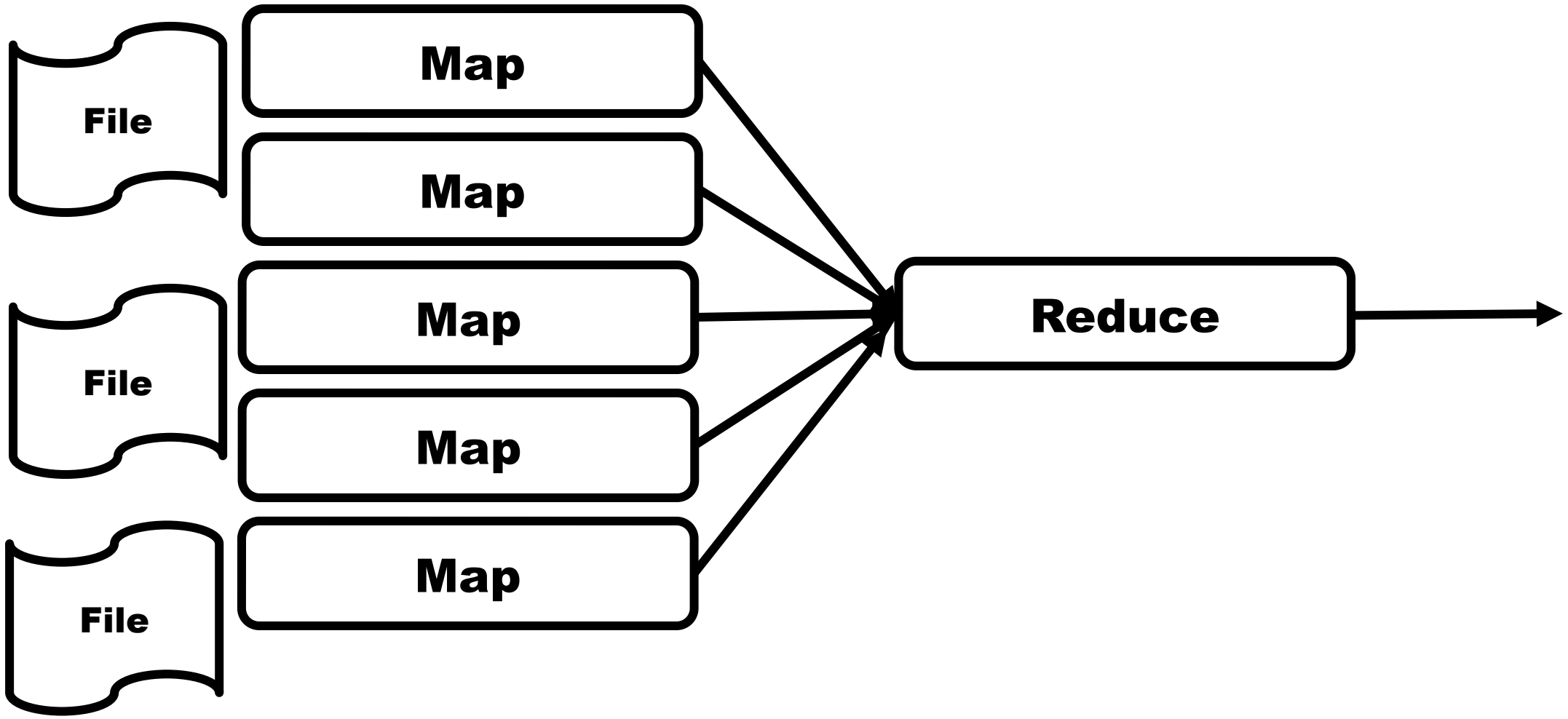
Revisiting Word Count

How can we implement word count
using only map and reduce?

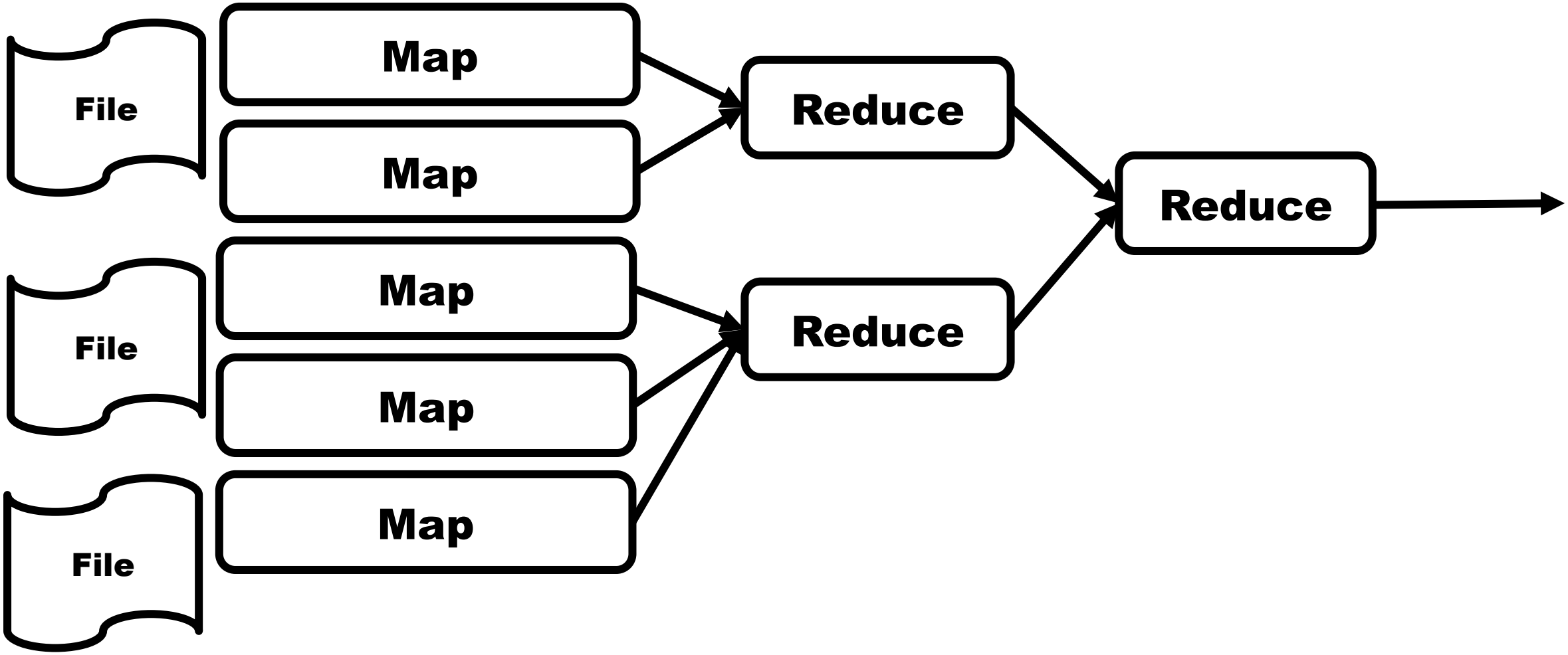
Three steps:

1) convert files into pairs of (key,value)
2) Define a map function. Apply to all files
3) Shuffle! All elements with same key
go to same reduce
4) Define a reduce function. Apply to result of the map function.

1000 ft view of Map Reduce

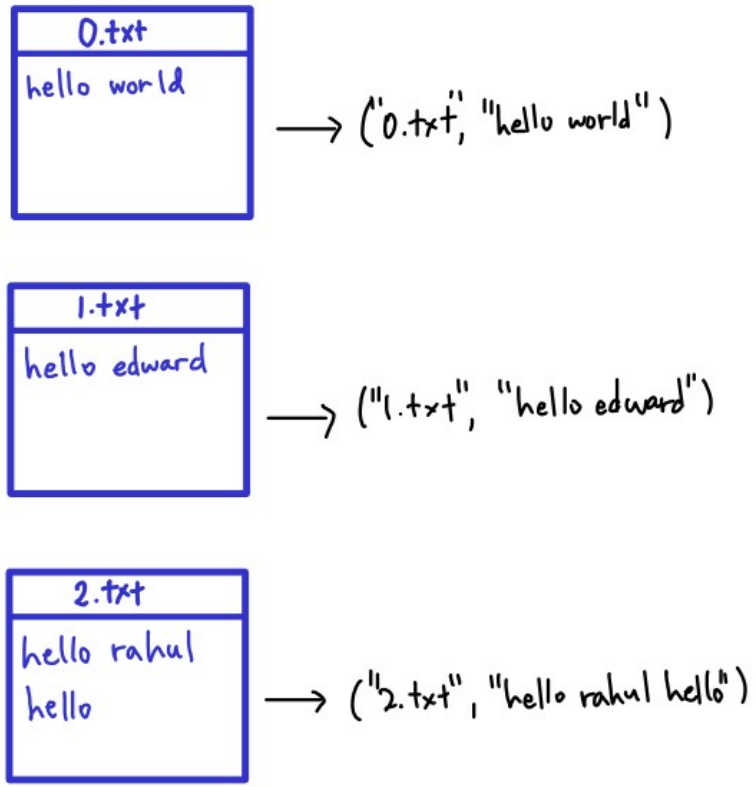


1000 ft view of Map Reduce



Word Count Map Reduce

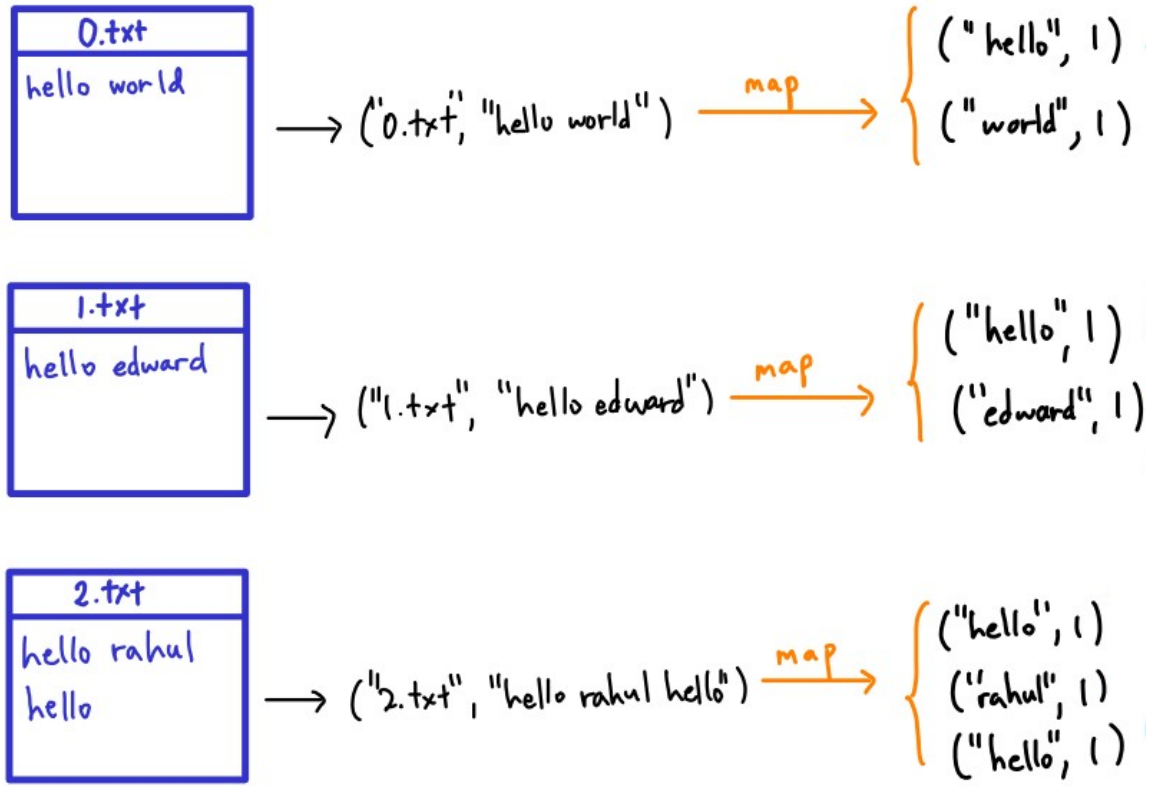
WC. Step 1: to (Key, Value)



Transform file into:
(File Name, List of words)

Map function takes (Key, List) and maps it to List (Key, Value).

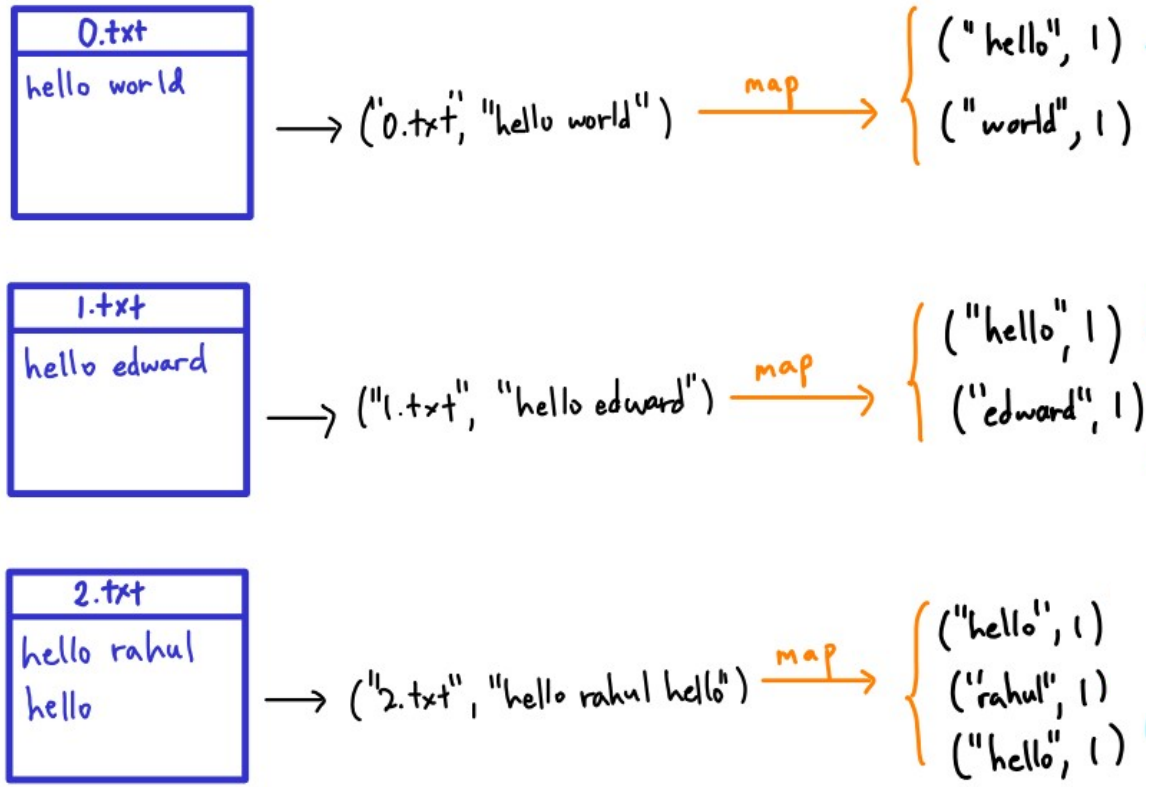
WC. Step 2: Map Function



Map function:

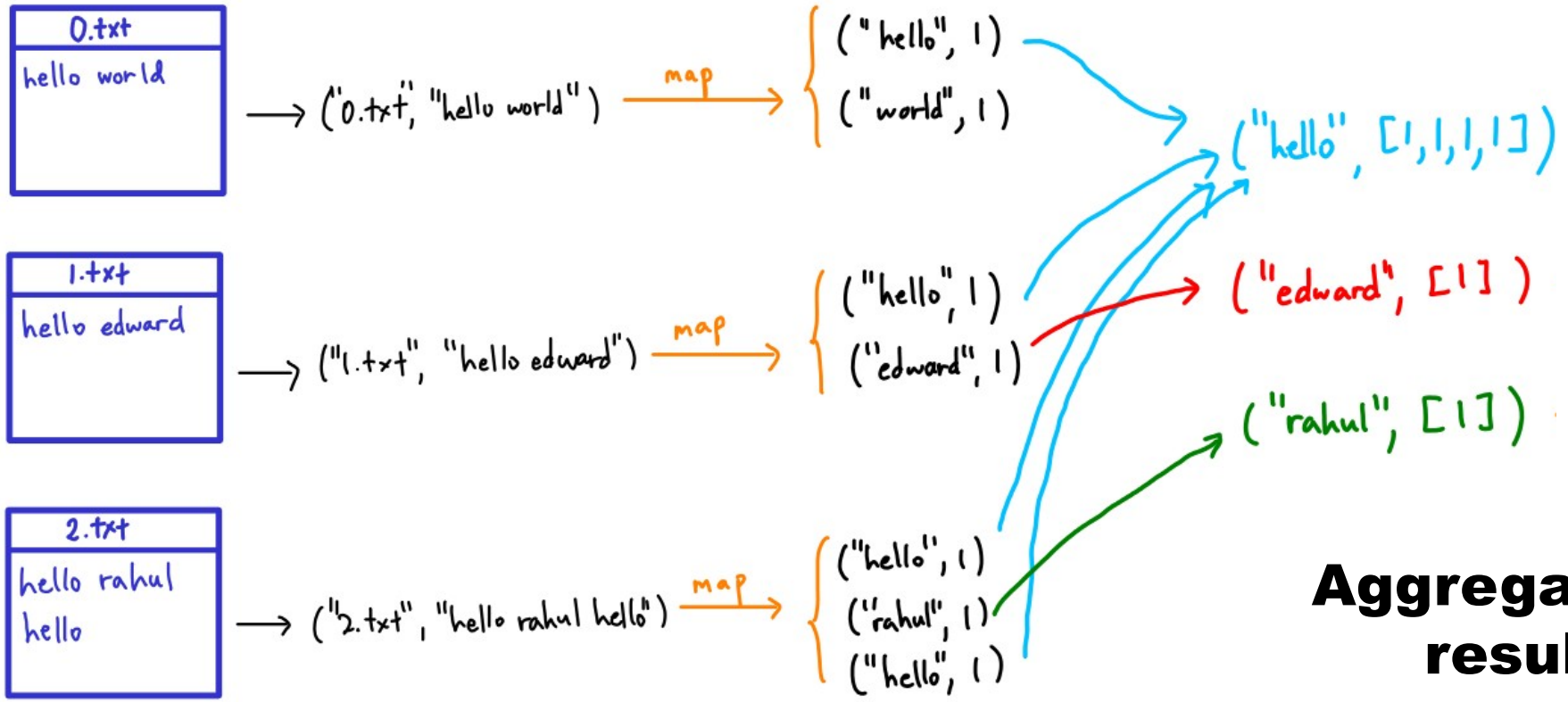
Associate each word with an associated count!

WC. Step 2: Map Function




```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

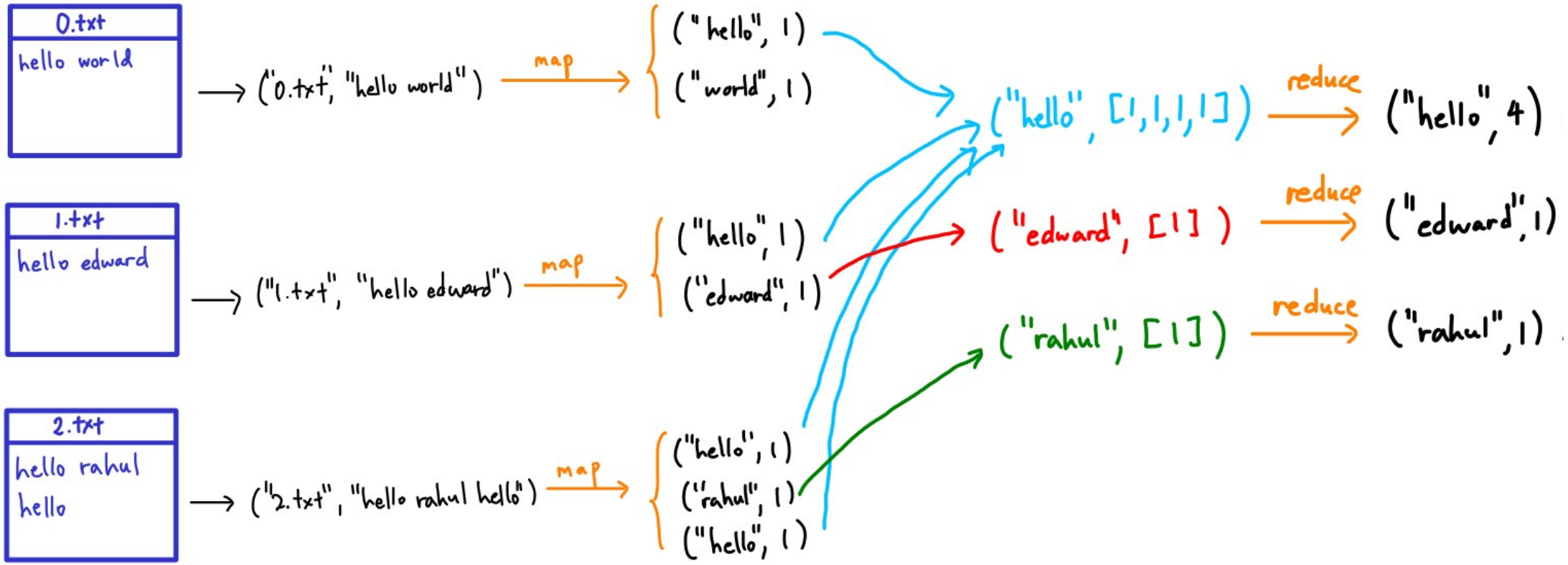
WC. Step 3: Let's do the shuffle!



Aggregate intermediate results by key

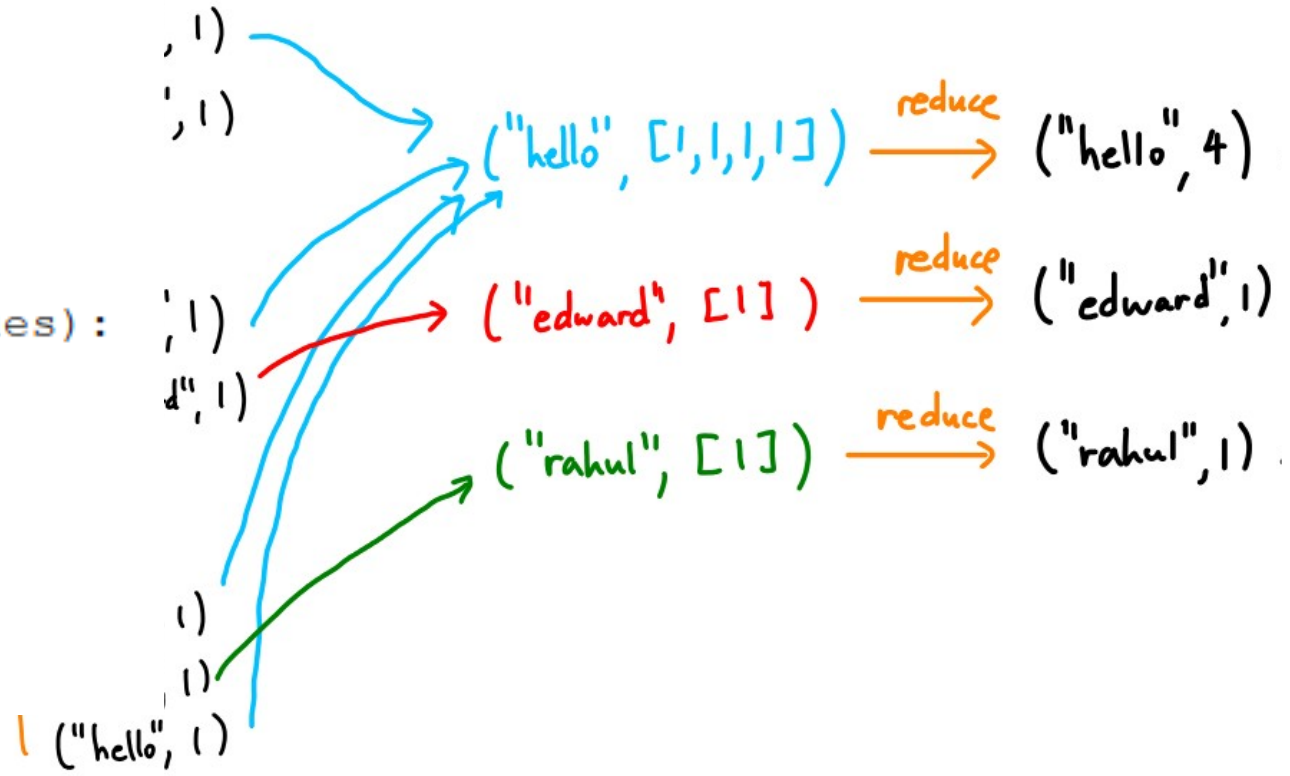
What should be the reduce function?

WC. Step 4: Reduce

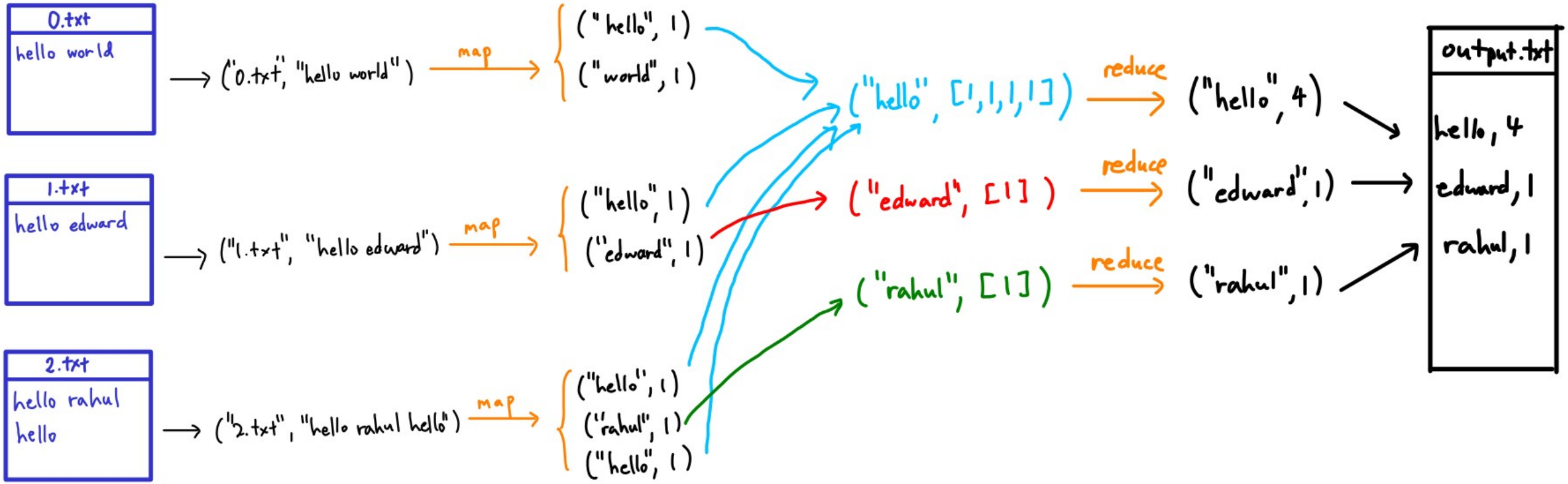


WC. Step 4: Reduce

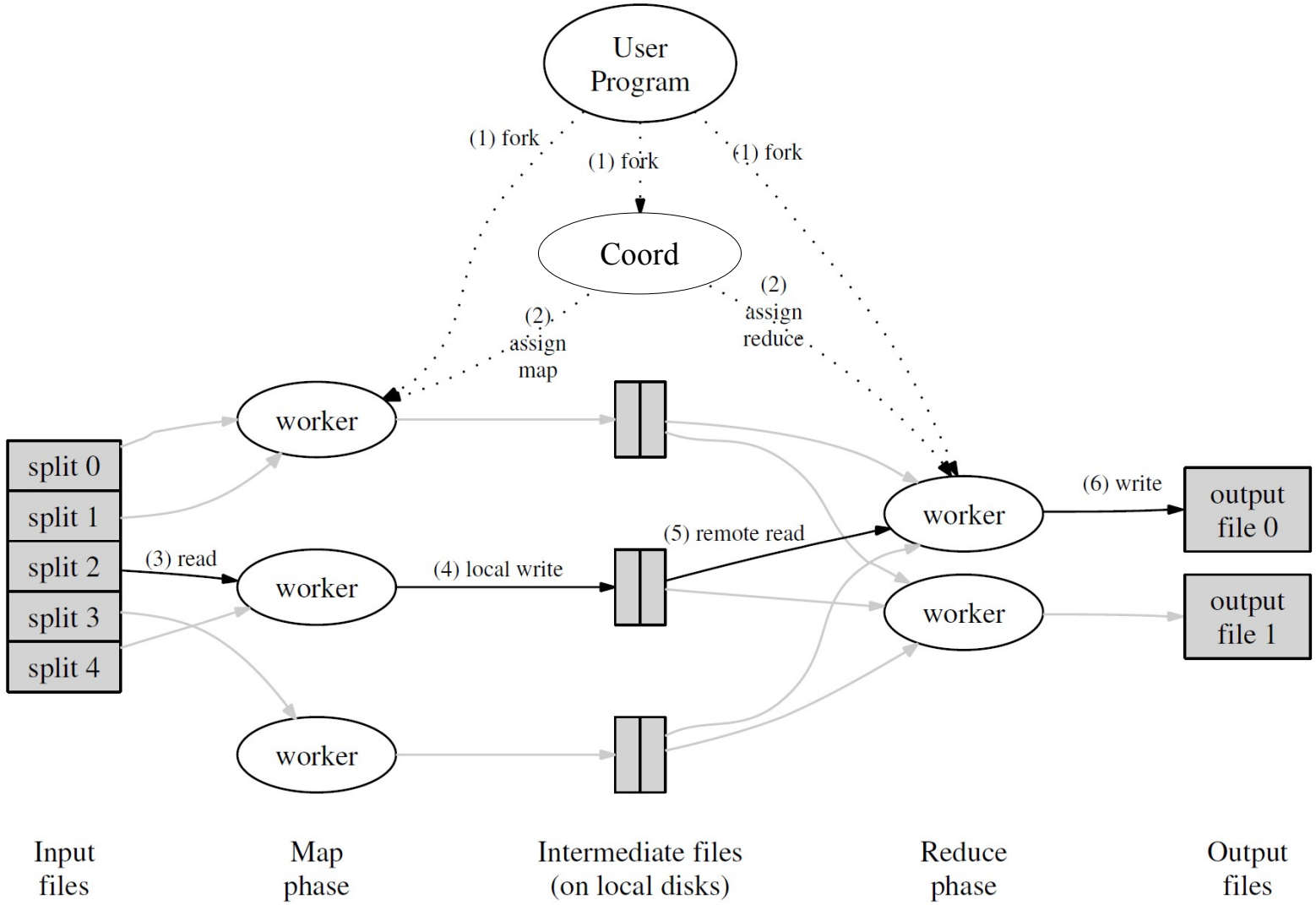
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



WC. Final Step, Generate output



Map Reduce System Architecture



Fault Tolerance

MapReduce assumes that:

Any individual machine is unlikely to crash
But large cluster of machines is likely to experience failures

MapReduce does not attempt to gracefully handle coordinator crashes.

MapReduce does handle worker failures

Idempotence is back!

When a worker fails, simply retry failed tasks!

Since failed tasks are retried, application map and reduce functions generally should be pure, deterministic functions of their arguments.

Should not depend on the current time, randomness, resources accessed over the network, etc.

Tasks that are not pure functions can be run on MapReduce, but the results may or may not be cohesive

Stragglers

With many machines, probability that one is slow increases.

Cannot begin reduce phase until map phase has completed.

Spawn “duplicate/backup” tasks to reduce probability of stragglers

Beyond MapReduce

Not all programs can be expressed in map/reduce structure.

Hard for programmers to think of computation in this way.

Disk-based and heavy network load (with shuffle)

A **lot of research in the area:**

in-memory processing (Spark), graph-processing (PowerGraph), incremental processing (Naiad), dataflow-based (Dryad), and many, many others