Lecture 10 - Go Concurrency / Start of Smart Contracts

News

1. European Space Agency Funds Blockchain Project Recording Satellite Data
2. Accounting in space
3. 3.6B in bitcoin ceased because it was be laundered by a married couple.

Reading

Consensus Algorithms, PoW, PoS and better ones: https://medium.com/coinbundle/consensus-algorithms-dfa4f355259d

Concurrency in Go

Go routines

A Go Routine:

```
go func( a int ) {
        fmt.Printf ( "a=%d\n", a )
}( 12 )
```

10 Go Routines:

```
 1: package main
 2:
 3: import "fmt"
 4:
 5: func main() {
 6:     for i := 0; i < 10; i++ {
 7:         go func(a int) {
 8:             fmt.Printf("a=%d\n", a)
 9:         }(12 + i)
10:     }
11: }
```

With NO output - Why?

```
1: package main
2:
3: import (
4:     "fmt"
5:     "sync"
6: )
```

```
 7:
 8: func main() {
 9:     var wg sync.WaitGroup
10:     for i := 0; i < 10; i++ {
11:         wg.Add(1)
12:         go func(a int) {
13:             defer wg.Done()
14:             fmt.Printf("a=%d\n", a)
15:         }(12 + i)
16:     }
17:     wg.Wait()
18: }
```

And OUTPUT!!!!

```
a=21
a=16
a=15
a=18
a=17
a=12
a=19
a=20
a=13
a=14
```

Locks

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "sync"
 6:     "time"
 7: )
 8:
 9: func main() {
10:     messages := make(chan int)
11:     var wg sync.WaitGroup
12:
13:     // you can also add these one at
14:     // a time if you need to
15:
16:     wg.Add(3)
17:     go func() {
18:         defer wg.Done()
19:         time.Sleep(time.Second * 3)
20:         messages <- 1
21:     }()
22:     go func() {
23:         defer wg.Done()
24:         time.Sleep(time.Second * 2)
25:         messages <- 2
26:     }()
27:     go func() {
```

```
28:         defer wg.Done()
29:         time.Sleep(time.Second * 1)
30:         messages <- 3
31:     }()
32:     go func() {
33:         for i := range messages {
34:             fmt.Println(i)
35:         }
36:     }()
37:
38:     wg.Wait()
39:     time.Sleep(time.Second * 1)
40: }
```

Output:

```
3
2

m4_ omment(
3
2
1
)
```

Also “map”’s are not concurrency protected. You have to lock/unlock them yourself.

Problems like this are easy to find. There is a “race detector” built into go and you can run it as a part of your tests.

You should decide if you need to protect a map. Why? When?

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "math/rand"
 6:     "sync"
 7:     "sync/atomic"
 8:     "time"
 9: )
10:
11: func main() {
12:     state := make(map[int]int)
13:     mutex := &sync.Mutex{}
14:
15:     var nRead uint64
16:     var nWrite uint64
17:
18:     const randRange = 15
19:
20:     for ii := 0; ii < 100; ii++ {
21:         go func() {
22:             total := 0
23:             for {
24:                 key := rand.Intn(randRange)
25:                 mutex.Lock()
26:                 total += state[key]
27:                 mutex.Unlock()
```

```
28:                 atomic.AddUint64(&nRead, 1)
29:                 time.Sleep(time.Millisecond)
30:             }
31:         }()
32:     }
33:     for jj := 0; jj < 50; jj++ {
34:         go func() {
35:             for {
36:                 key := rand.Intn(randRange)
37:                 val := rand.Intn(100)
38:                 mutex.Lock()
39:                 state[key] = val
40:                 mutex.Unlock()
41:                 atomic.AddUint64(&nWrite, 1)
42:                 time.Sleep(time.Millisecond)
43:             }
44:         }()
45:     }
46:
47:     time.Sleep(time.Second * 1)
48:
49:     nReadTotal := atomic.LoadUint64(&nRead)
50:     nWriteTotal := atomic.LoadUint64(&nWrite)
51:
52:     mutex.Lock()
53:     fmt.Printf("ReadOps: %d\nWriteOps: %d\nFinal State: %+v\n", nReadTotal, nWriteTotal, state)
54:     mutex.Unlock()
55: }
```

The Output (run twice - it will produce non-deterministic output!):

```
$ go run atomic.go
ReadOps: 81881
WriteOps: 40936
Final State: map[10:70 3:81 9:81 12:55 5:67 1:38 6:89 14:28 0:40 8:13 4:11 13:19 2:40 11:23 7:30]
$ go run atomic.go
ReadOps: 82500
WriteOps: 41250
Final State: map[2:34 10:2 4:28 5:80 14:42 0:46 3:55 1:65 12:63 9:10 13:50 7:17 6:19 11:91 8:14]
```

Channels

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "os"
 6:     "sync"
 7:     "time"
 8: )
 9:
10: func main() {
11:     msg := make(chan string)
12:     msg2 := make(chan string)
13:     var wg sync.WaitGroup
14:     for i := 0; i < 10; i++ {
15:         wg.Add(1)
```

```
16:         go func(n int) {
17:             for {
18:                 time.Sleep(time.Millisecond * 50)
19:                 msg <- fmt.Sprintf("ping:%d", n)
20:             }
21:         }(i)
22:     }
23:     for i := 0; i < 10; i++ {
24:         wg.Add(1)
25:         go func(n int) {
26:             for {
27:                 time.Sleep(time.Millisecond * 55)
28:                 msg2 <- fmt.Sprintf("PONG:%d", n)
29:             }
30:         }(i)
31:     }
32:     nMsg := 0
33:     for {
34:         select {
35:         case out := <-msg:
36:             nMsg++
37:             fmt.Printf("%s\n", out)
38:         case out := <-msg2:
39:             nMsg++
40:             fmt.Printf("%s\n", out)
41:         }
42:         if nMsg > 20 {
43:             os.Exit(0)
44:         }
45:     }
46:     wg.Wait()
47: }
```

Output (Again - non-deterministic output!):

```
ping:2
ping:7
ping:6
ping:9
ping:4
ping:3
ping:0
ping:8
ping:1
ping:5
PONG:2
PONG:0
PONG:3
PONG:6
PONG:9
PONG:4
PONG:5
PONG:8
PONG:7
PONG:1
ping:1
```

Consensus

The original Byzantine Fault Tolerance is from research done by Leslie Lamport in 1978. This was revolutionary because it was the first look at actual failures in control systems for flight. Lamport used formal methods to prove that faiure was not just, "it quit working" but that falure could also be "it became malicious and it lied". The solution was not really a practical one.

That leads us to: Practical Byzantine Fault Tolerance (pBFT) is a consensus algorithm developed in 90s by Barbara Liskov and Miguel Castro. They took a new look at (BFT) and figured out how to do it with a much better communication and performance and how to implement it so it can be made to work. Lamport had gone on to develop a system that worked with clocks syncronizing nodes - and this is the basis of Googles F1 database. pBFT works on an asyncronous network with no upper bound on when the requests will be received. It is optimized for low overhead. It is used in important real world systems like AirBus, probes sent to Mars and in my Mersadies Sprinter Van. It also happens to be at the heart of concensus for Bitcoin.

Concensus is when a set of nodes that are distributed reach a common agreement on the "state" of the world. This will not be taken to mean that they reach the "correct" or "truthful" state of the world. It only means that they reach an "agreement" that the world is in "a" state.

BFT's can aggree even when some of the nodes fail to repond or lie about a response. The objective of BFT is to safeguard agains the failures in the system by applying a collective decision making process that reduces the risk of faulty nodes. BFT is a solution to the Byzantine Generals' Problem.

Byzantine Generals' Problem

From the paper by Leslie Lamport, Robert Shostak and Marshall Pease at Microsoft Research in 1982:

"Imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching an agreement. The generals must decide on when to attack the city, but they need a strong majority of their army to attack at the same time. The generals must have an algorithm to guarantee that (a) all loyal generals decide upon the same plan of action, and (b) a small number of traitors cannot cause the loyal generals to adopt a bad plan. The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition (a) regardless of what the traitors do. The loyal generals should not only reach agreement, but should agree upon a reasonable plan."

A timeout with a default vote can be added to the system - and this is often done. This puts a time bound on getting messages and what happens when that time bound is reached. It also tends to hide nodes that have permanently disapeard.

Lamport proved that with `3m+1` working processors a concensus on state can be calculated if at most `m` systems are faulty. This implies that two thirds, `2/3` , of the number of computers should be properly working.

There are a bunch of failure modes:

1. no reponce
2. stuck responce (the all 1's case, or continuous lies)
3. responce with an invalid or incorrect result
4. deliberate lies

Bitcoin is using a Proof-of-Work system that is much closer to Lamport's 1978 paper. pBFT has a number of advantages.

1. Finality is achieved immediatly. Bitcoin will not finalize a transaction until 6 more blocks are added - and at a rate of 10min per block this is an hour.
2. in pBFT all active nodes participate in voting. This means that it is not a race - it is a common goal. Rewards in the systme can be distributed evenly - instead of one winner.

3. Much more enegrgy effecient. No PoW - Zilliqa runs a PoW only once in every 100 blocks - instead of on every block. Veification of blocks is not done like Bitcoin on every block every time before adding blocks.

pBFT approach.

pBFT a leader node is chosen. If no leader is availabe the system can vote for a new leader. All non-leader nodes are called Secondary or Backup nodes.

A vote will be taken by sharing blocks where a majority of total nodes results in choosing the correct result. Usually when a majority is achieved - the non-majority is then ignored. IF youhave 5 notes, and 3 say "state is X", then the other 2 can now be disregarded. They may-or-may-not aggee - but concensus has been achieved.

There are 4 phases in the pBFT consensus system (from the pBFT paper):

1. The client sends a request to the primary(leader) node.
2. The primary(leader) node broadcasts the request to the all the secondary(backup) nodes.
3. The nodes(primary and secondaries) perform the service requested and then send back a reply to the client.
4. The request is served successfully when the client receives 'm+1' replies from different nodes in the network with the same result, where m is the maximum number of faulty nodes allowed.

Security increases as you have more nodes. Communication overhead increases at a `k**n` rate - so cost of communication grows Rapidly - making large networks very expensive.