MapReduce II, RPCs in Go

CS 475: Concurrent & Distributed Systems (Fall 2021)

Lecture 5

Yue Cheng

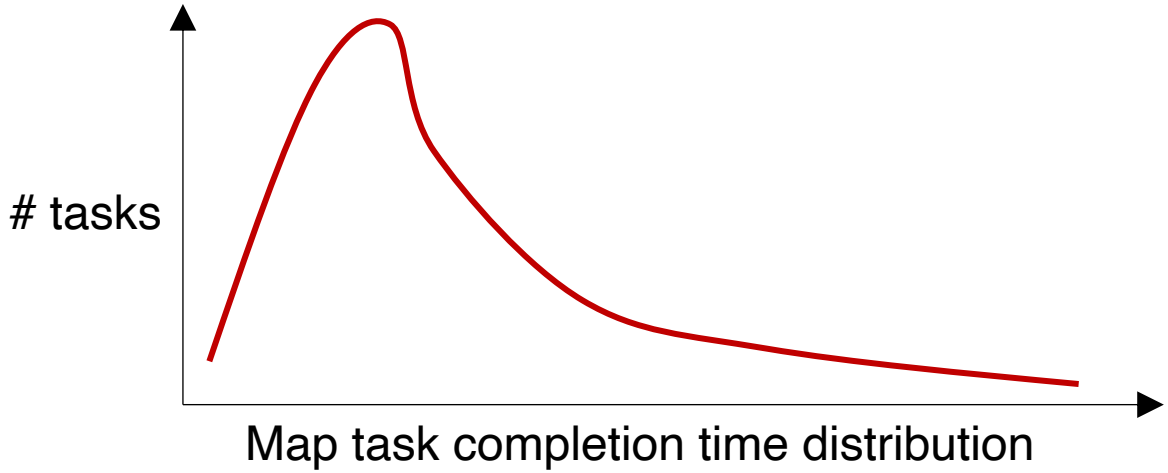
Some material taken/derived from:

- Princeton COS-418 materials created by Michael Freedman and Wyatt Lloyd.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

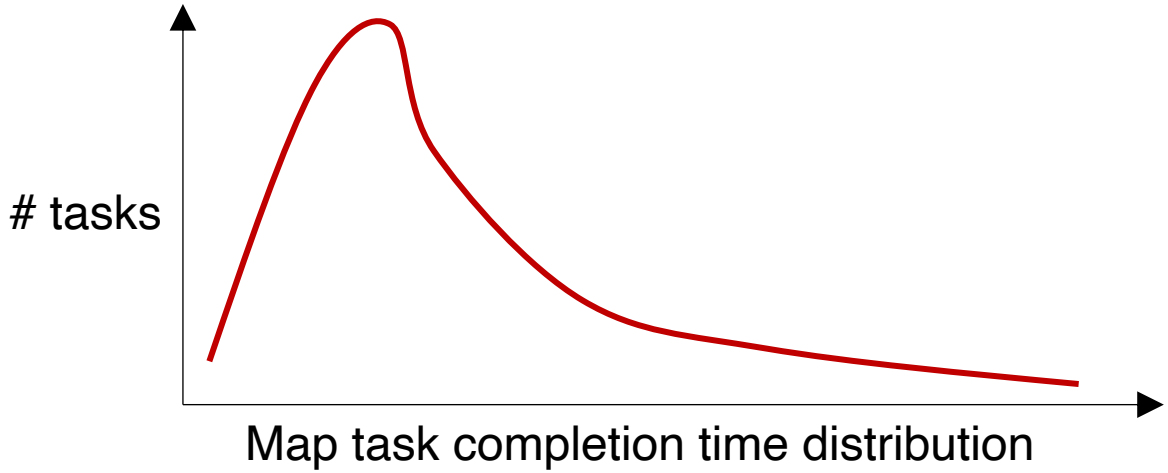
Outline

- Fault tolerance in MapReduce
- RPCs in Go
- Lab 1: MapReduce
  - Due Thursday, 09/28, 11:59pm (3 weeks)

Stragglers



Stragglers



- Tail latency means some workers (always) finish late

- Q: How can MapReduce work around this?
  - **Hint**: its approach to **fault-tolerance** provides the right tool

Resilience against stragglers

- If a task is going slowly (i.e., straggler):
  - Launch second copy of task on another node
  - Take the output of whichever finishes first

MapReduce usage statistics over time

	Aug, '04	Mar, '06	Sep, '07	Sep, '09
Number of jobs	29K	171K	2,217K	3,467K
Average completion time (secs)	634	874	395	475
Machine years used	217	2,002	11,081	25,562
Input data read (TB)	3,288	52,254	403,152	544,130
Intermediate data (TB)	758	6,743	34,774	90,120
Output data written (TB)	193	2,970	14,018	57,520
Average worker machines	157	268	394	488

* Jeff Dean, LADIS 2009

MapReduce discussion

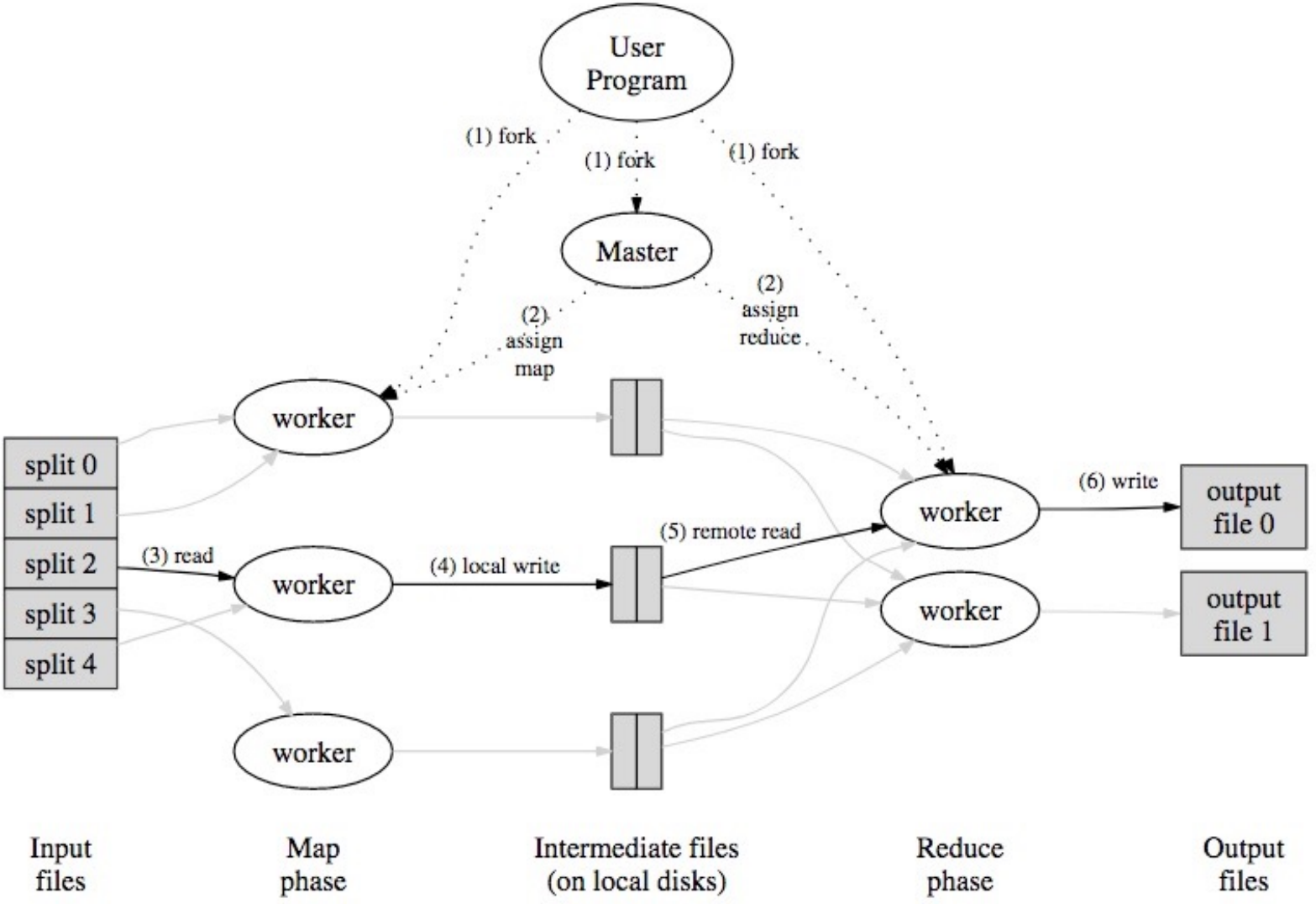
- What will likely serve as a performance bottleneck for Google's MapReduce back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

MapReduce discussion

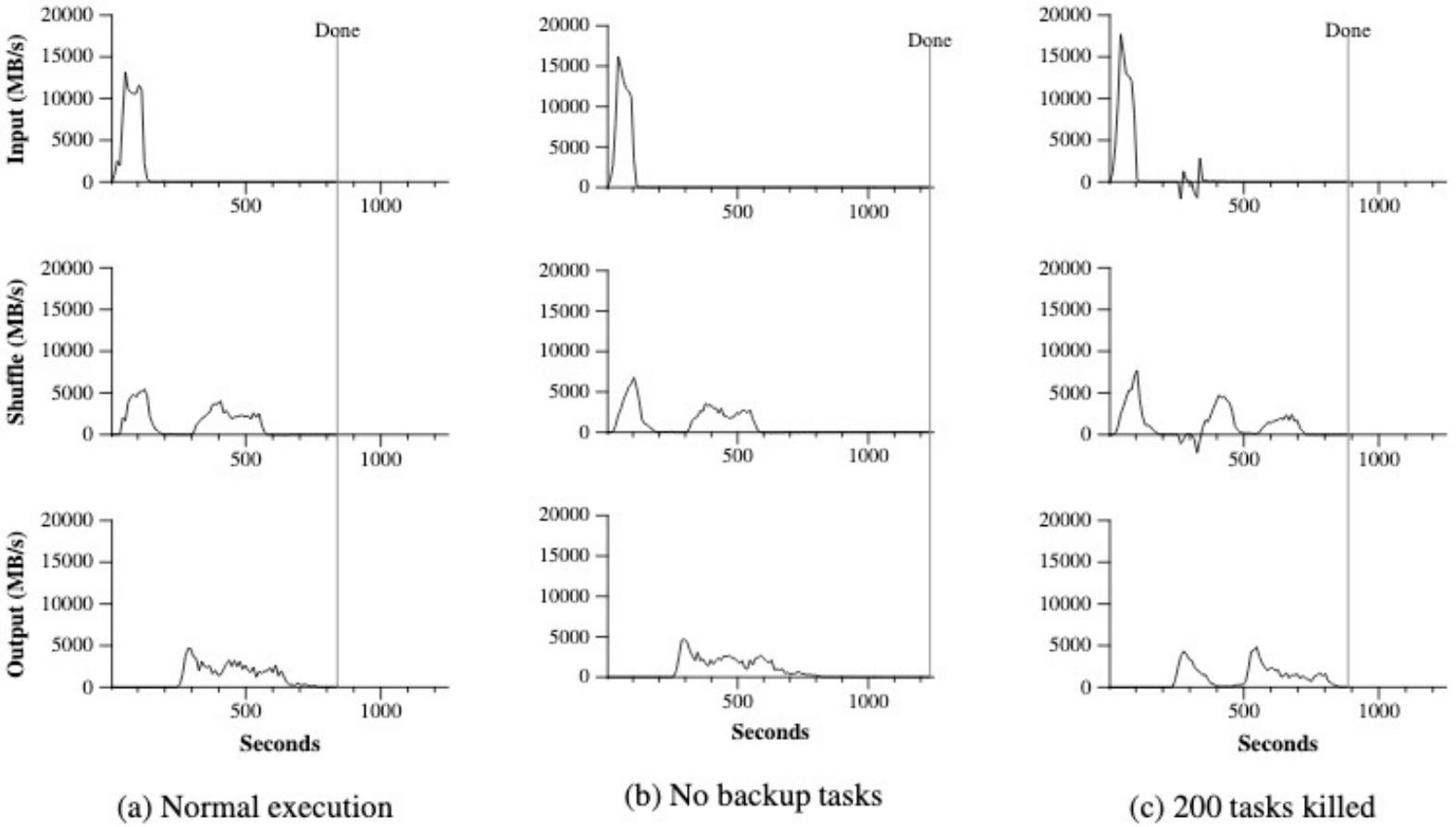
- What will likely serve as a performance bottleneck for Google's MapReduce back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

- How does MapReduce reduce the effect of slow network?

MapReduce data flows



MapReduce discussion



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

Outline

- Fault tolerance in MapReduce
- RPCs in Go
- Lab 1: MapReduce
  - Due Thursday, 09/28, 11:59pm (3 weeks)

Go RPCs

- Implementation in built-in library net/rpc

Go RPCs

- Implementation in built-in library net/rpc
- Write stub receiver methods of the form
  - **`func (t *T) MethodName(args T1, reply *T2) error`**

Go RPCs

- Implementation in built-in library net/rpc
- Write stub receiver methods of the form
  - **`func (t *T) MethodName(args T1, reply *T2) error`**
- Register receiver methods

Go RPCs

- Implementation in built-in library net/rpc
- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`
- Register receiver methods
- Create a listener (i.e., server) that accepts requests

Writing a WordCount RPC server in Go

```
type WordCountServer struct {
      addr string
}

type WordCountRequest struct {
      Input string
}

type WordCountReply struct {
      Counts map[string]int
}
```

Writing a WordCount RPC server in Go

```
type WordCountServer struct {
      addr string
}

type WordCountRequest struct {
      Input string
}

type WordCountReply struct {
      Counts map[string]int
}
```

```
func (*WordCountServer) Compute(
            request WordCountRequest,
            reply *WordCountReply) error {
      counts := make(map[string]int)
      input := request.Input
      tokens := strings.Fields(input)
      for _, t := range tokens {
            counts[t] += 1
      }
      reply.Counts = counts
      return nil
}
```

Writing a WordCount RPC server in Go

```
type WordCountServer struct {
      addr string
}

type WordCountRequest struct {
      Input string
}

type WordCountReply struct {
      Counts map[string]int
}
```

```
func (*WordCountServer) Compute(
            request WordCountRequest,
            reply *WordCountReply) error {
      counts := make(map[string]int)
      input := request.Input
      tokens := strings.Fields(input)
      for _, t := range tokens {
            counts[t] += 1
      }
      reply.Counts = counts
      return nil
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) Listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) Listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) Listen() {
	rpc.Register(server)
	listener, err := net.Listen("tcp", server.addr)
	checkError(err)
	go func() {
		rpc.Accept(listener)
	}()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) Listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client-server

```
func main() {
	serverAddr := "localhost:8888"
	server := WordCountServer{serverAddr}
	server.Listen()
	input1 := "hello I am good hello bye bye bye bye good night hello"
	wordcount, err := makeRequest(input1, serverAddr)
	checkError(err)
	fmt.Printf("Result: %v\n", wordcount)
}
```

WordCount client-server

```
func main() {
	serverAddr := "localhost:8888"
	server := WordCountServer{serverAddr}
	server.Listen()
	input1 := "hello I am good hello bye bye bye bye good night hello"
	wordcount, err := makeRequest(input1, serverAddr)
	checkError(err)
	fmt.Printf("Result: %v\n", wordcount)
}
```

```
Result: map[hello:3 I:1 am:1 good:2 bye:4 night:1]
```

Is this synchronous or asynchronous?

```
func makeRequest(input string, serverAddr string) (map[string]int, error)
{
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) chan Result {
      client, err := rpc.Dial("tcp", serverAddr)
      checkError(err)
      args := WordCountRequest{input}
      reply := WordCountReply{make(map[string]int)}









      return ch
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) chan Result {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	ch := make(chan Result)
	go func() {
		err := client.Call("WordCountServer.Compute", args, &reply)
		if err != nil {
			ch <- Result{nil, err} // something went wrong
		} else {
			ch <- Result{reply.Counts, nil} // success
		}
	}()
	return ch
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) *Call {
      client, err := rpc.Dial("tcp", serverAddr)
      checkError(err)
      args := WordCountRequest{input}
      reply := WordCountReply{make(map[string]int)}
      return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) *Call {
      client, err := rpc.Dial("tcp", serverAddr)
      checkError(err)
      args := WordCountRequest{input}
      reply := WordCountReply{make(map[string]int)}
      return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

```
call := makeRequest(...)
<-call.Done
checkError(call.Error)
handleReply(call.Reply)
```

Outline

- Fault tolerance in MapReduce
- RPCs in Go
- Lab 1: MapReduce out
  - Due Thursday, 09/28, 11:59pm (3 weeks)