Parallel and Distributed Computing
CS3006 (BCS-6C/6D)
Lecture 23

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

20 April, 2023

Previous Lecture

- File Systems
- Distributed File Systems
  - Components:
    - Client module
    - Directory service
    - Flat file service
  - Examples
    - NFS (Architecture, client-server, RPCs, Virtual File System, Hierarchy, Mounting)
    - HDFS (Hadoop, YARN, MapReduce, etc.) – NameNode, DataNode
    - GFS
    - Andrew, Sprite

HDFS Security

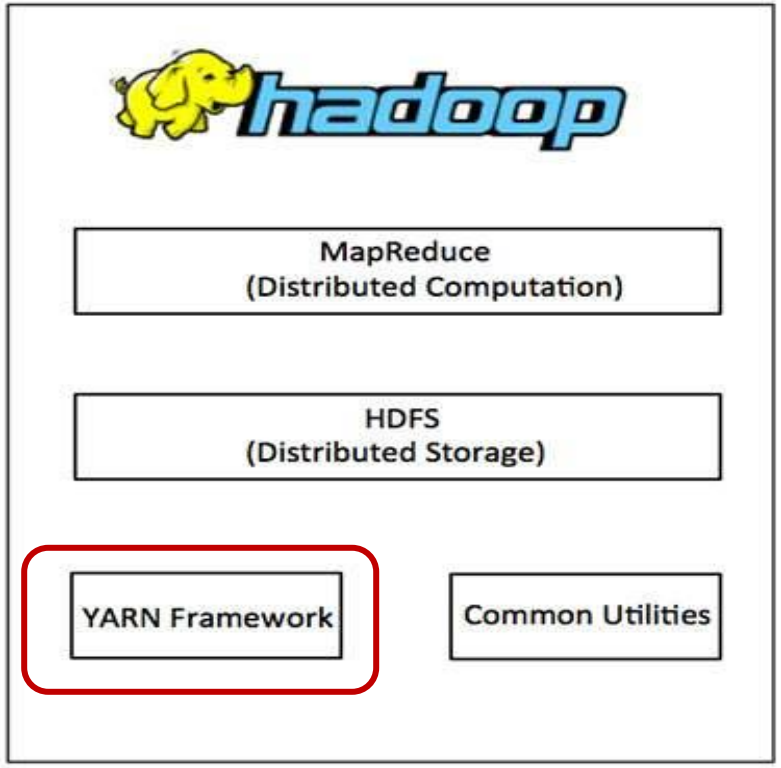
- Authentication to *Hadoop*
  - Simple – insecure way of using OS username to determine hadoop identity
  - Kerberos – authentication using kerberos ticket
  - Set by hadoop.security.authentication=simple|kerberos
- File and Directory permissions are the same as in *POSIX*
  - read (r), write (w), and execute (x) permissions
  - also has an owner, group and mode
  - enabled by default (dfs.permissions.enabled=true)
- ***ACL***s are used for implementing permissions that differ from the natural hierarchy of users and groups
  - enabled by dfs.namenode.acls.enabled=true

Interfaces to HDFS

- Java API (`DistributedFileSystem`)
- C wrapper (`libhdfs`)
- HTTP protocol
- WebDAV protocol
- Shell Commands

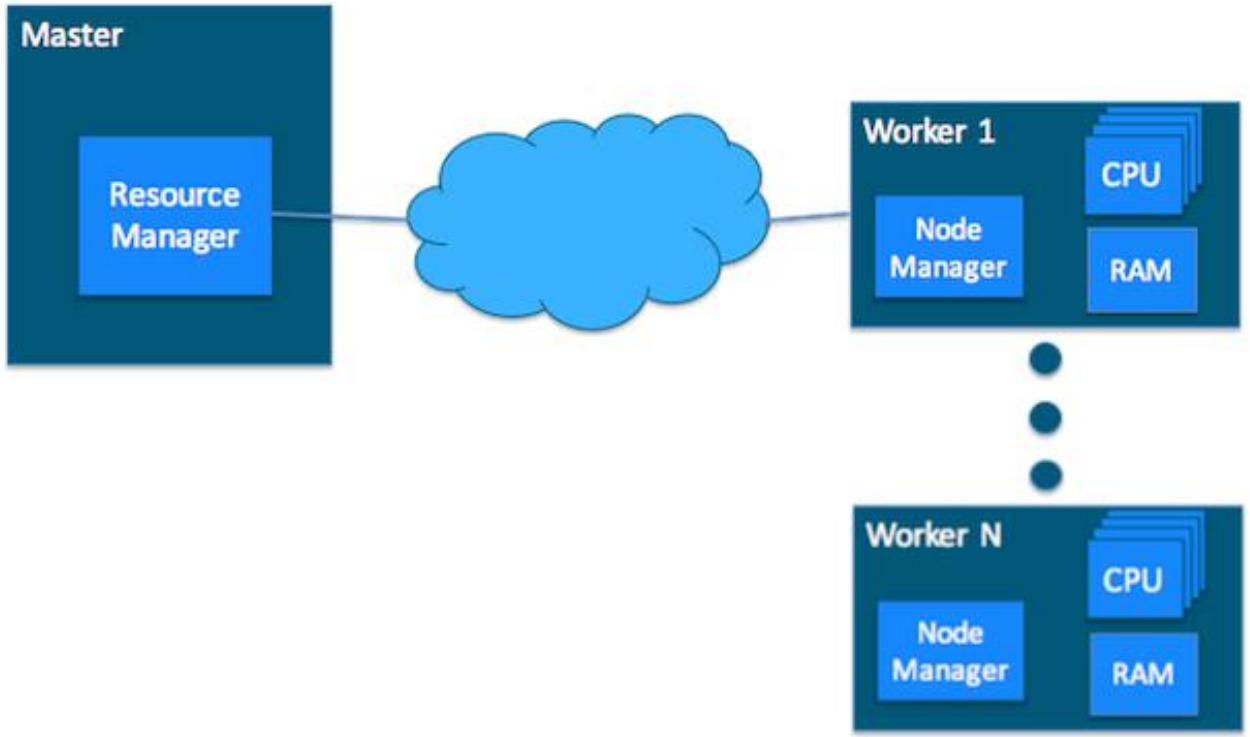
Yarn

- YARN is the prerequisite for Enterprise Hadoop
  - providing resource management and a central platform to deliver consistent operations, security, and data governance tools across Hadoop clusters.



YARN Cluster Basics

- In a *YARN cluster*, there are two types of hosts:
  - The *ResourceManager* is the master daemon that communicates with the client, tracks resources on the cluster, and orchestrates work by assigning tasks to *NodeManagers*.
  - A *NodeManager* is a worker daemon that launches and tracks processes spawned on worker hosts.



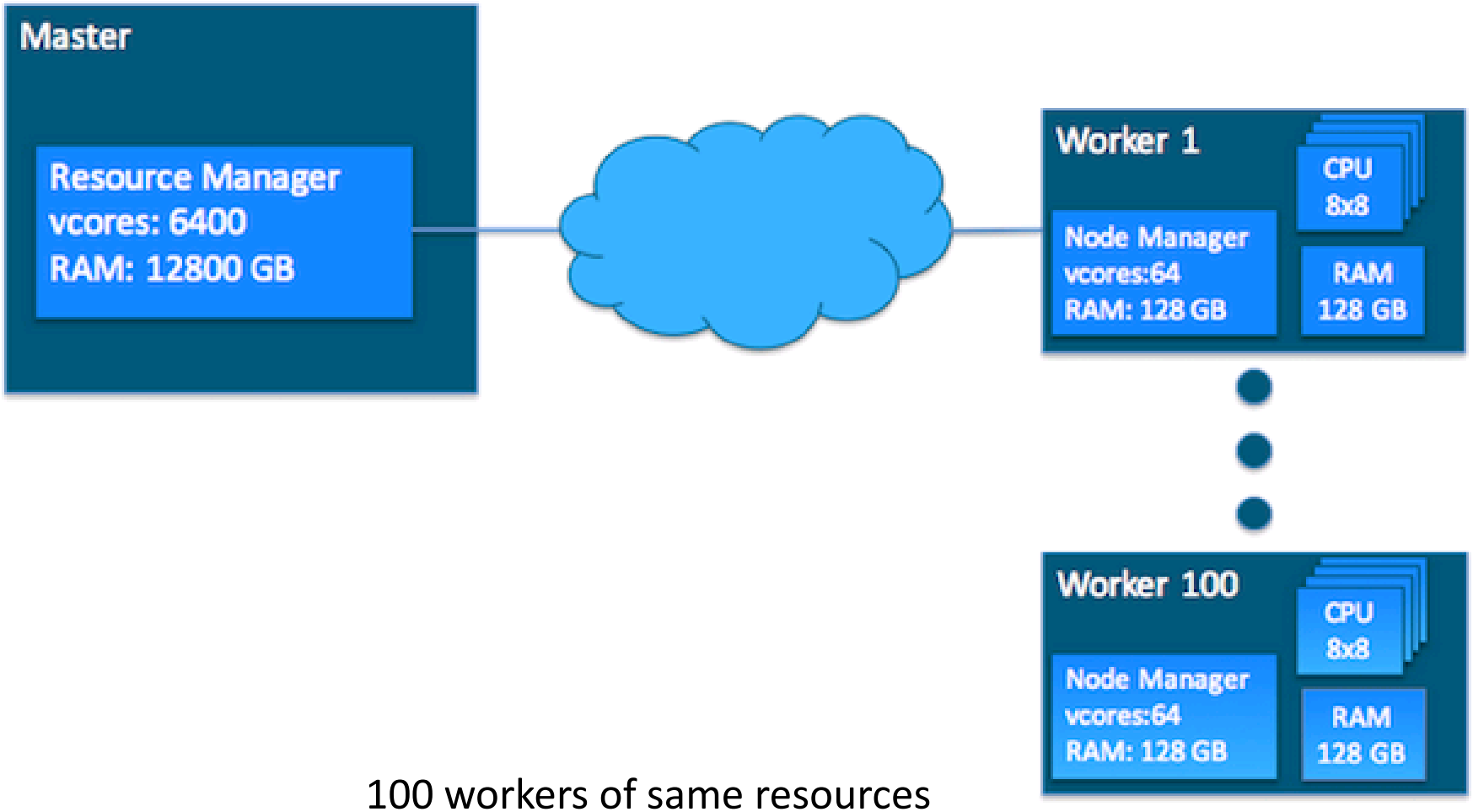
Yarn Resource Monitoring (i)

- YARN currently defines two resources:
  - *v-cores*
  - *Memory*

- Each *NodeManager* tracks
  - its own local resources and
  - communicates its resource configuration to the ResourceManager

- The *ResourceManager* keeps
  - a running total of the cluster's available resources.

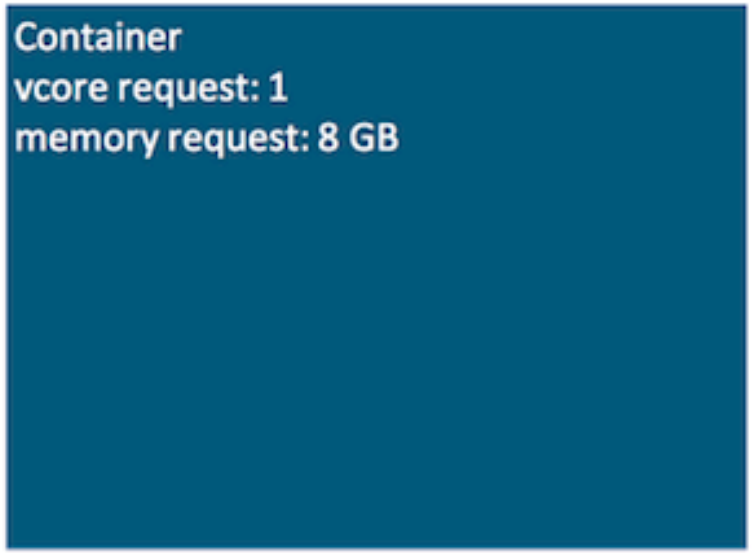
Yarn Resource Monitoring (ii)



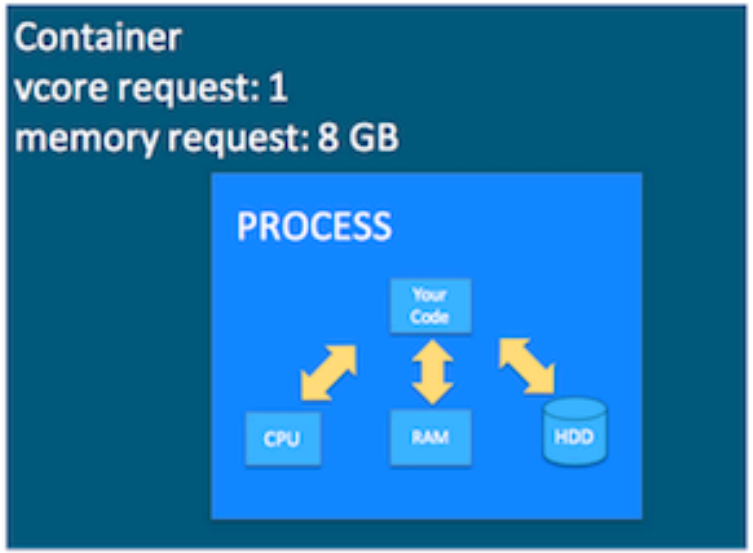
100 workers of same resources

Yarn Container

- Containers
  - a request to hold resources on the YARN cluster.
  - a *container hold request* consists of vcore and memory



Container as a hold



The task running as a process inside a container

Yarn Application and ApplicationMaster

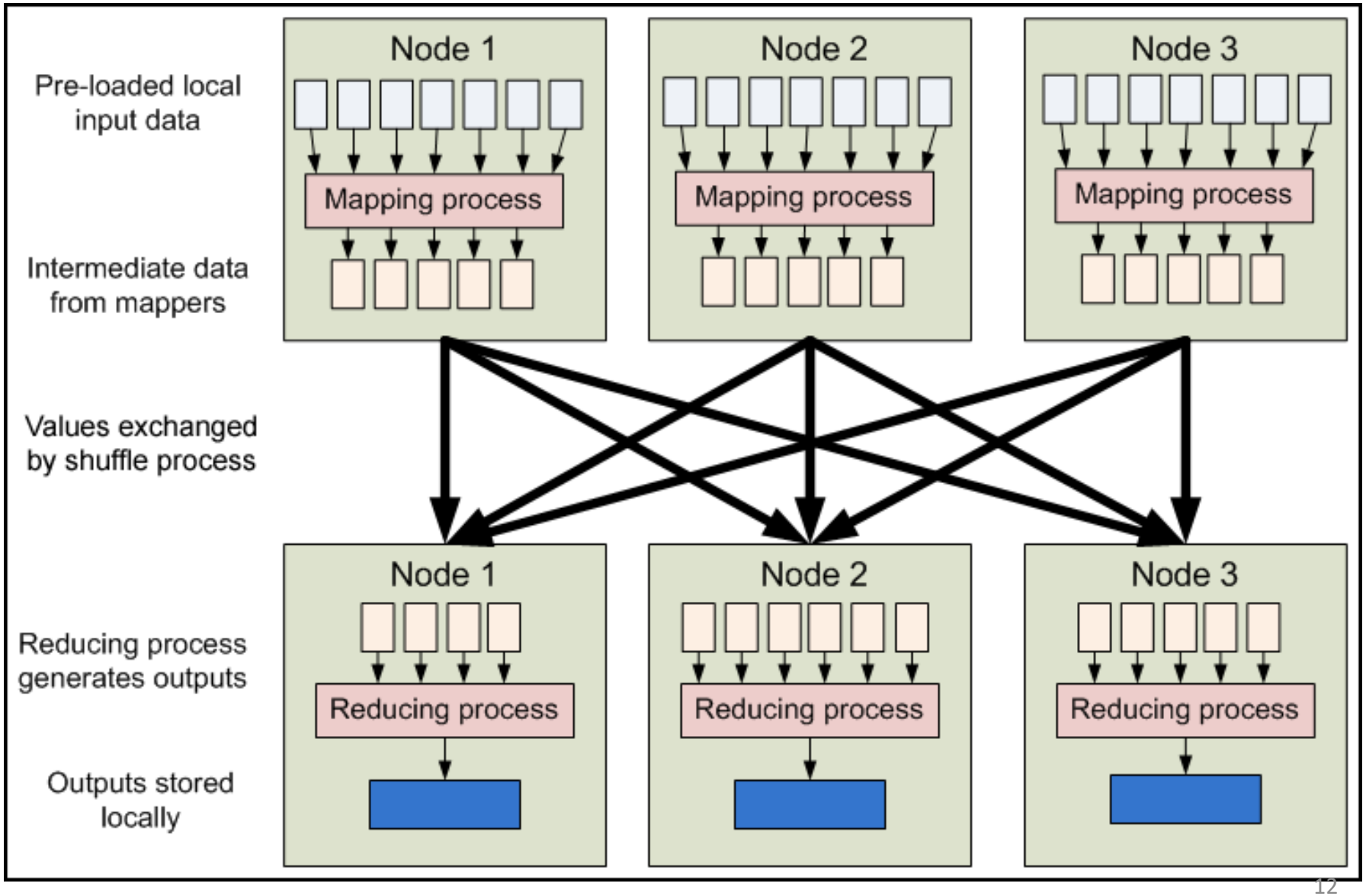
- Yarn application
  - It is a YARN client program that is made up of one or more tasks.
  - Example: MapReduce Application

- ApplicationMaster
  - It helps coordinate tasks on the YARN cluster for each running application.
  - It is the first process run after the application starts.

MapReduce - What?

- MapReduce is a programming model for efficient distributed computing
- It works like a Unix pipeline
  - cat input | grep | sort | uniq -c | cat > output
  - **Input | Map |** Shuffle & Sort **| Reduce | Output**
- Efficiency from
  - Streaming through data, reducing seeks
  - Pipelining
- A good fit for a lot of applications
  - Log processing
  - Web index building

MapReduce - Dataflow



MapReduce - Features

- Fine grained Map and Reduce tasks
  - Improved load balancing
  - Faster recovery from failed tasks

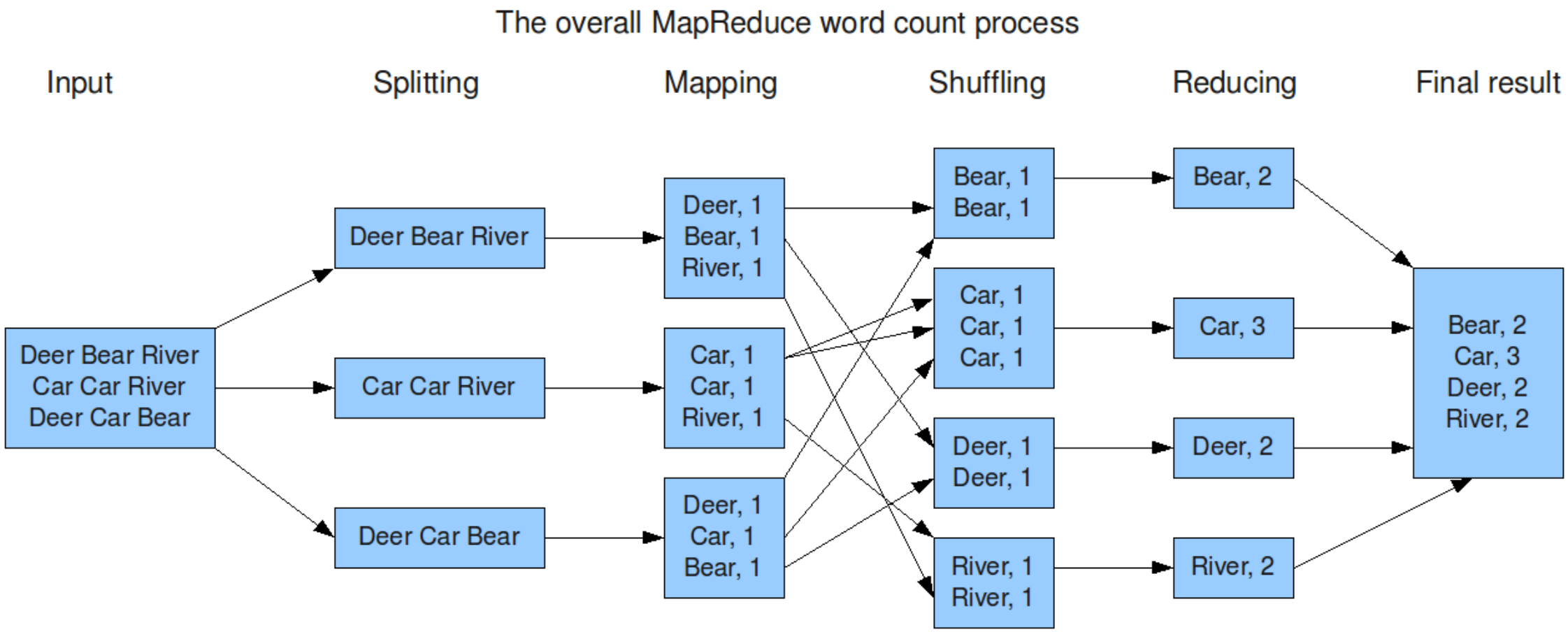
- Automatic re-execution on failure
  - In a large cluster, some nodes are always slow or flaky
  - Framework re-executes failed tasks

- Locality optimizations
  - With large data, bandwidth to data is a problem
  - Map-Reduce + HDFS is a very effective solution
  - Map-Reduce queries HDFS for locations of input data
  - Map tasks are scheduled close to the inputs when possible

Word Count Example

- Mapper
  - Input: value: lines of text of input
  - Output: key: word, value: 1
- Reducer
  - Input: key: word, value: set of counts
  - Output: key: word, value: sum
- Launching program
  - Defines this job
  - Submits job to cluster

Word Count Dataflow



Hadoop Related Subprojects

- Pig
  - High-level language for data analysis
- HBase
  - Table storage for semi-structured data
- Zookeeper
  - Coordinating distributed applications
- Hive
  - SQL-like Query language and Metastore
- Mahout
  - Machine learning

MapReduce – Walk-through

How can we parallelize data processing across many machines?

MapReduce - Parallelizing Programs

- ***Task***: we want to count the frequency of words in a document
- ***Possible Approach***: program that reads document and builds a
  - word → frequency map

How can we parallelize this?

Idea: split document into pieces, count words in each piece concurrently

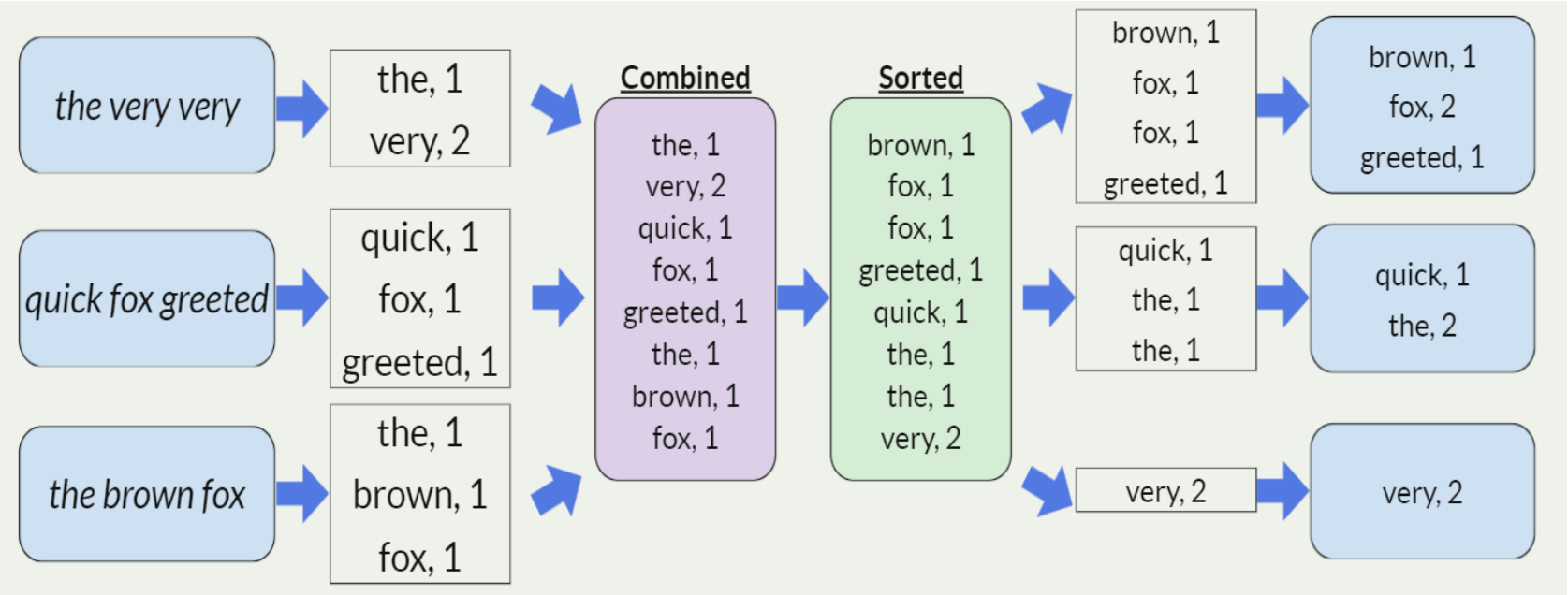
Problem: what if a word appears in multiple pieces? We need to then merge the counts

Idea: combine all the output, sort it, split into pieces, combine in each one concurrently

Example: Counting Word Frequencies

- Idea: *split documents into pieces, count words in each piece concurrently.* Then, *combine* all the text output, *sort* it, *split* into pieces, *sum each one concurrently*.

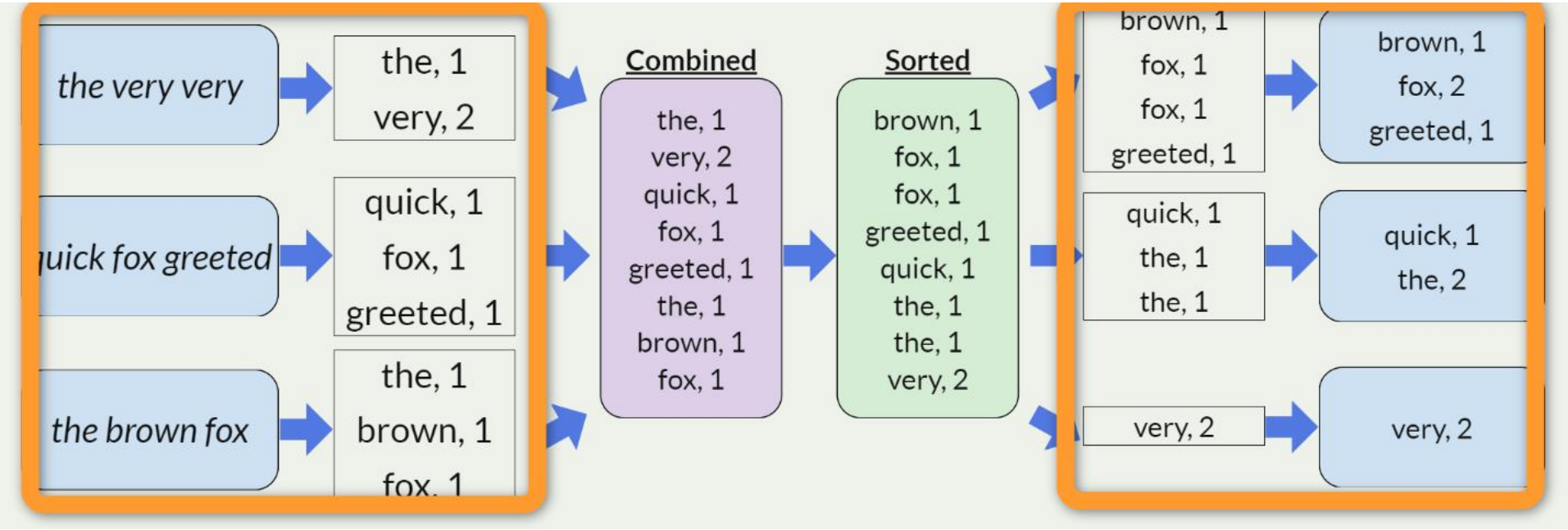
Example: "the very very quick fox greeted the brown fox"



Example: Counting Word Frequencies

2 “phases” where we parallelize work

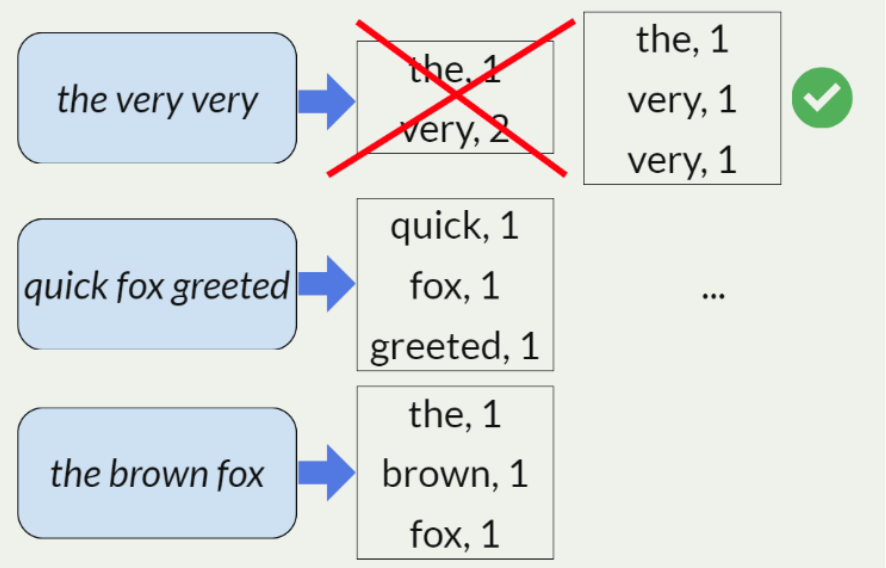
1. ***Map*** the input to some intermediate data representation
2. ***Reduce*** the intermediate data representation into final results



Example: Counting Word Frequencies

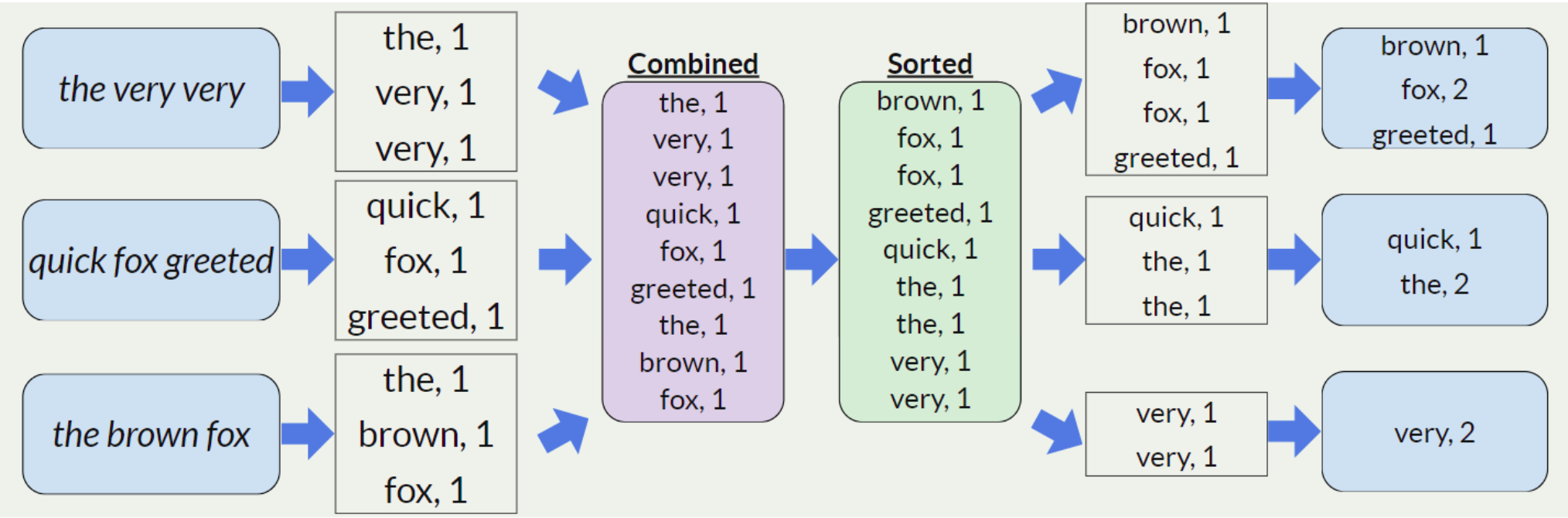
- The first phase focuses on *finding*, and the second phase focuses on *summing*. So the first phase should only *output 1s*, and leave the *summing for later*.

Example: "the very very quick fox greeted the brown fox"



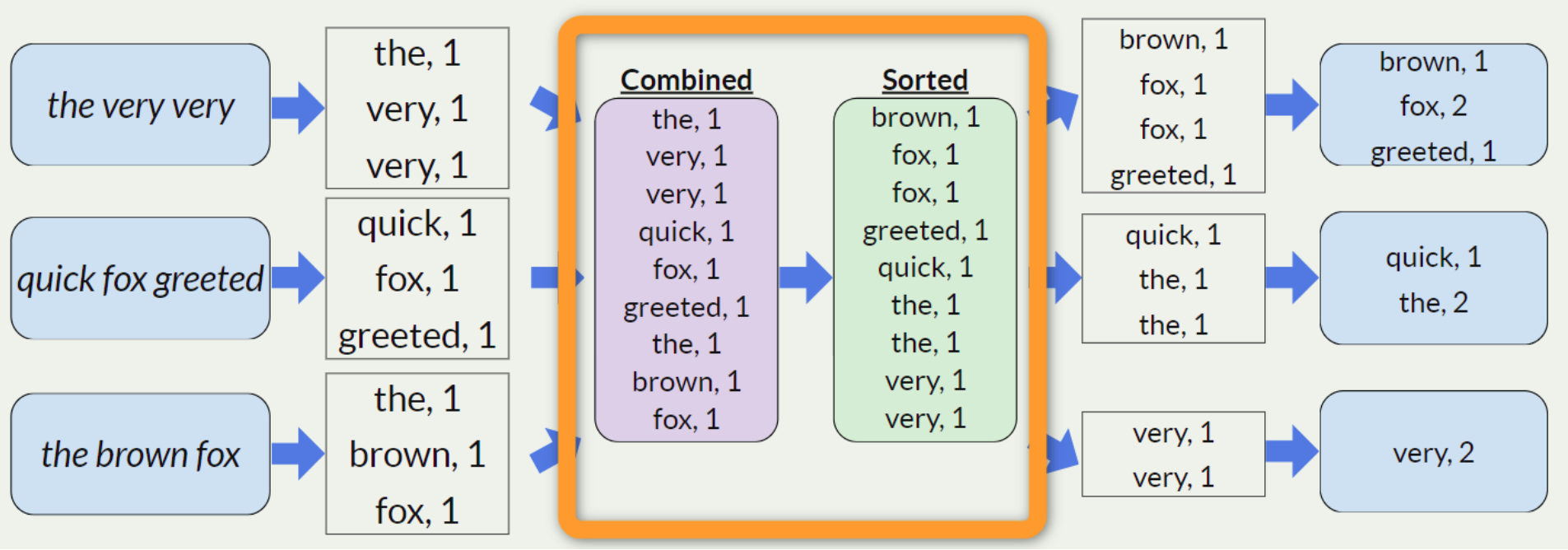
Example: Counting Word Frequencies

- 2 “phases” where we parallelize work
  - ***Map*** the *input* to some *intermediate data representation*
  - ***Reduce*** the *intermediate* data representation into ***final results***



Example: Counting Word Frequencies

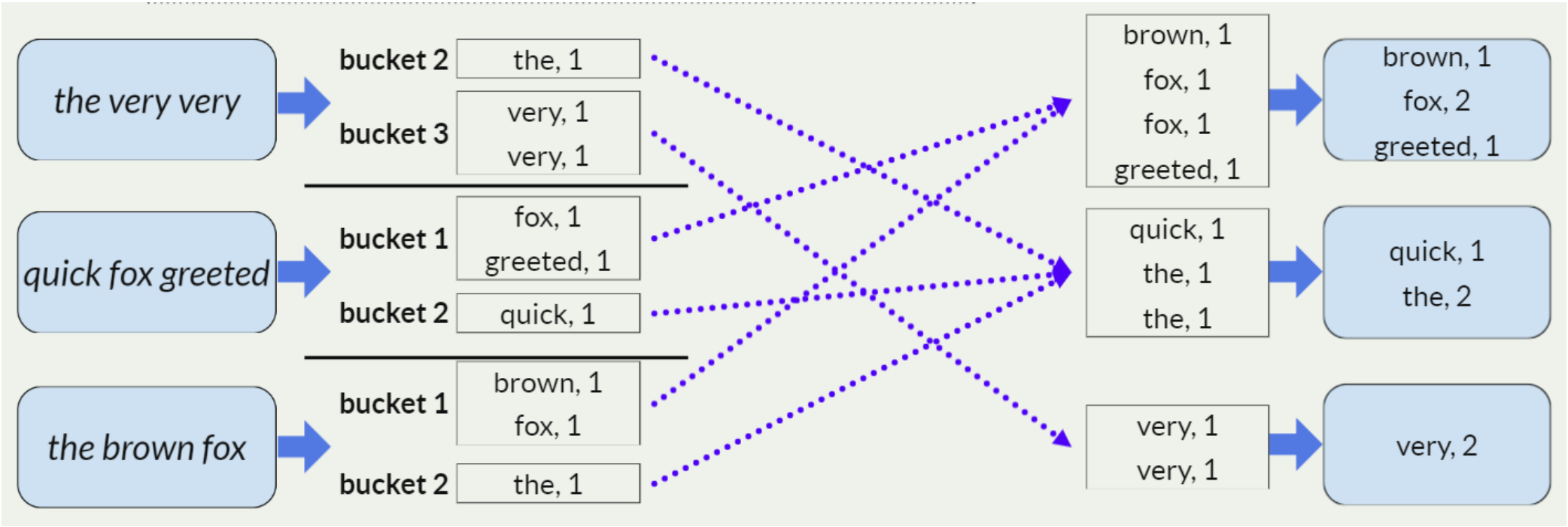
- Question: is there a way to *parallelize* this *operation* as well?
- Idea: have each *map* task separate its data in advance for each *reduce* task. Then each *reduce* task can *combine* and *sort* its own data.



Example: Counting Word Frequencies

- Question: is there a way to *parallelize* this *operation* as well?
- Idea: have each *map* task separate its data in advance for each *reduce* task. Then each *reduce* task can *combine* and *sort* its own data.

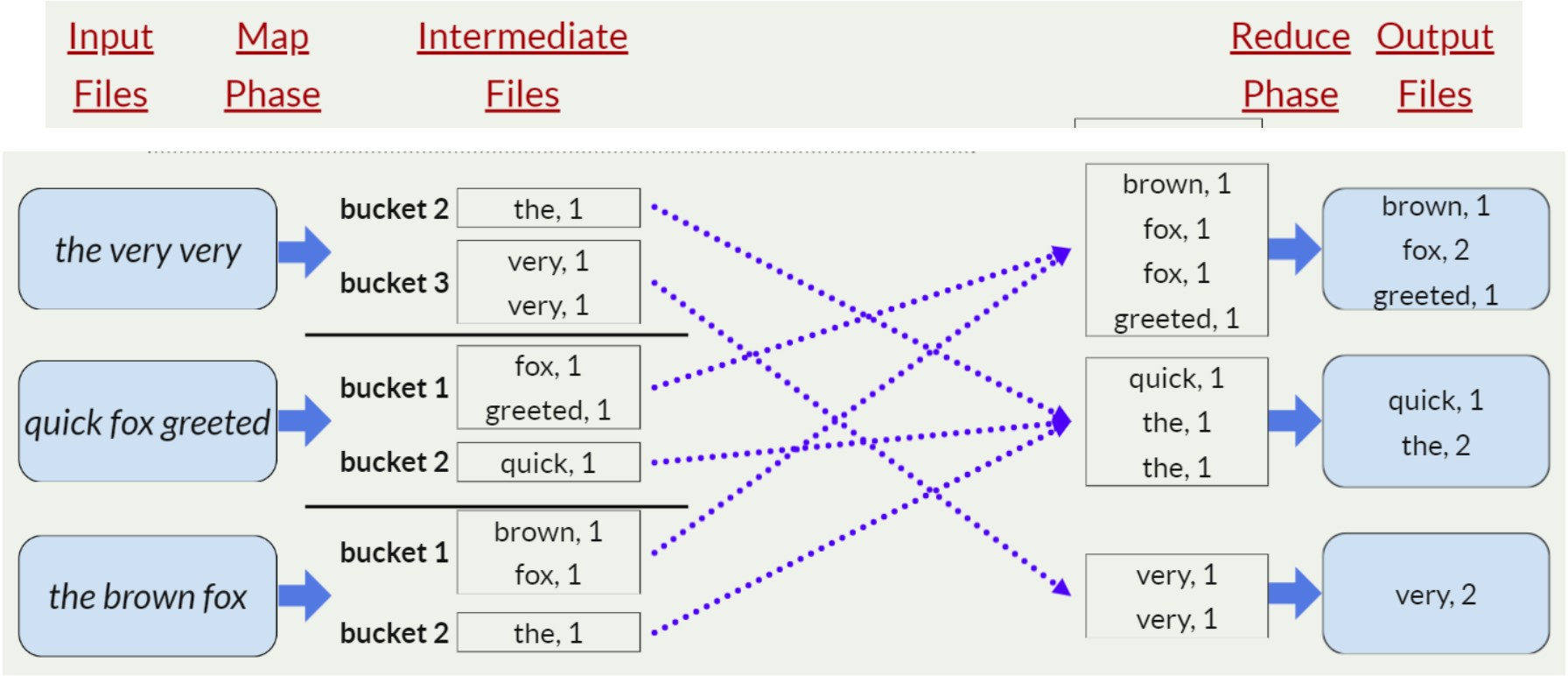
bucket # = *hash(key) % R* where R = # reduce tasks (3)



Example: Counting Word Frequencies

- Idea: have each *map* task separate its data in advance for each *reduce* task. Then each *reduce* task can *combine* and *sort* its own data.

bucket # = *hash(key) % R* where R = # reduce tasks (3)



Parallelizing Programs – Case Study: Counting Word Frequencies

- *Standard Approach*: program that reads document and builds a word -> frequency map

- *Parallel Approach*: split document into pieces, count words in each piece concurrently, partitioning output. Then, sort and reduce each chunk concurrently.

Parallelizing Programs

- *Word frequencies: split document into pieces, count words in each piece concurrently, partitioning output. Then, sort and reduce each chunk concurrently.*

We expressed these problems in this two-step structure:

- ***map** the input to some intermediate data representation*
- ***reduce** the intermediate data representation into final result*

Parallelizing Programs

- *Word frequencies: split document into pieces, count words in each piece concurrently, partitioning output. Then, sort and reduce each chunk concurrently.*

We expressed these problems in this two-step structure:

- ***map** the input to some intermediate data representation*
- ***reduce** the intermediate data representation into final result*

Not all problems can be expressed in this structure. But if we can express it in this structure, we can parallelize it!

What is MapReduce?

- ***MapReduce*** is a library that runs an operation in parallel for you if you specify the input, map step and reduce step.

- Original Paper:

MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean & Sanjay Ghemawat, OSDI '04

http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Cited over 22,000 times!

MapReduce

- Goal: a *library* to make running *programs across multiple machines* easy
- Many *challenges* in *writing programs spanning many machines*, including:
  - Machines failing
  - Communicating over the network
  - Coordinating tasks
  - Etc.
- MapReduce handles these challenges! The programmer just needs to express their problem as a *MapReduce program* (*map* + *reduce* steps) and they can easily run it with the ***MapReduce library***.
- Example of how the *right abstraction* can *revolutionize* computing
- An *open-source implementation* immediately appeared: *Hadoop* (not optimized well)

Remote Procedure Call (RPC)

- Request/response protocols are useful and widely used but rather clunky to use
  - e.g. need to define the set of requests, including how they are represented in network messages

- A nice abstraction is remote procedure call
  - Programmer simply invokes a procedure
  - But it executes on remote machine (the server)
  - By using RPC, programmers of distributed applications avoid the details of the interface with the network.
  - RPC subsystem handles message formats, sending and receiving, handling timeouts, etc

Remote Procedure Call (RPC)

- Goal: making the programming of distributed systems look similar, if not identical, to conventional programming – that is, achieving a high level of distribution transparency
- The role of underlying RPC system is to
  - hide important aspects of distribution, including the encoding and decoding of parameters and results
  - the passing of messages and the preserving of the required semantics for the procedure call
  - Programmers also do not need to know the programming language or underlying platform used to implement the service (an important step towards managing heterogeneity in distributed systems).

RPC call semantics

- Maybe semantics: With maybe semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied

- At-least-once semantics: the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received

- At-most-once semantics: With at-most-once semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

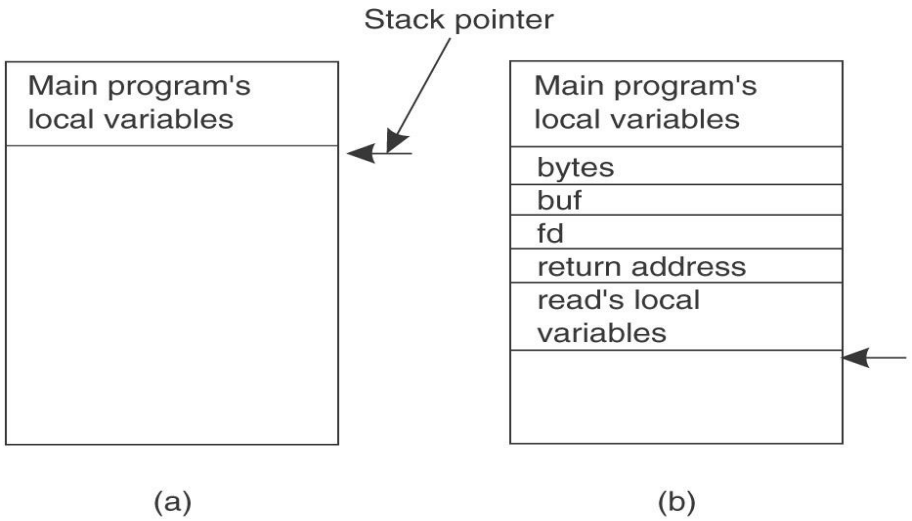
RPC implementation

When a process on machine *A calls a* procedure on machine *B, the calling process on A is suspended, and execution of* the called procedure takes place on *B.*

Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.

No message passing at all is visible to the programmer. This method is known as Remote Procedure Call, or often just RPC.

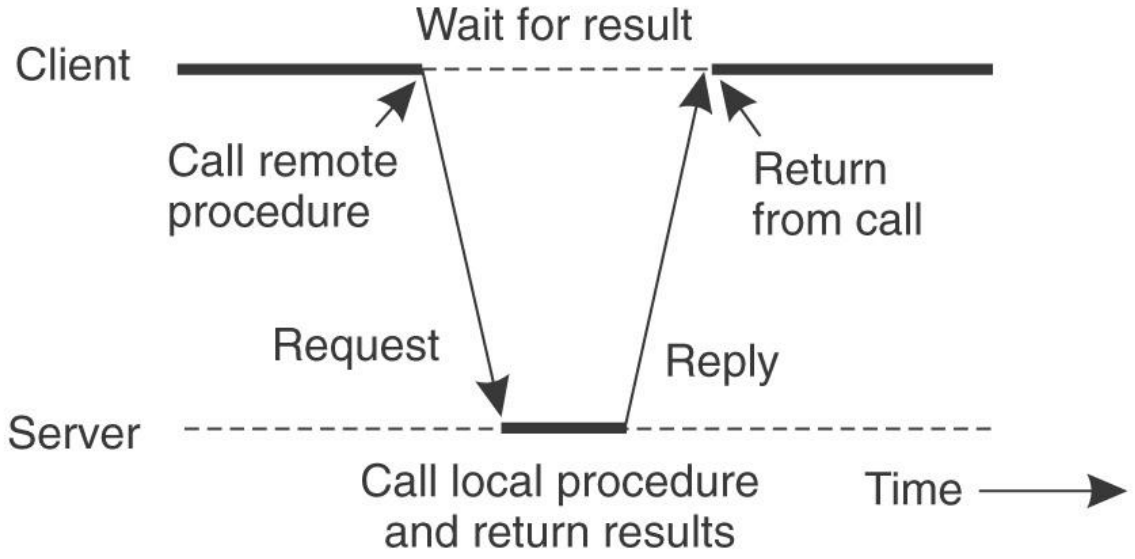
Ordinary procedure/function call



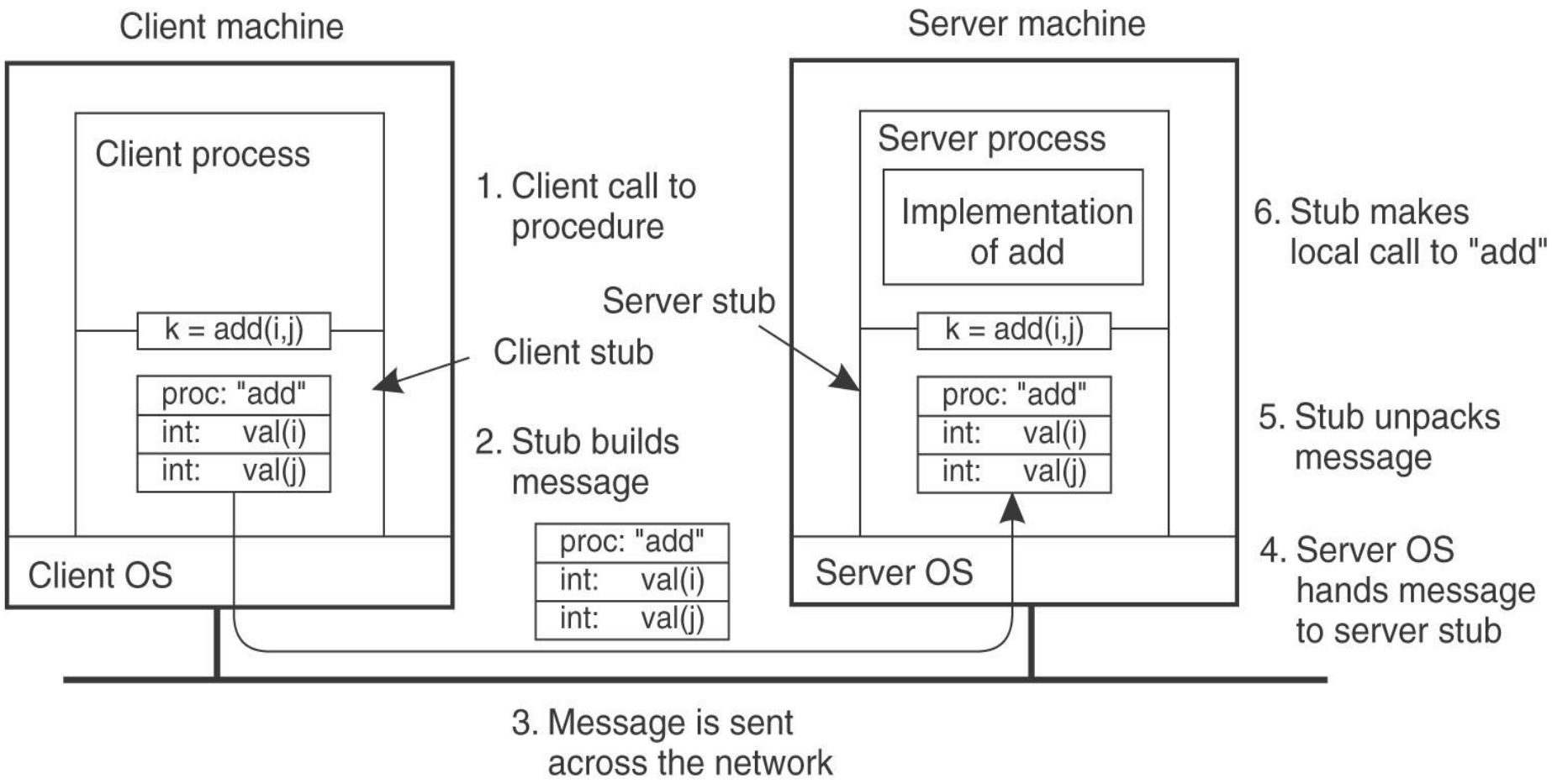
- `count = read(fd, buf, bytes)`
- After the read procedure has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The Caller then removes the parameters from the stack, returning the stack to the original state it had before the call.

Remote Procedure Call

- We would like to do the same if called procedure or function is on a remote server



RPC Stubs


Client machine
Server machine
Client process
Server process
Implementation of add
1. Client call to procedure
Server stub
Client stub
k = add(i,j)
k = add(i,j)
proc: "add"
int: val(i)
int: val(j)
2. Stub builds message
proc: "add"
int: val(i)
int: val(j)
6. Stub makes local call to "add"
5. Stub unpacks message
4. Server OS hands message to server stub
Client OS
Server OS
3. Message is sent across the network

Marshalling Arguments

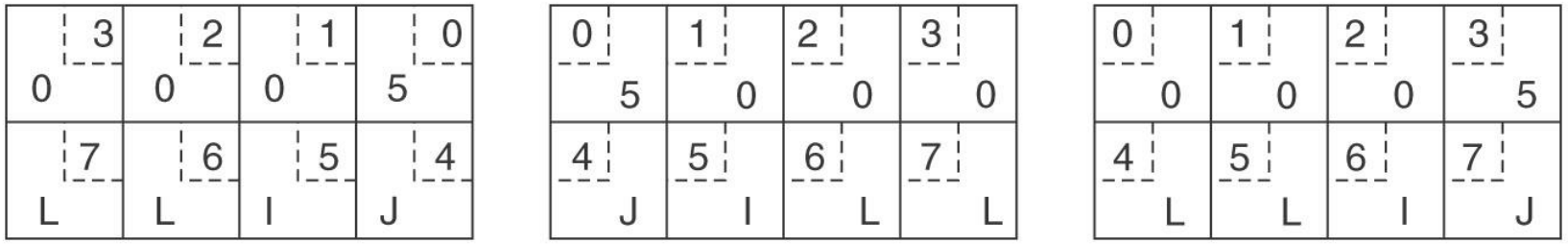
- *Marshalling* is the packing of function parameters into a message packet
  - the RPC stubs call type-specific functions to *marshal* or *unmarshal* the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

Issue #1 — representation of data

- Big endian *vs.* little endian

Solution

- Each stub converts machine representation to/from network representation
- Clients and servers must *not* try to cast data!



(a) (b) (c)

Issue #2 — Pointers and References

- Pointers are only valid within one address space
- Cannot be interpreted by another process

Even on same machine!

Solution:

1. Copy the array into the message and send it to the server.
2. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read has.
3. Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub.
4. When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client.

References

1. Slides of Dr. Haroon Mahmood

Helpful Links (on gRPC):

1. https://www.youtube.com/watch?v=gnchfOojMk4 (ByteByteGo)
2. https://www.youtube.com/watch?v=XRXTsQwyZSU (Stephane Maarek)
3. https://www.youtube.com/watch?v=hVrwuMnCtok (IBM)