RPCs in Go

COS 418: *Distributed Systems*
Precept 2

Themis Melissaris and Daniel Suo

Plan

- Assignment 1 takeaways
- Assignment 2 preview
- RPCs in Go
- Implementing an RPC client

Assignment 1 takeaways

Challenges

- How do I do X in Go?
  - Go is more verbose than many expected
  - Dealing with particulars (e.g., pointers, en/decoders)
- Wait, what exactly do I need to implement?
  - Reading existing code base in new language
  - Understanding the API
- Mental model (maps and reduces) vs. implementation (queue for single worker)
  - More detail later
- Anything else?

Sorting

- Controversial!
- Two stages of sorting
  - Once in the reducer after it gathers data from each of the mappers
  - Once in the merge phase after the reducers finish
- In some sense, both sorts are design choices
  - First: see Section 4.2 of paper (also in Appendix)
  - Second: makes it easy for us to grade (and, ostensibly, for other processing, including more maps/reduces)
- No one submitted a test case!
  - Ok, because would've involved changing the API of the assignment

Assignment 2 preview

Overview

- Moving from sequential to ‘distributed’
  - Not really. Distributing across channels via RPC on Unix stream sockets
- Need to implement a scheduler that assigns map and reduce tasks to workers
- Workers may fail, but assume master does not
- Will post later today

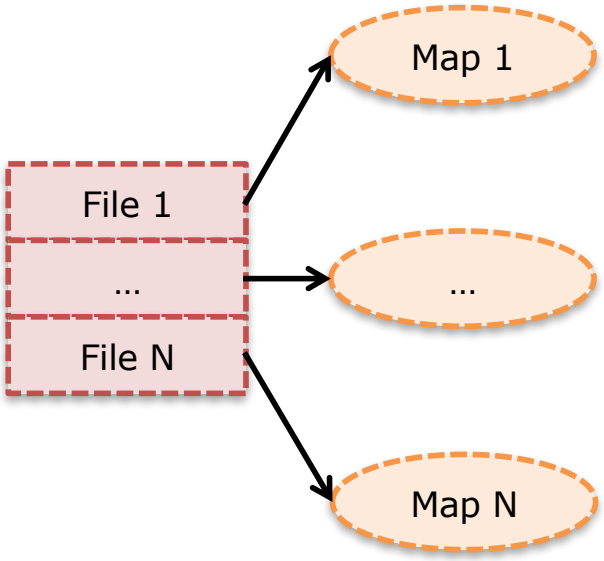
Sequential MapReduce

Mental Model

- **Start with *N* files to process in parallel**
- **MapReduce refers to these as 'splits'. Whether these *N* files or splits were from one large file or some number of existing files doesn't matter**

Sequential MapReduce

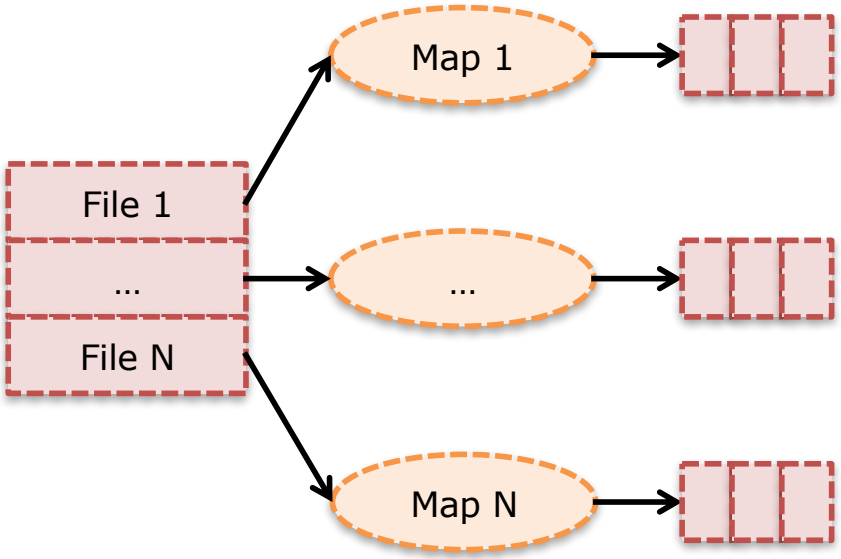
Mental Model



- **Our mental model suggests we process the *N* files in parallel, applying the map function to each file**
- **However, in the sequential implementation, we apply the map function to each file, one after another**

Sequential MapReduce

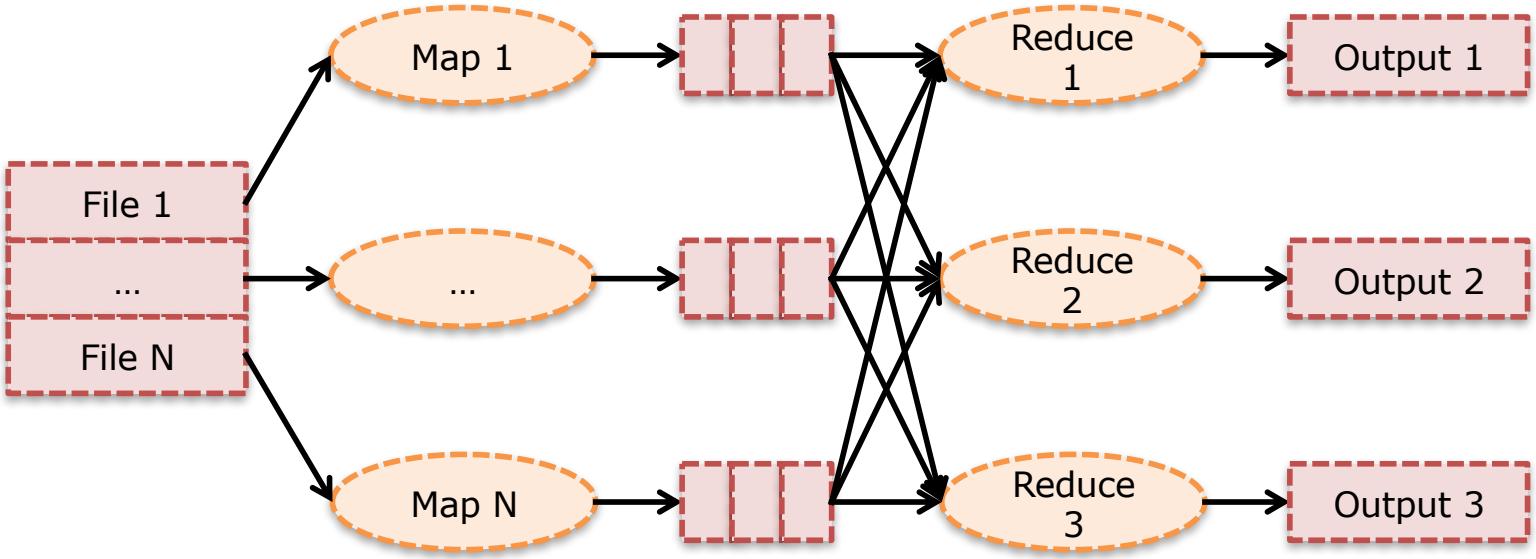
Mental Model



- **Apply the map function to each file, one after another**
- **Each map task writes its results to some number of intermediate files (equal to the number of reduce tasks; in our example, 3)**

Sequential MapReduce

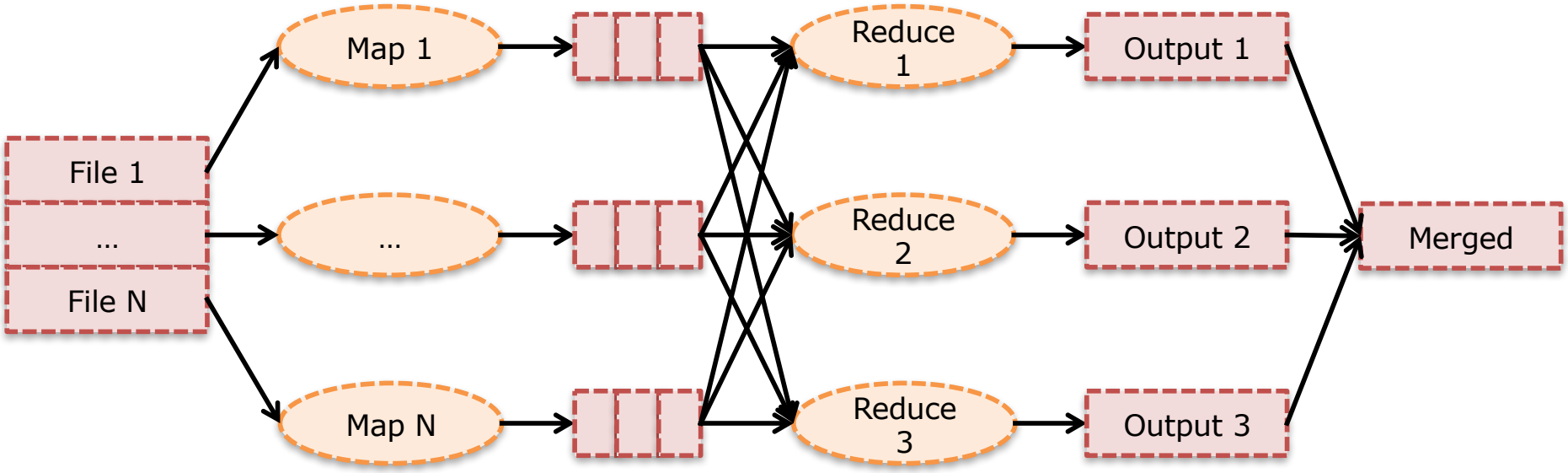
Mental Model



- **Once we complete all map tasks and write all intermediate results to disk, each reduce task reads in the relevant file produced by each map task**
- **Each reduce task reduces its input and writes its output to file**

Sequential MapReduce

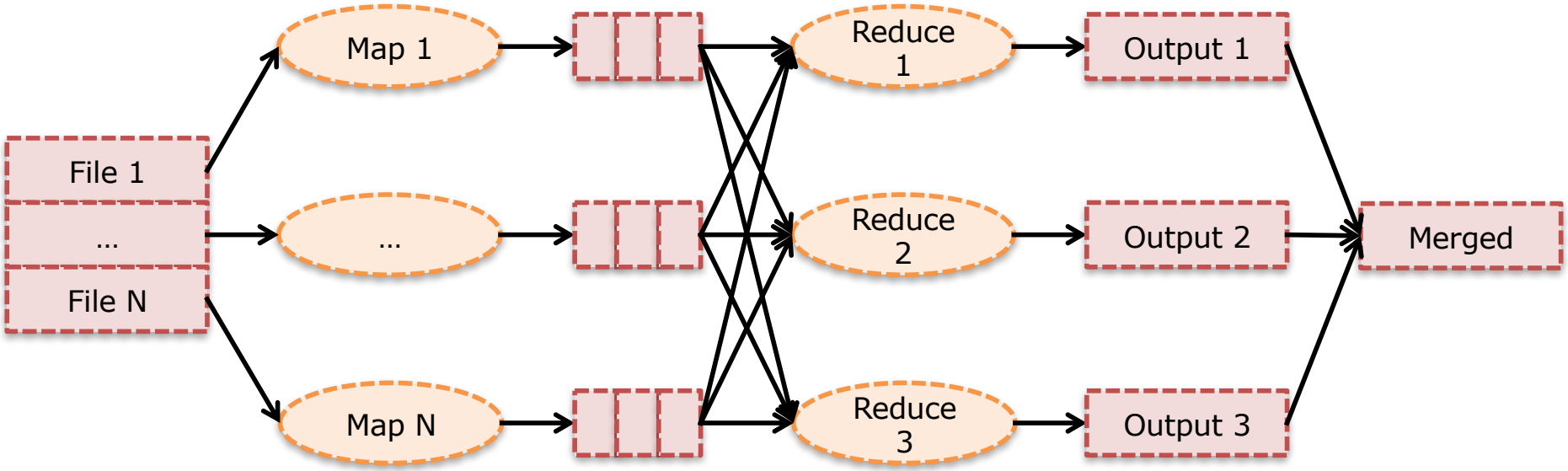
Mental Model



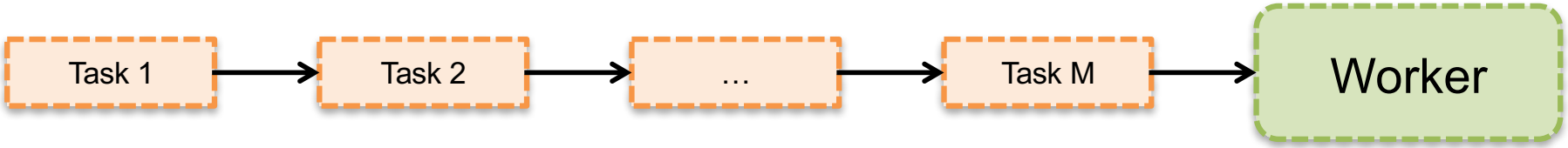
- **Finally, the master program merges all outputs from the reduce step**
- **We didn't use the word 'worker' in sequential MapReduce, but we implicitly had one worker**

Sequential MapReduce

Mental Model

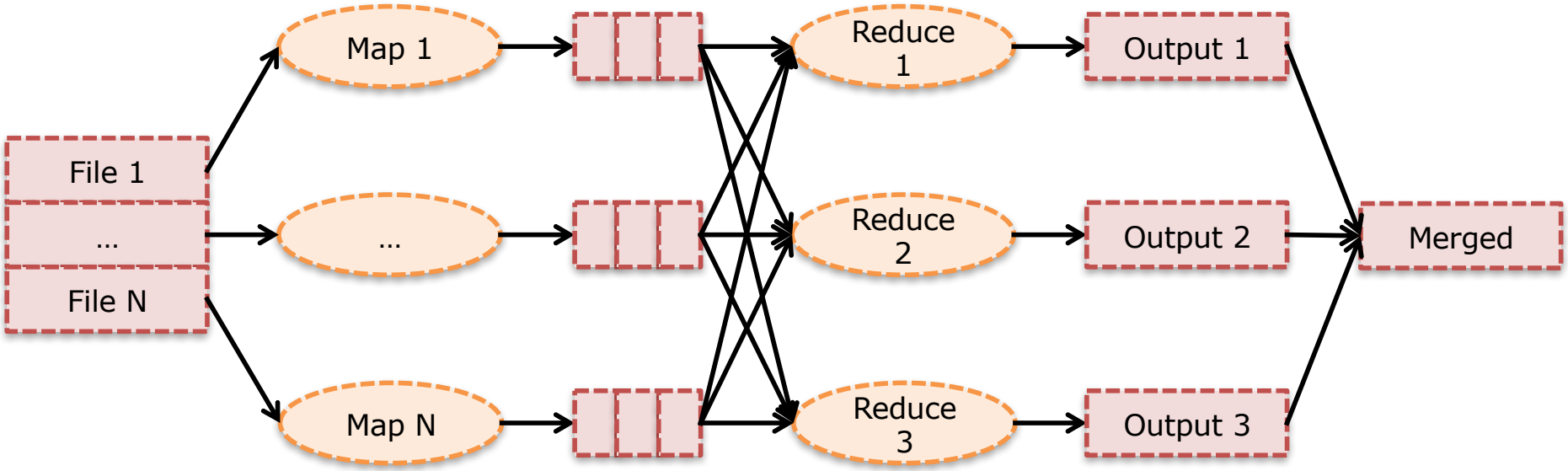


Implementation

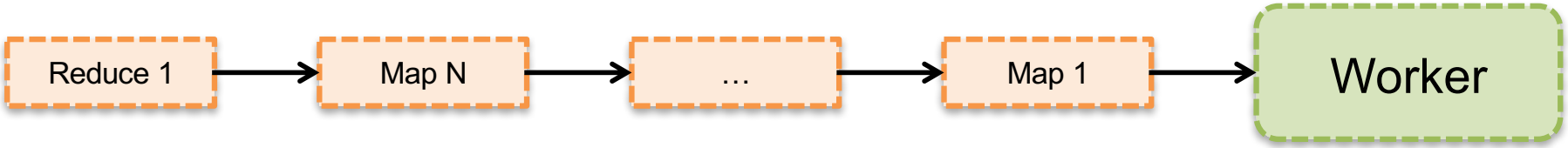


Sequential MapReduce

Mental Model

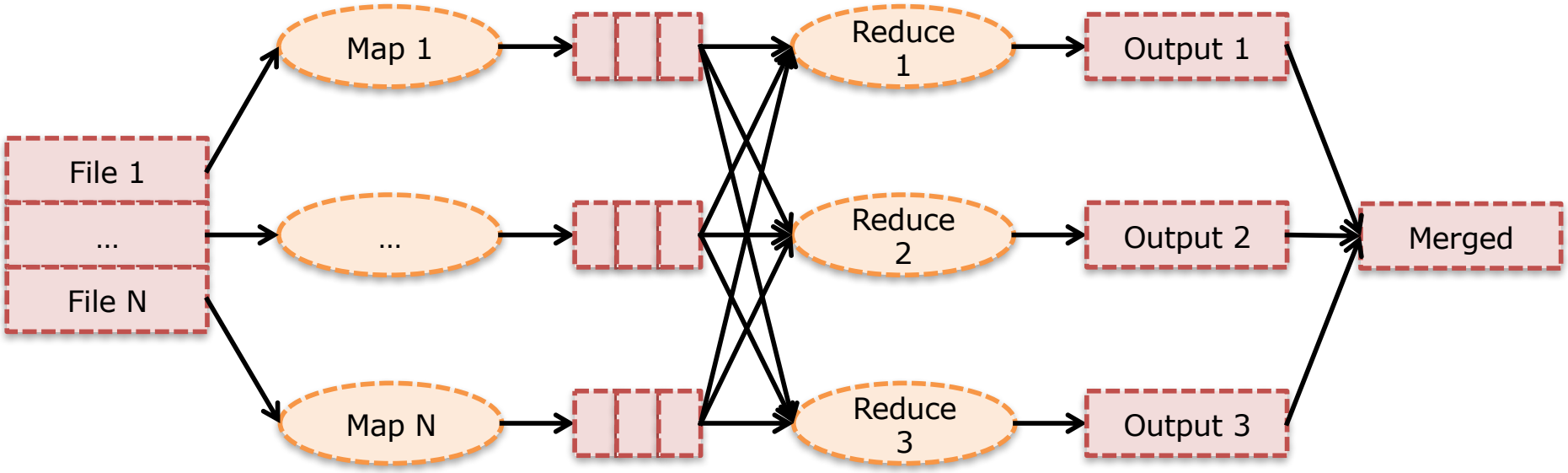


Implementation



Sequential MapReduce

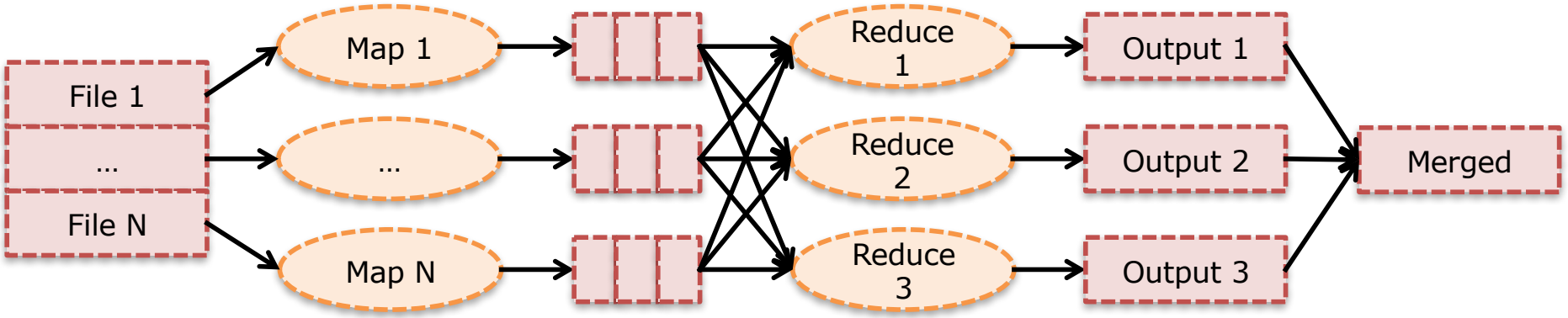
Mental Model



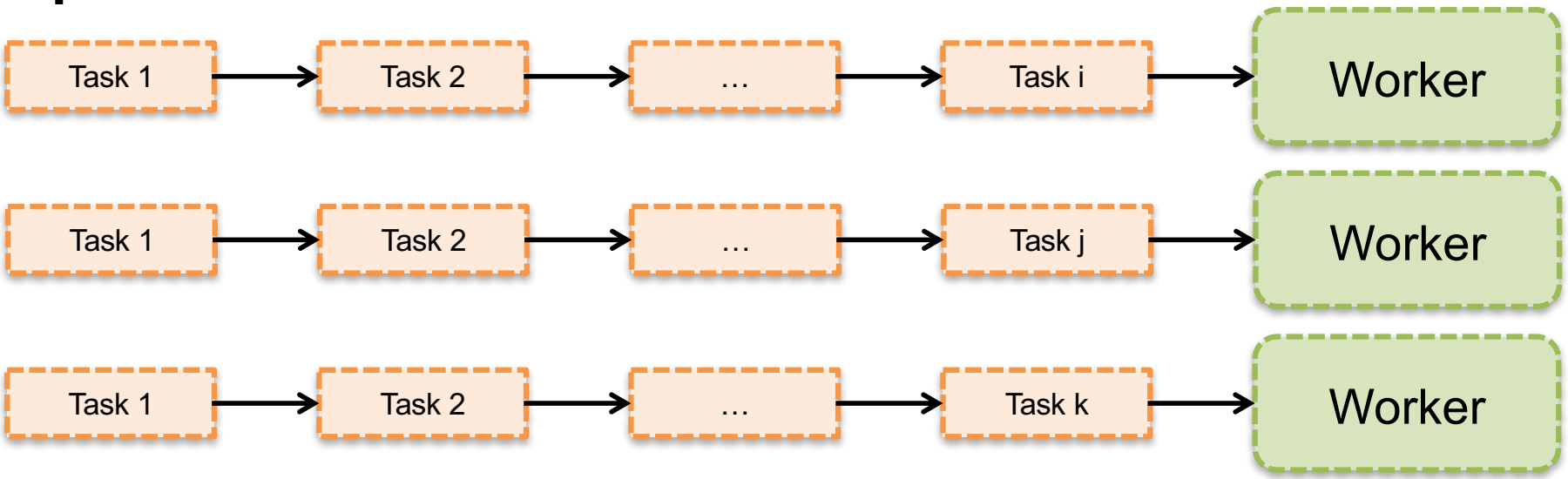
- **What if we had multiple workers that could take tasks at the same time?**
- **Do we need to coordinate the workers in any way?**

Distributed MapReduce

Mental Model

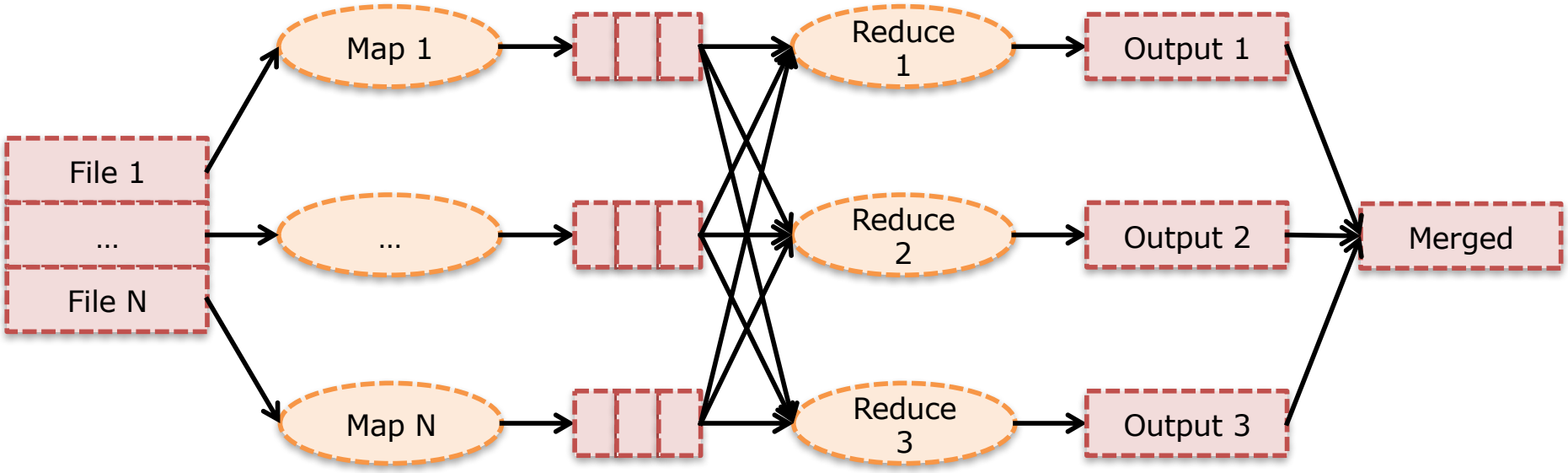


Implementation



Distributed MapReduce

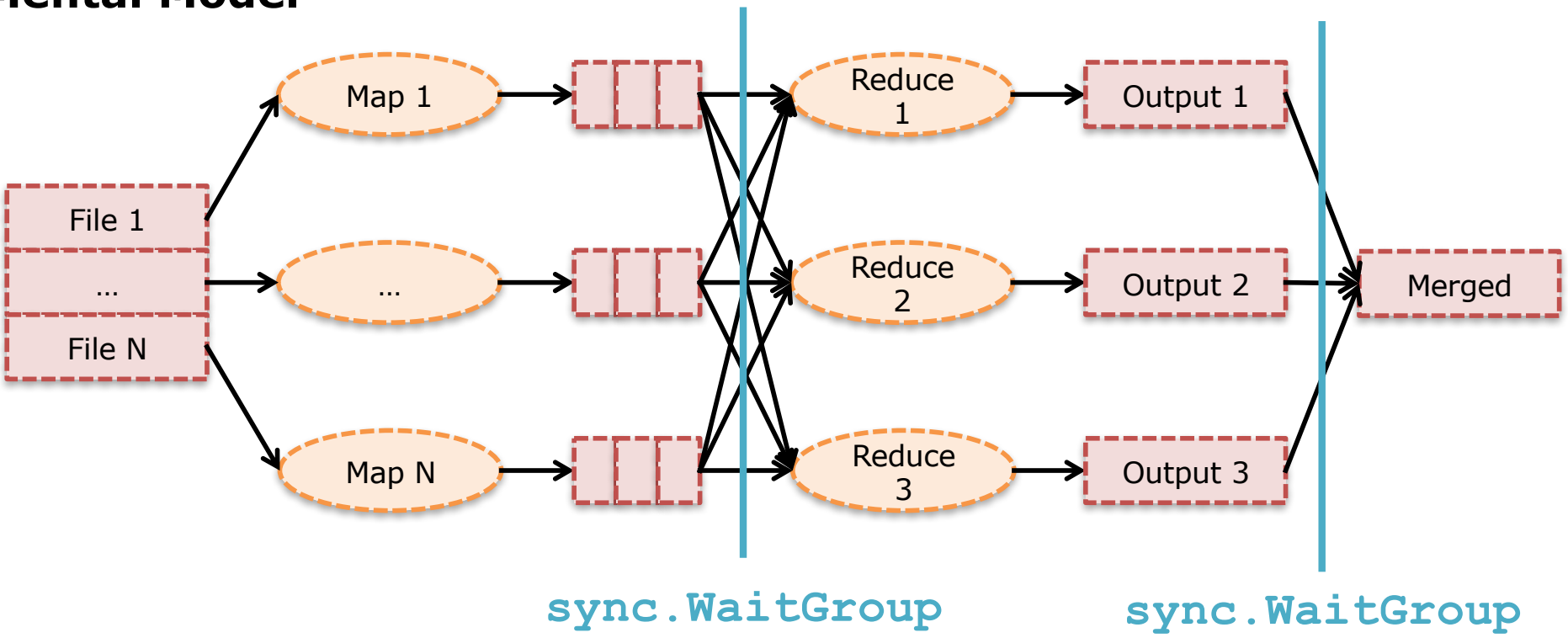
Mental Model



- **Can we run all map tasks and reduce tasks in parallel?**
- **If not, which tasks can be run in parallel?**
- **In what order?**

Distributed MapReduce

Mental Model



- **We can run all map tasks in parallel, but cannot run any reduce task until all the map tasks have finished. Why?**
- **When can we merge all the outputs from the reduce tasks?**

Distributed MapReduce

- Write a new computation (inverted index instead of word count)
- Implement a scheduler: which task should go on which worker when? How do we know:
  - What tasks to assign?
  - When workers are available?
  - Where to submit tasks?
  - When workers are done?
- Hint: the algorithm should be very similar to the sequential case, except now we have more queues

Concurrency

- In our implementation, do we need to worry about concurrency?
  - We have embarrassingly parallel computation
- What about in failure?
  - We have idempotent operation (write same file)

RPCs in Go

Reviewing RPCs

- RPCs let us execute a procedure in a different address space (e.g., on another computer) and call as though it were local

- Challenges
  - Latency
  - Network failures
  - Server failures

$$$ in 7 easy steps

1. Client calls stub (local procedure call)
2. Client stub marshals parameters
3. Client OS sends message to server
4. Server OS passes message to server stub
5. Server stub unmarshals parameters
6. Server stub calls the server procedure
7. Trace back in reverse direction

RPCs in Go (`net/rpc` server)

- Creating a server
  - Create a TCP server (or some other server to receive data)
  - Create a listener that will handle RPCs
  - Register the listener and accept inbound RPCs
- Write stub functions

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

- See https://golang.org/pkg/net/rpc/ for more details

RPCs in Go (`net/rpc` client)

- Creating a client

```
client, err := rpc.DialHTTP("tcp", serverAddress + port)
```

- Making an RPC

```
var reply int
err = client.Call("Arith.Multiply", args, &reply)
```

- Unpacking return value
  - Treat as any normal variable

Implementing an RPC client

- We are running a time server
- Goal is to implement an RPC client that uses Cristian's algorithm to get the server's clock time
- Code skeleton is available on the syllabus
- You will need need the `time` and `net/rpc` packages. Beware the difference between Time and Duration!

Cristian's Algorithm

$$ServerTime = T_3 + \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

Implementing an RPC client

```
// GetServerTime implements Cristian's algorithm
// 1. Keep track of the appropriate timestamps from the
//    local machine. Remember to get T1 close to the
//    beginning of the function!
// 2. Request T2 and T3 from the server via an RPC call
//    - serviceMethod: Listener.GetServerTimestamps
//    - args: Request
//    - reply: ServerTimestamps
// 3. Compute the server timestamp (watch out for
//    duration vs. time)

go run client.go [NetID] [ServerIP] [ServerPort]
```

Appendix

Excerpt from MapReduce 4.2

- "...When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. **The sorting is needed because typically many different keys map to the same reduce task**…

- … We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee **makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.** …"

Excerpt from MapReduce 4.2

- RPC examples
  - https://talks.golang.org/2013/distsys.slide
  - http://blog.prevoty.com/writing-your-first-rpc-in-golang
- NTP example
  - https://github.com/beevik/ntp/blob/master/ntp.go