Map Reduce in C

IPC - Pipelining

Grace K. Susanto

Functional Programming (Mapper - Filter - Reducer)

Mapper:

- Input: [a, b, c, d, e] and a function f(x)
- Output: [f(a), f(b), f(c), f(d), f(e)]

Filter:

- Input: [a, b, c, d, e] and a function that returns boolean f(x)
- Output: [a, c, e]

Reducer:

- Input: [a, b, c, d, e] and a function that returns one output
- Output: x

Examples

Mapper

map() applies the function func, to a sequence (eg. a list) seq.

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
>>> print Fahrenheit
[102.56, 97.70000000000003, 99.14000000000001, 100.03999999999999]
>>> Centigrade = map(lambda x: (float(5)/9)*(x-32), Fahrenheit)
>>> print Centigrade
[39.20000000000003, 36.5, 37.30000000000004, 37.79999999999997]
>>>
```

Examples

Filter

filter() filters (outputs) all the elements of a list l, for which the passed-in Boolean-valued function func returns True.

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]

>>> # for an odd number x, x%2 is 1, ie. True; so the following filters odd numbers
>>> result = filter(lambda x: x % 2, fib)
>>> print result
[1, 1, 3, 5, 13, 21, 55] # odd nums
>>> # to filter out even numbers, make it true that the mod has no remainder, ie. x%2==0
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print result
[0, 2, 8, 34] # only evens
```

Examples

Reducer

reduce() applies func repeatedly to the elements of l, to generate a single value and output it.

```
>>> r = reduce(func, l)

>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```

Map Reduce

MapReduce is a programming paradigm invented at Google, one which has become wildly popular since it is designed to be applied to Big Data in NoSQL DBs, in data and disk parallel fashion - resulting in **dramatic** processing gains.

MapReduce works like this:

0. [big] data is split into file segments, held in a compute cluster made up of nodes (aka partitions)

1. a mapper task is run in parallel on all the segments (ie. in each node/partition, in each of its segments); each mapper produces output in the form of multiple (key,value) pairs

2. key/value output pairs from all mappers are forwarded to a shuffler, which consolidates each key's values into a list (and associates it with that key)

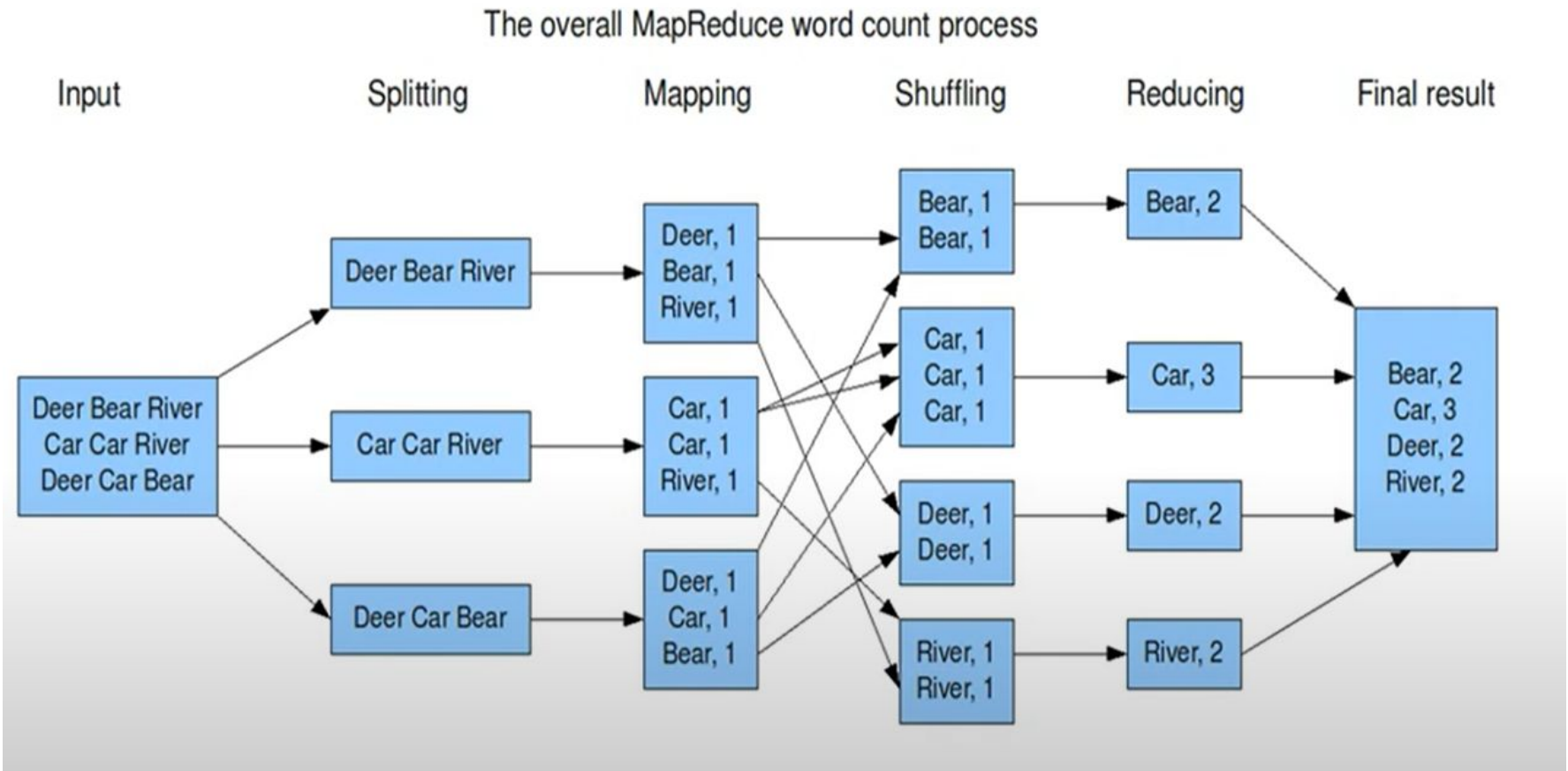
3. the shuffler forwards keys and their value lists, to multiple reducer tasks; each reducer processes incoming key-value lists, and emits a single value for each key

Map Reduce Word Count

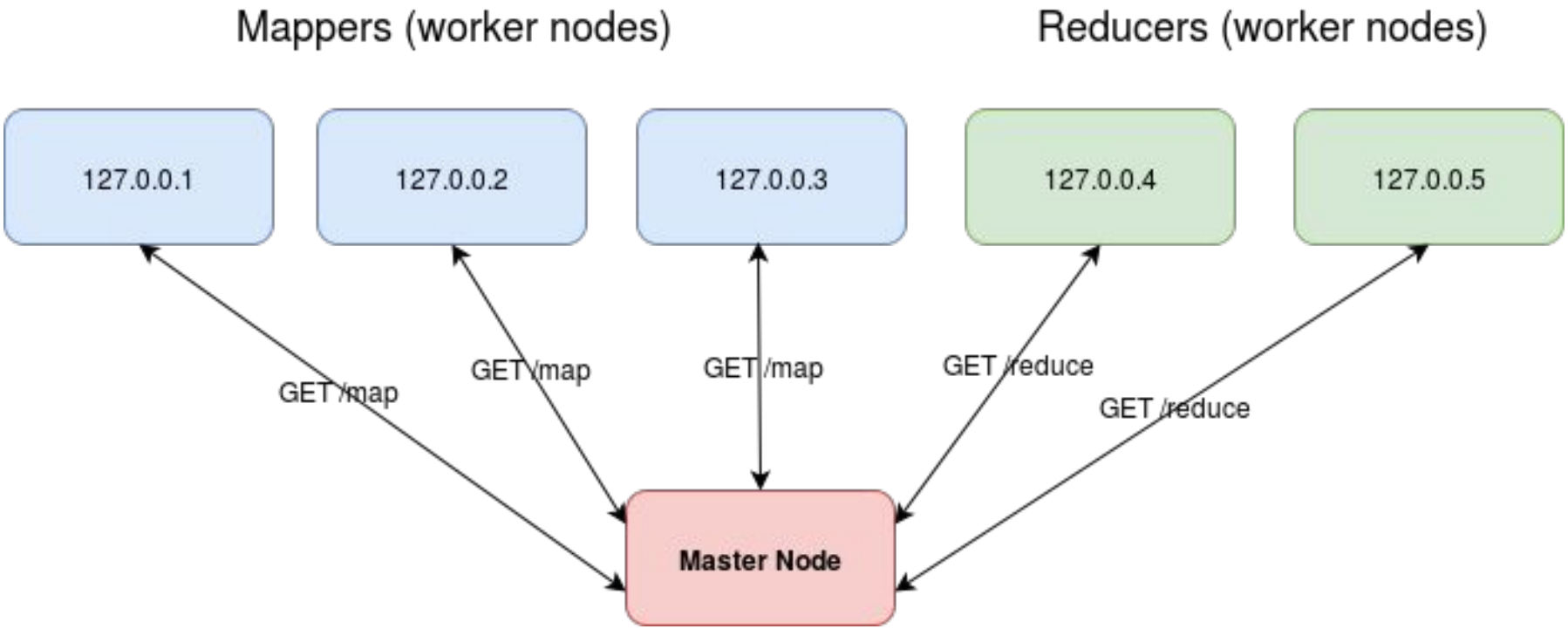
```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

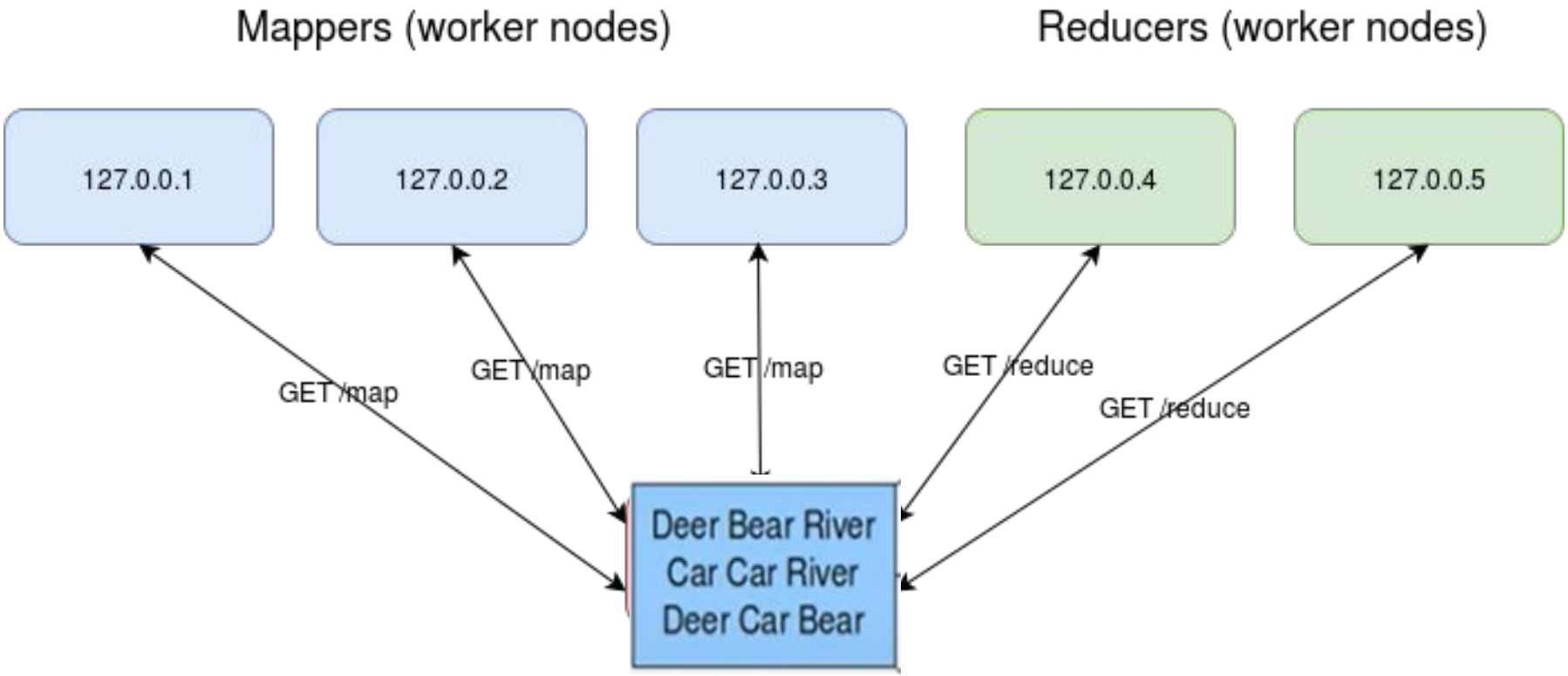
MapReduce Word Count



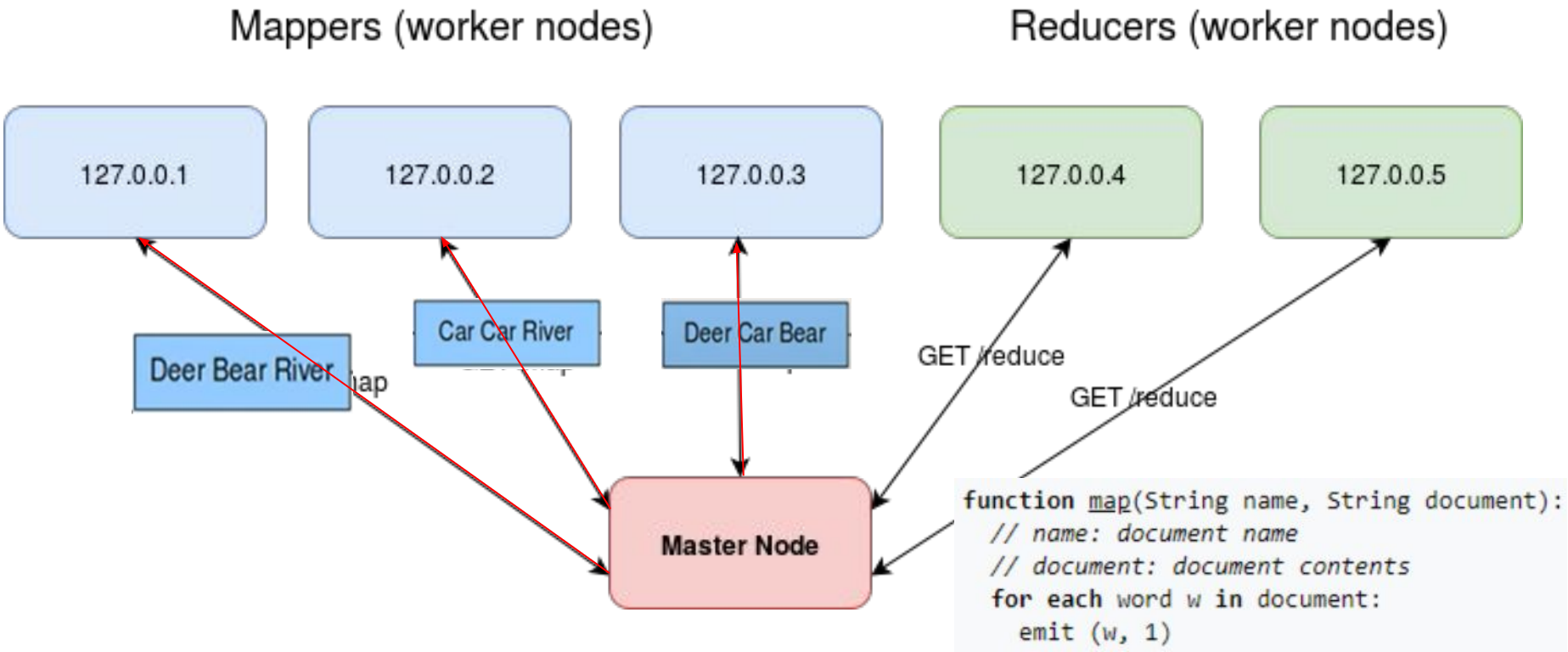
Map Reduce in distributed computing



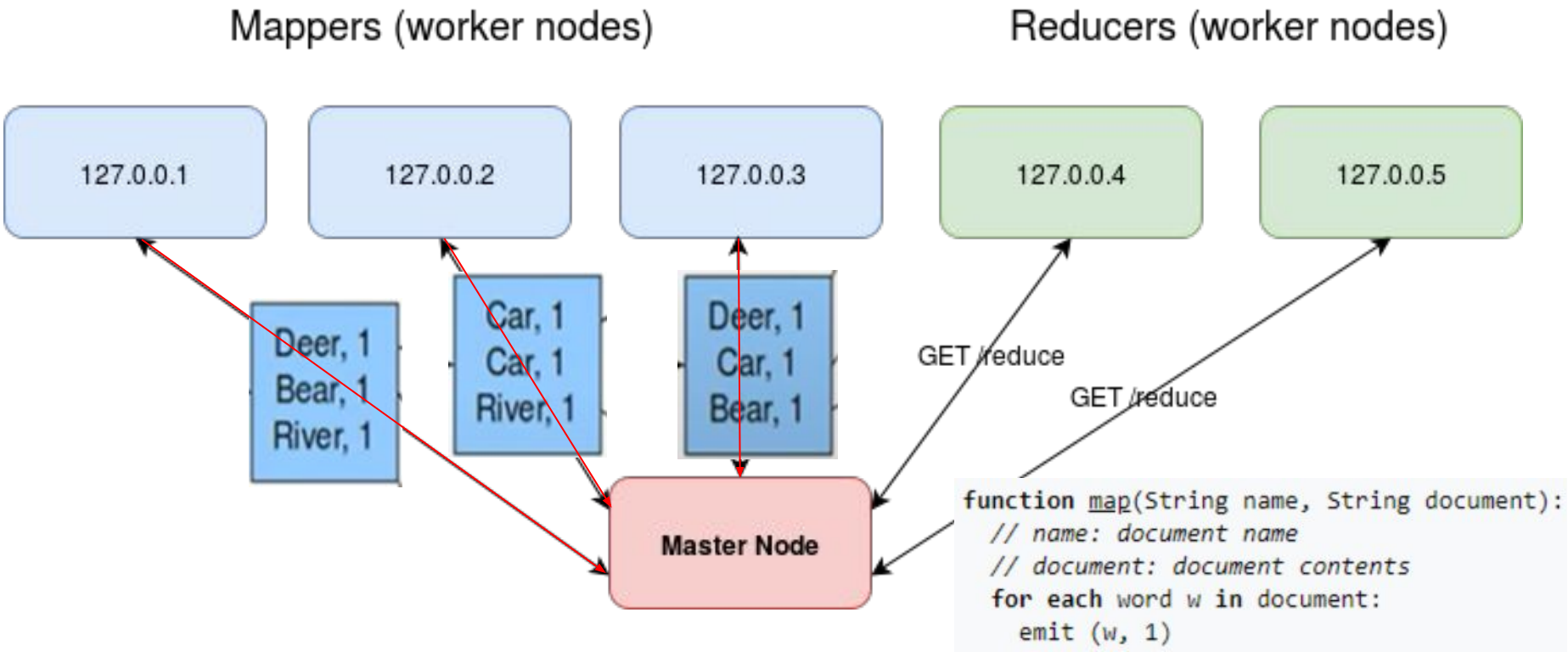
Input (master)



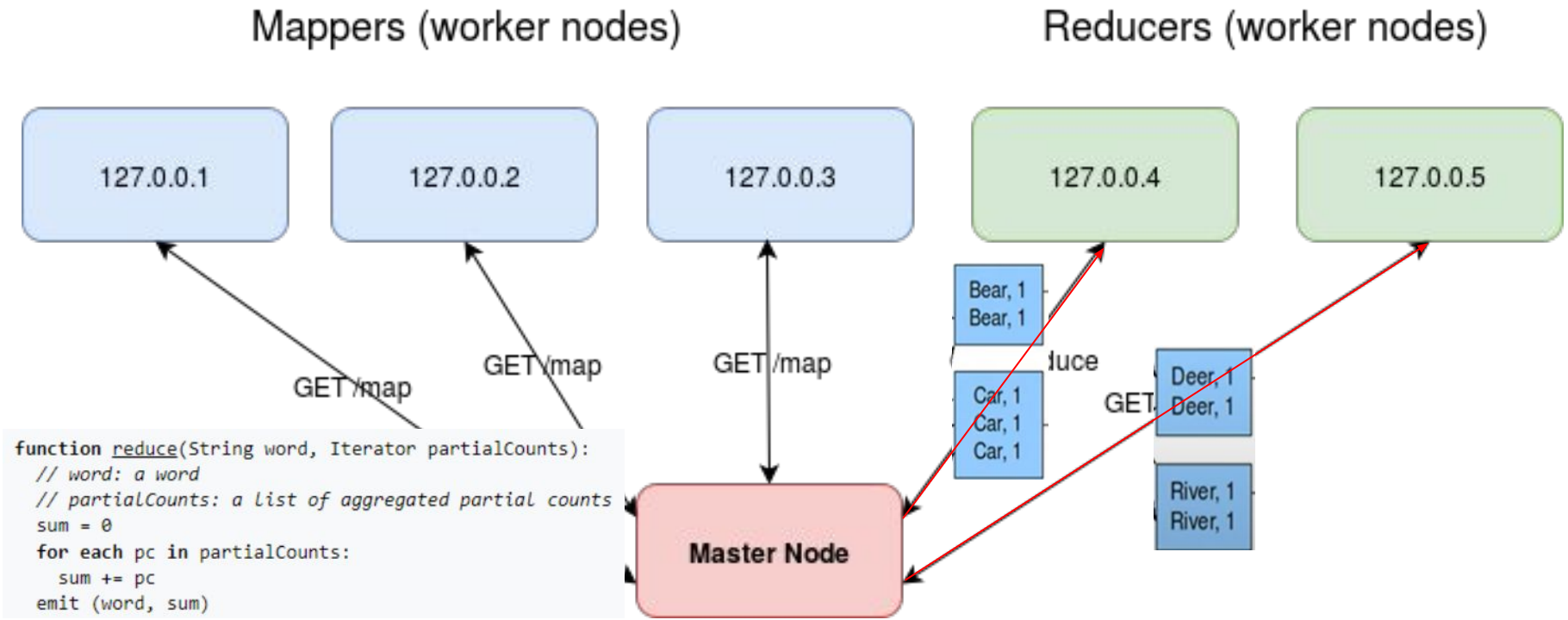
Splitting (master -> mapper)



Mapping (mapper -> master)

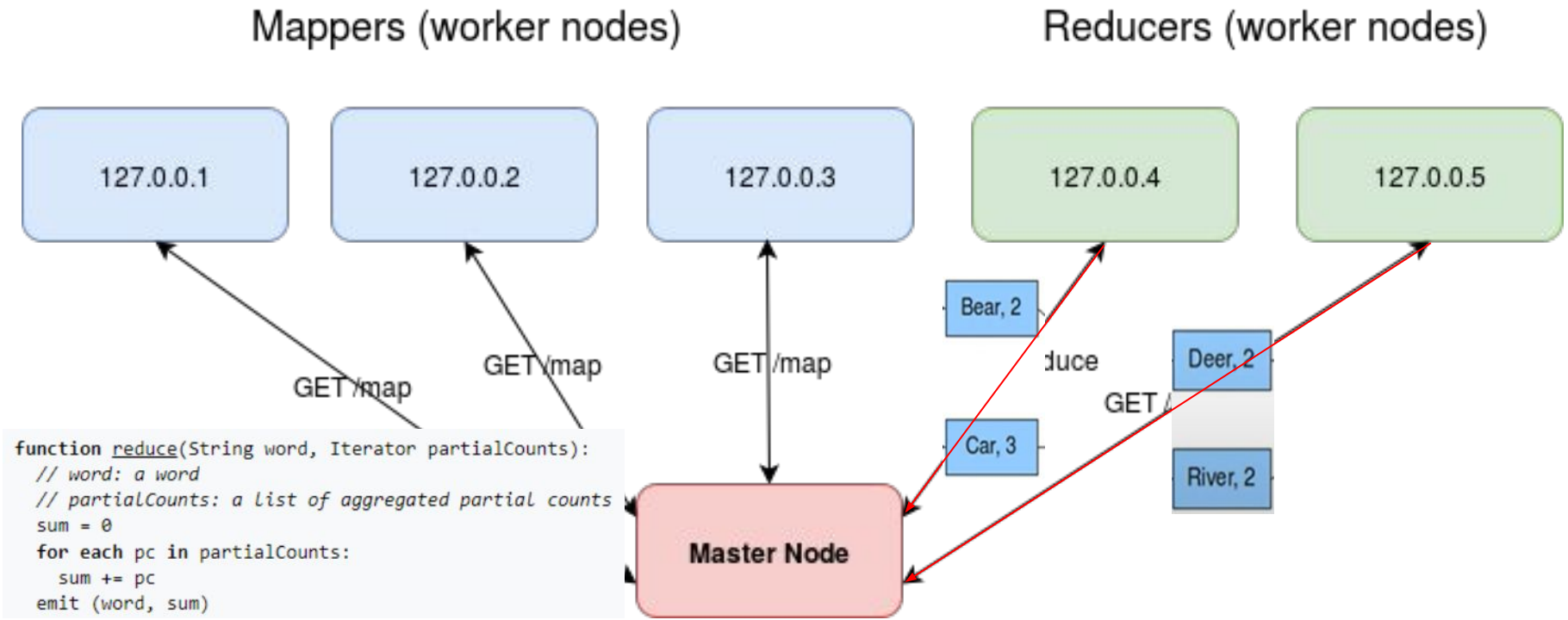


Shuffling (master -> reducer)



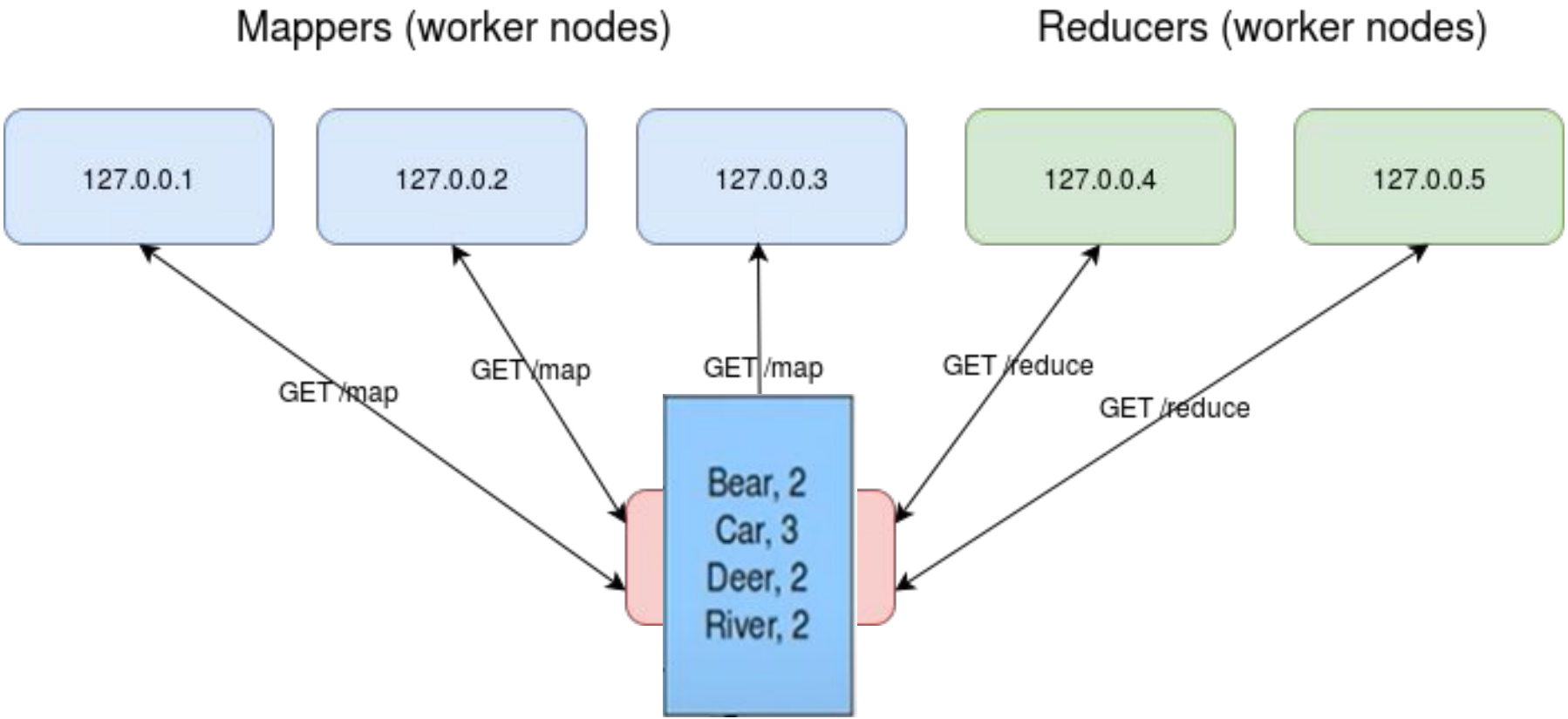

```
function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

Reducing (reducer -> master)




```
function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

Final result (master) → aggregate results from reducers



Map Reduce (summary)

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.

Map Reduce in C

- You start with one process
- You want to fork new processes to be the mapper and reducer
- There must be a way for the parent process (master) to talk to the mapper and reducer (inter process communication IPC)
- To do this, Unix OS provides an API to for different processes to talk to each other using **pipe**

Pipe

```
int pipe2(int pipefd[2], int flags);
```

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see pipe(7).

- More about Unix pipe can be found in the man page:
  - https://man7.org/linux/man-pages/man2/pipe.2.html

File Descriptor

proc.h

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct thread threads[NTHREAD]; // static thread array
};
```

File Open

- fd = open(“ls.c”, O_READONLY) – open a file. OS returns a file descriptor.

Each process has its own file descriptor table.

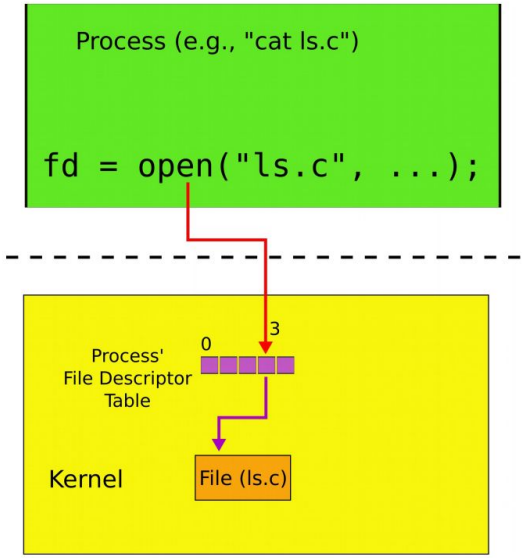
An fd is just an integer, an index to the file descriptor table

The table maintains pointers to “file” objects

- Abstracts files, devices, pipes
- In UNIX everything is a file – all objects provide file interface

Process may obtain file descriptors through

- Opening a file, directory, device
- By creating a pipe
- Duplicating an existing descriptor



Read / Write interface

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see pipe(7).

ssize_t read(int *fd*, **void** **buf*, **size_t** *count*);

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

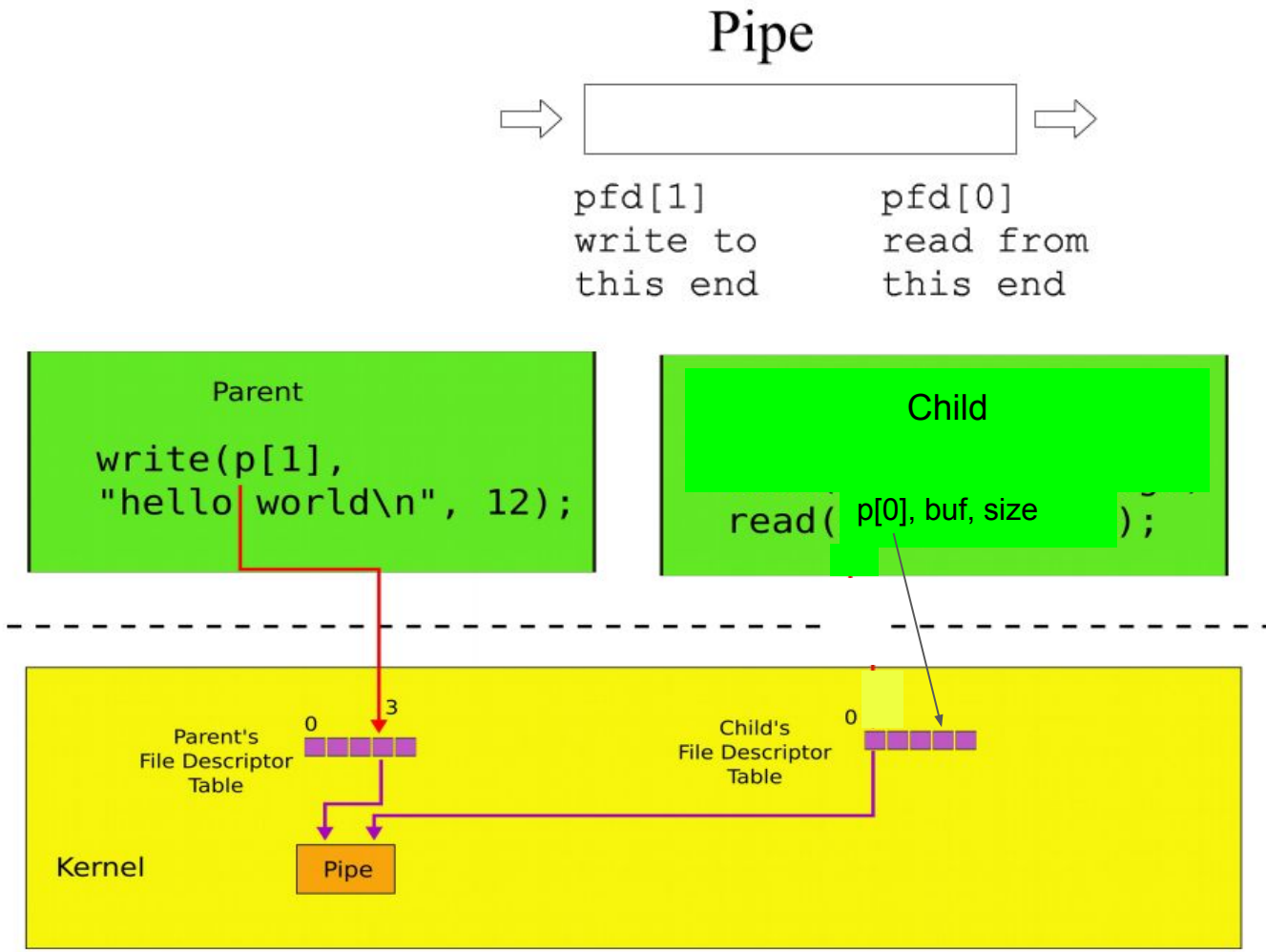
ssize_t write(int *fd*, **const void** **buf*, **size_t** *count*);

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

IPC

Parent:

```
int p[2];
pipe(p);
fork(); //Child inherits
        parent's open fd
```



IPC Example: Creates pipe, fork, child inherits fd, child writes to parent

```c
int main(int argc, char* argv[]) {
    int fd[2];
    // fd[0] - read
    // fd[1] - write
    if (pipe(fd) == -1) {
        printf("An error ocurred with opening the pipe\n");
        return 1;
    }
    int id = fork();
    if (id == 0) {
        close(fd[0]);
        int x;
        printf("Input a number: ");
        scanf("%d", &x);
        write(fd[1], &x, sizeof(int));
        close(fd[1]);
```

Source: https://www.youtube.com/watch?v=Mqb2dVRe0uo

IPC (cont'd) Child writes to parent, parent reads from child

```c
int id = fork();
if (id == 0) {
    close(fd[0]);
    int x;
    printf("Input a number: ");
    scanf("%d", &x);
    write(fd[1], &x, sizeof(int));
    close(fd[1]);
} else {
    close(fd[1]);
    int y;
    read(fd[0], &y, sizeof(int));
    close(fd[0]);
    printf("Got from child process %d\n", y);
}
```

Source:
https://www.youtube.com/watch?v=Mqb2dVRe0uo

IPC (cont'd)

```
int id = fork();
if (id == 0) {
    close(fd[0]);
    int x;
    printf("Input a number: ");
    scanf("%d", &x);
    write(fd[1], &x, sizeof(int));
    close(fd[1]);
} else {
    close(fd[1]);
    int y;
    read(fd[0], &y, sizeof(int));
    close(fd[0]);
    printf("Got from child process %d\n", y);
}
```

It is important to close unnecessary pipe.

Closing write end of the pipe is important because read is a blocking call. Read will not stop reading until write pipe is closed.

Closing read end of the pipe is important because the process will not exit until all read/write pipes have been closed.

Source: https://www.youtube.com/watch?v=Mqb2dVRe0uo

Project 5

What you need to do:

- Do word count using map reduce in C
- IPC should be handled using pipes
- You need to use 8 mappers and 2 reducers (N = 8, M = 2) → do fork 10 times, 8 times you call mappers function, 2 times you call reducers function
- What you do in mapper and reducer is up to you.
  - Mapper should read words from master (parent) and write key-value pair
  - Reducer should read key-value pair and write a sorted list of word-wordCnt to master (parent)
- Then master will print the resulting word-wordCnt to stdout

You will not be using xv6 for this project.

You will code directly in C using Linux APIs (mainly pipe, read, write, fork).

Understanding starter code

https://github.com/gsusanto/CSCI350P5

2 data structures:

- Hashmap (key=word, value=word count) to use its `find` functionality
- Array of `count` struct to aggregate the result

```
- cmp_count --> comparison for sort
- read input into word
- search for word in hash table
    - if found, add 1 to word count
    - else, insert data with count 1
      then add the word to words array
- after read is done, go through all the words in the words array
    - for each words, find the word count in hash map
    - add it as word count int the words array
- sort words array
- output
```

Roughly, what you need to write is

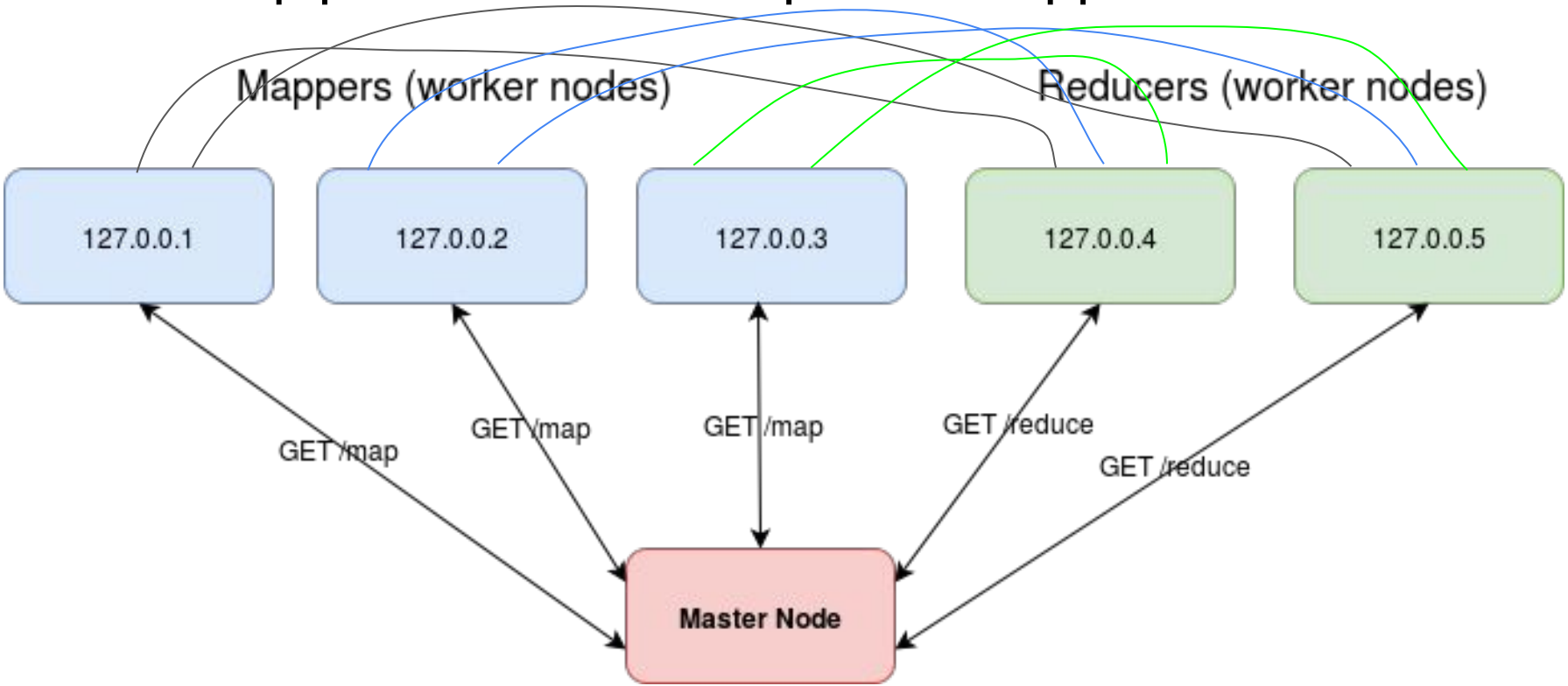
```
- main will send data to mapper
- mappers will wait for input from main
- mappers will process the data, resulting in kv pair
- mappers will send the kv pair to reducers
- reducers will wait for input from mappers
- reducers will sort the data
- reducers will send the sorted data to main
- main will wait for input from reducer
- main will will write to output file
```

(Or some other implementation, it is up to you. But this should give you a rough idea on what is expected from project 5)

In this implementation: master -> mapper, mapper -> reducer, reducer -> master (mapper communicate directly with reducer)

Other possible implementation (as shown in the word count MapReduce in distributed system example) :
master -> mapper, mapper -> master, master -> reducer, reducer -> master

Will need pipe between each pair of mapper-reducer



Functions you need to write:

- Main function
  - Create pipes
  - Will do fork (N+M) times
  - In the child, N times it wants to call mapper, M times it wants to call reducer
  - Read input file, split words into N chunks, send to mapper
  - Wait for reducer to send back the resulting word-wordCount
- Mapper function that takes in int as an input (pipe index)
  - Read data from main
  - Add word to word count list and keep track of how many occurrences
  - Write word-wordCount kv pair to reducer depending on what letter the word starts with
- Reducer function that takes in int as an input (pipe index)
  - Read the word-wordCount kv pair
  - Sort the word
  - Write resulting list to master

Why do you need pipe index?

Example code from Github

https://github.com/jeffrey-garcia/C-MapReduce

```
int masterToParentPipe[2]; // for Master -> Parent pipe

int masterToMapperPipe[2]; // for Master -> Mapper pipe
int mapperToMasterPipe[2]; // for Mapper -> Master pipe

int masterToReducerPipe[2]; // for Master -> Reducer pipe
int reducerToMasterPipe[2]; // for Reducer -> Master
```

In our case, we want to use 8 mappers and 2 reducers

N = 8, M = 2

```
static int masterToMappers[N][2]; // File Descriptors
static int reducersToMaster[M][2];
```

And some other arrays that contains pipe.

Then create the pipes in main function

```
// Creating all the pipes
for (int i=0; i<N; ++i) {
  if (pipe(masterToMappers[i]) < 0) {
    printf("Error in creating pipes\n");
  }
}
```

Why do you need pipe index?

Master will create lots of pipe, one pipe for every inter process communication.

When master forks, it will create child processes that inherits parent's file descriptors, including all of the pipes.

However, each process only needs a subset of the pipes.

How do you know which pipe is yours?

By passing a unique index to each child process (mapper or reducer).

N=4 (mappers)
M=2 (reducers)

```
mas2map[N][2]
red2mas[M][2]

for i = 0, ..., N
  pipe(mas2map[i])
for i = 0, ..., M
  pipe(red2mas[i])

fork (M+N) times
```

read write
0 1

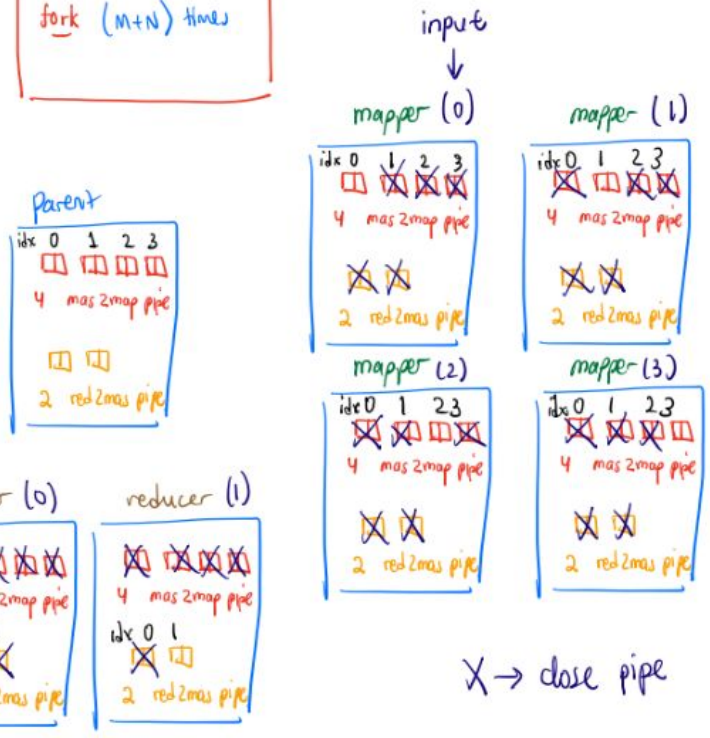
N=4 (mappers)

M=2 (reducers)

```
mas2map[N][2]
red2mas[M][2]
for i = 0, ..., N
  pipe(mas2map[i])
for i = 0, ..., M
  pipe(red2mas[i])
fork (M+N) times
```

read write

0	1



IMPORTANT!!

Close unnecessary pipes in all functions

Ex: In master (parent), you will not be reading from masterToMappers pipe.
So you close fd[0]. (Depending on your implementation, you might need to read from masterToMappers pipe)

```
// Close useless pipes
for (int i=0; i<N; ++i) {
  // Since master will write to this
  // we can close recieving end
  close(masterToMappers[i][0]);
}
```

Reading from Pipes

In mapper function,
mapper function takes as input *int pipe* (index of the pipe for pipe array)
Mapper will then read data from master.

```
char word[101], letter[1];
int status, w_status;
int currIdx;

while ((status = read(masterToMappers[pipe][0], letter, 1)) > 0) {
  currIdx = 0;
  while (*letter != '\0') {
    word[currIdx++] = *letter;
    status = read(masterToMappers[pipe][0], letter, 1);
  }
  word[currIdx] = '\0';
  // Do something with word
}
```

Writing to Pipes

In reducer,
Reducer function also takes as input *int pipe* (index of the pipe for pipe array)
Reducer will then write word-wordCnt to master.

```
w_status = write(reducersToMaster[pipe][1], words[i].word, strlen(words[i].word)+1);
w_status = write(reducersToMaster[pipe][1], &words[i].count, sizeof(words[i].count));
```

As a context, `words` is an array of `count` struct (from starter code)

```
typedef struct {
    char* word;
    int count;
} count;
```

```
count* words = calloc(MAX_UNIQUES, sizeof(count));
int num_words = 0;
```