Background: Map function

- Take Python as an example

```
def mult2(number):
    return number * 2
numbers = [0,1,2,3]
doubled = map(mult2, numbers)
```

- Output is [0,2,4,6]

Background: Reduce function

- Take Python as an example

```
import functools
L = [0,2,4,6]
sum = functools.reduce(lambda a, b: a+b, L)
// or sum = functools.reduce(operator.add, L)
```

- Output is 12

Background: Map and Reduce together

- Take Python as an example

```
import functools
import operator
def mult2(number):
    return number * 2
numbers = [0,1,2,3]
sum = functools.reduce(operator.add, map(mult2, numbers))
```

- Output is 12

Parallelizing with MapReduce

- Google's MapReduce involves:
  - First, applying a map function to each logical record in the input
    - Produces a set of intermediate key/value pairs
  - Then, applying a reduce function to all values that have a common key

This is a functional model, so there are no side effects (will become quite important in the implementation)

(Silly) MapReduce Example: Count Occurrences of Each Word

```
Map(String key, String value):
    for each word w in value:
        EmitIntermediate(w, "1");
```

key: document name
value: document contents

```
Reduce(String key2, Iterator values):
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

key2: word
values: list of counts (1's)

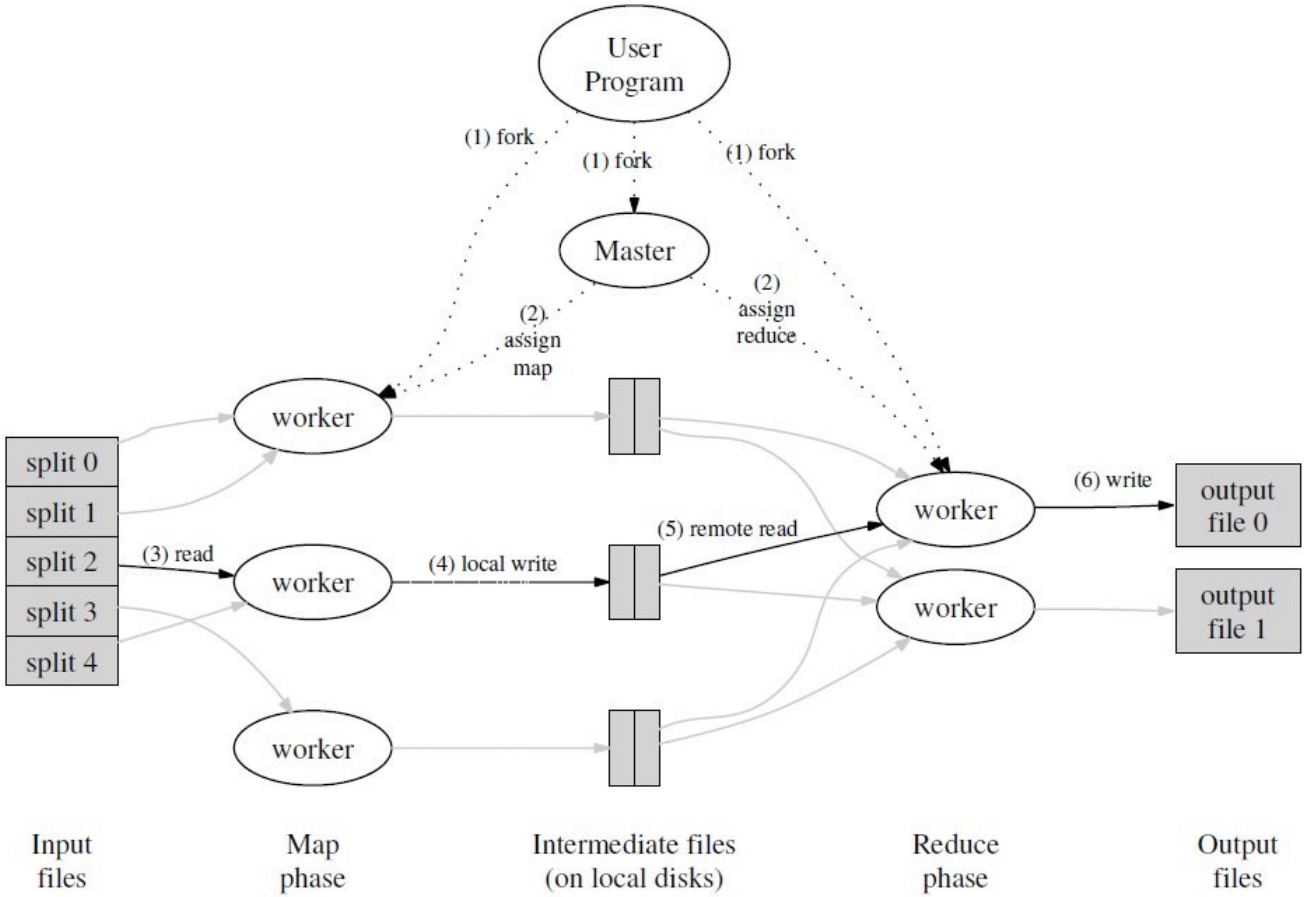
Note: key is different in Map and in Reduce

Picture of WordCount (on board)

Other MapReduce Examples

- URL Access Frequency (same as WordCount)
  - Map outputs <URL, 1> for each URL
  - Reduce adds for same URL
- Reverse Web-Link Graph
  - Map outputs <target, source> for each link to a target URL found in source
  - Reduce concatenates source URLs for each target
- Inverted Index
  - Map outputs <word, documentID>
  - Reduce outputs <word, list(documentID)>

MapReduce Implementation



Taken from Dean et al., OSDI 2004

MapReduce Implementation
(Numbers do not correspond to figure)

- Basic idea (for clusters of commodity machines):
  - 1. Use Administrator/Worker paradigm; one admin.
  - 2. Split input files into M chunks; size chosen by user
  - 3. Split reduce tasks into R pieces (hash on key and apply “mod R”, for example)
    - Admin assigns the M map tasks and R reduce tasks to workers
  - 4. Workers doing a map task write key/value lists into different files
    - If hashing the key (and applying mod R) is equal to i, then write the key/value lists to file "i".
    - Pass back this file info to admin, who tells reduce tasks

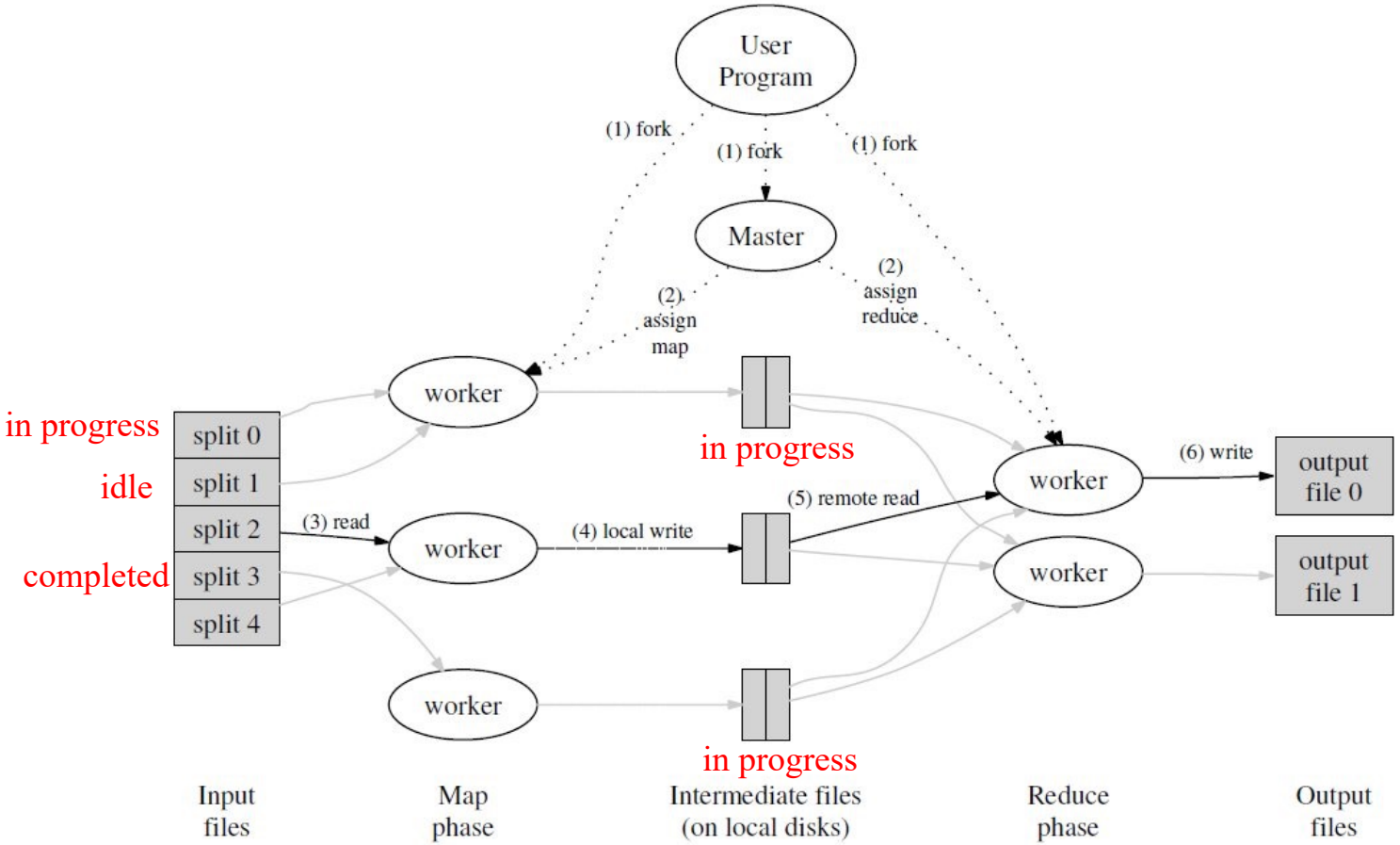
MapReduce Implementation

- 5. Reduce worker grabs its data from all local disks via RPC (remote procedure call), sorts, and reduces.
  - Sorts because many keys may be assigned to each reduce task; this groups all occurrences of the same key
  - Makes one call to user's reduce function per unique key; parameter is list of all values for that key
  - Appends output into final output file.
- 6. When everything is done, wake up MapReduce call.

Fault Tolerance with MapReduce

- Critical to handle fault tolerance because there will be thousands of machines involved
  - Probability that at least one fails is high
- Administrator keeps track of each map and reduce task
  - Marks it as *idle*, *in progress*, or *completed*
  - Administrator assumed to never fail

MapReduce Implementation

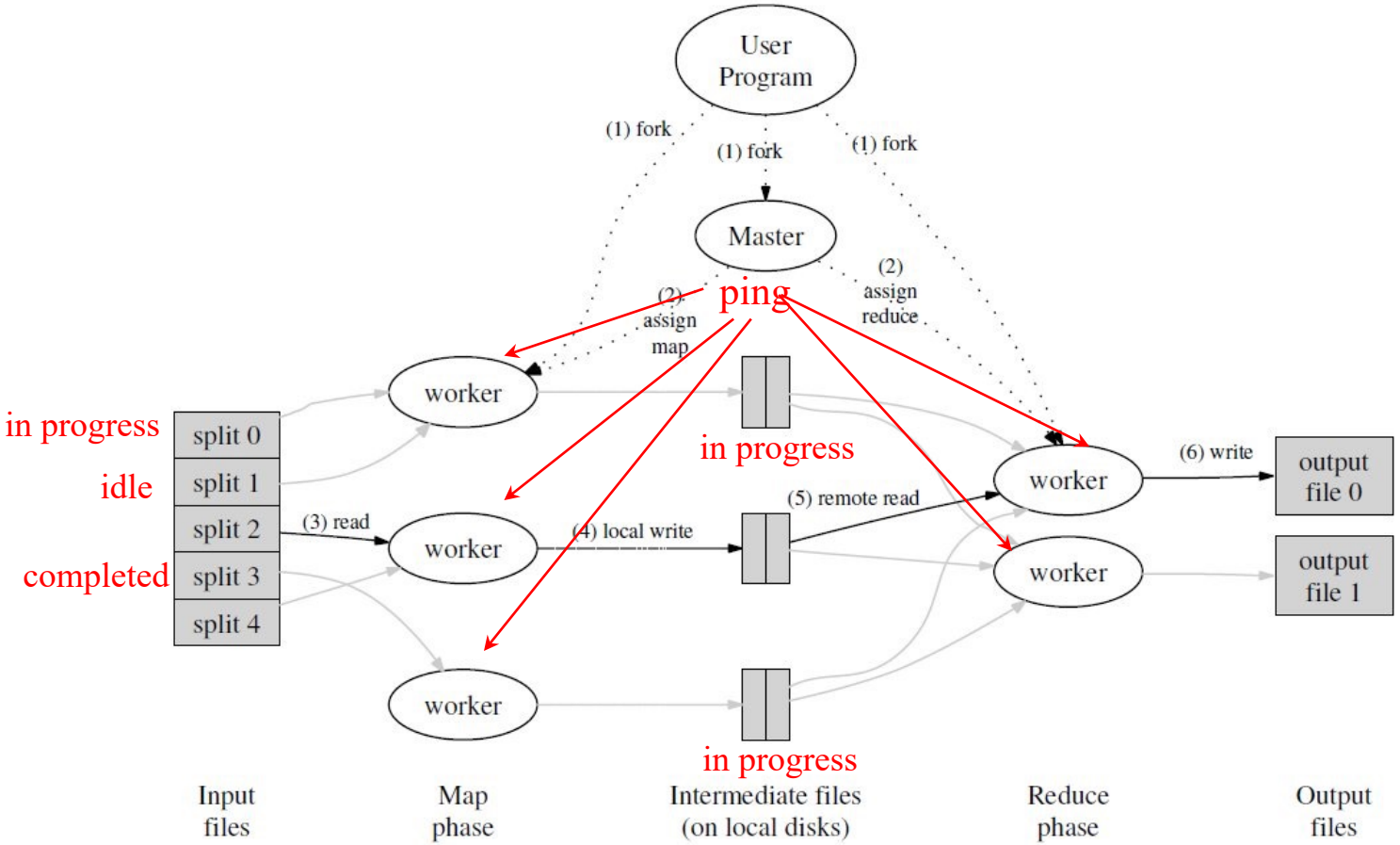


Taken from Dean et al., OSDI 2004

Fault Tolerance with MapReduce

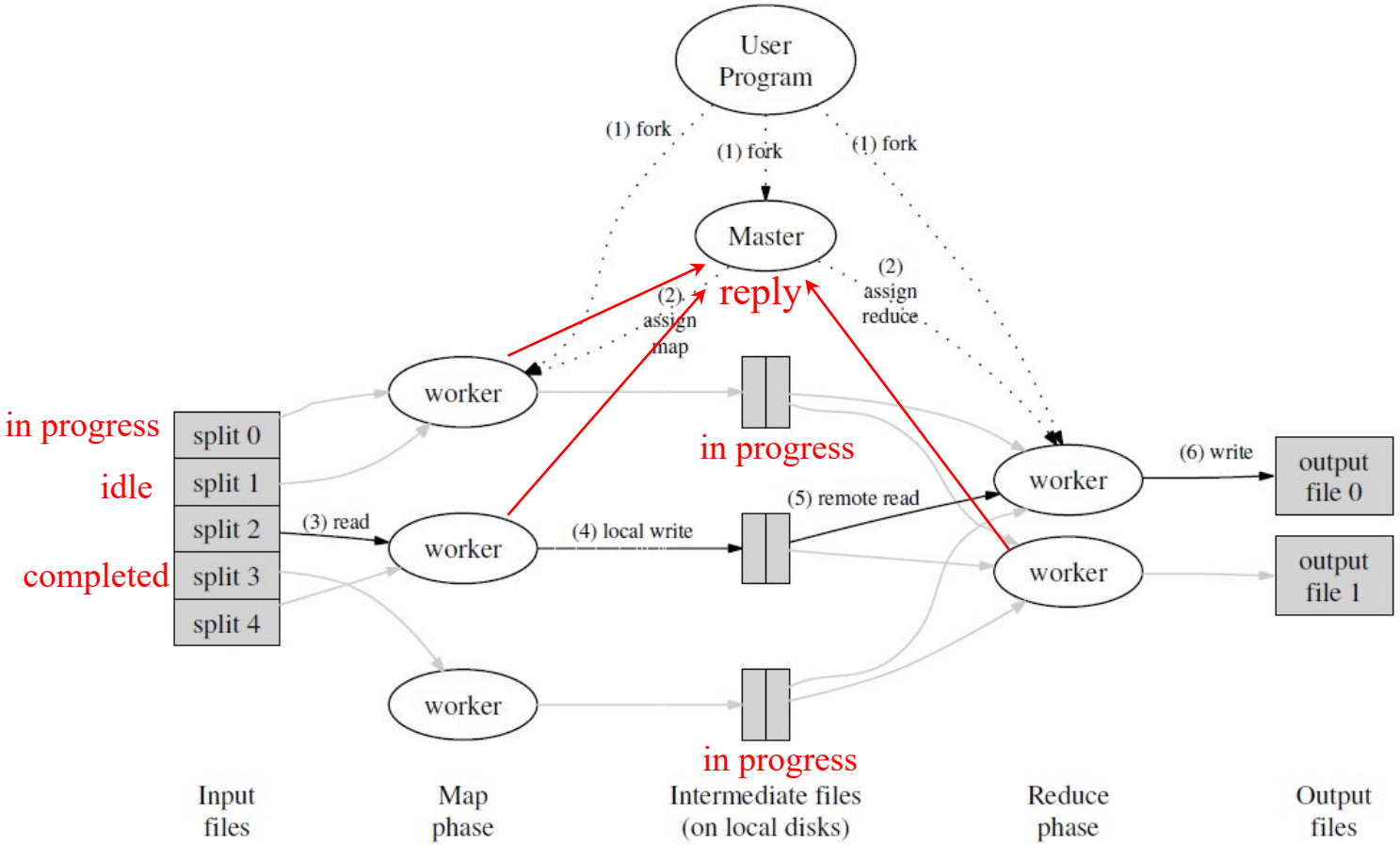
- For workers:
  - Ping periodically; if no response, mark worker as "failed"; mark worker's task as idle
  - If map worker fails, completed map tasks are *re-executed* because the worker's local disk is assumed inaccessible
  - Notify reduce tasks if a map task changes hands, so reduce tasks know where to read from
  - If reduce worker fails, assign new worker and *re-execute*

MapReduce Implementation



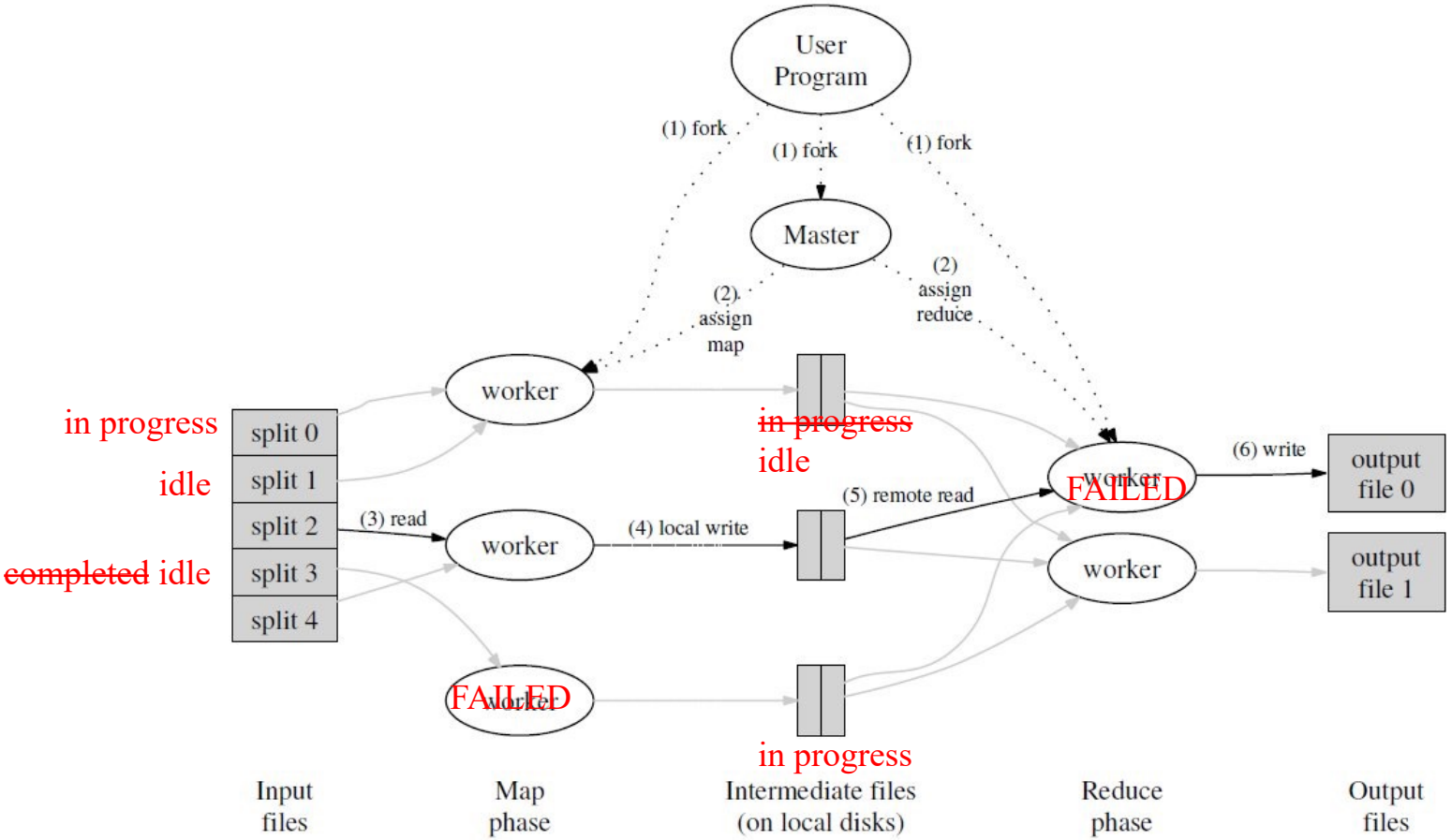
Taken from Dean et al., OSDI 2004

MapReduce Implementation



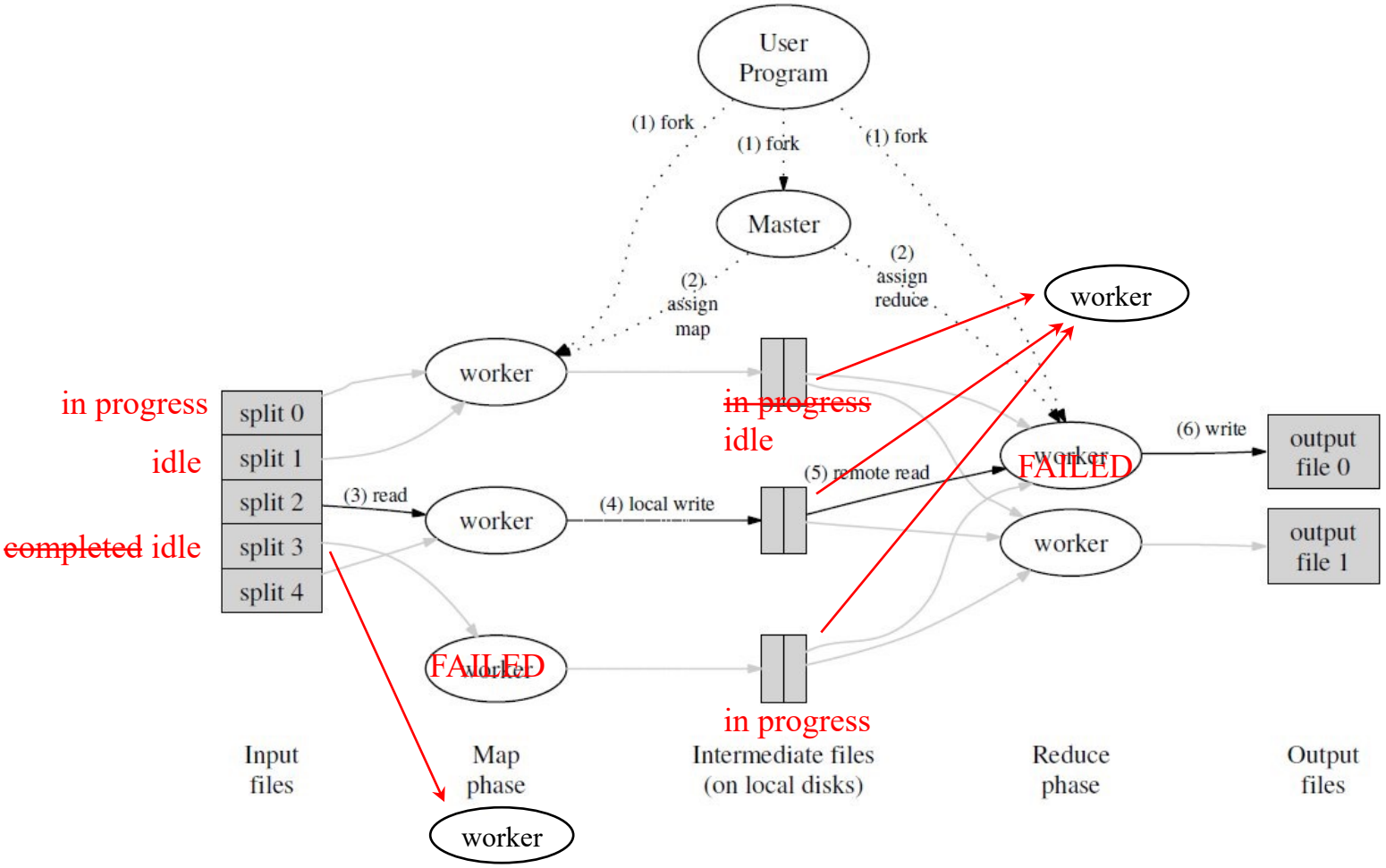
Taken from Dean et al., OSDI 2004

MapReduce Implementation



Taken from Dean et al., OSDI 2004

MapReduce Implementation

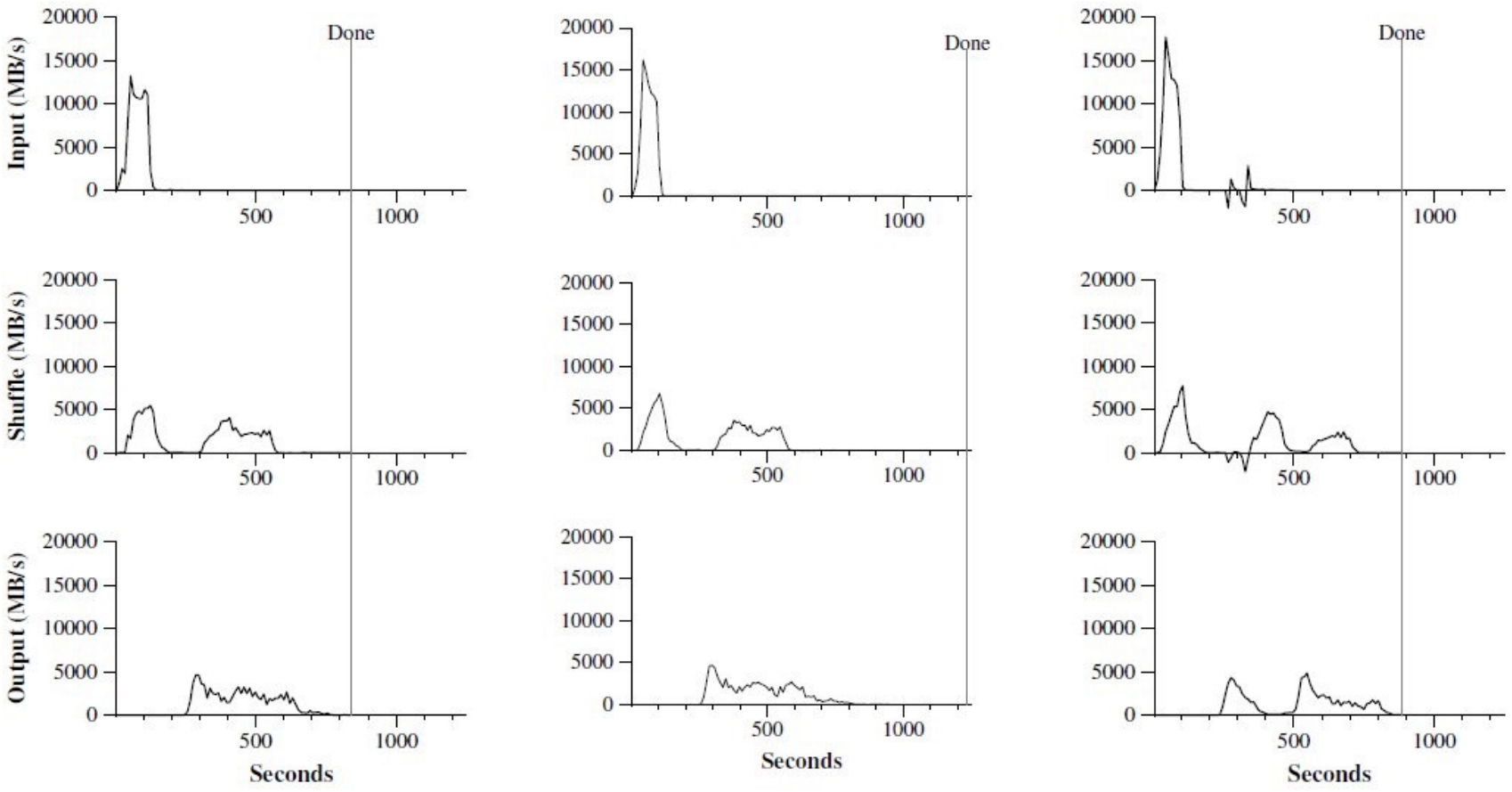


Taken from Dean et al., OSDI 2004

Fault Tolerance with MapReduce

- Fault tolerance is simple because MapReduce is a functional model
  - Can have duplicate tasks, for example---no side effects (and use “backup” tasks when nearly done)

MapReduce Performance



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

Taken from Dean et al., OSDI 2004