Saddle Ranch

Centralized p2p is possible (some coordinating server(s)) - BitTorrent - Cassandra (distrib. Database) - Gossip model - Each node is client and server - Can also share disk space/cpu etc between nodes

NETWORK BANDWIDTH SHARING? - piggy-backing - Node A send to node Z need to pass nodes so use their bandwidth OVERLAY NETWORK: logical network fully connected (imagined) NOT physically (super-imposed on top of actual physical connections) like abstraction - mirror casting, Bluetooth, etc. use P2P but low lvl (not app layer) - Late 90s: Napster pirating songs. - Search by name and color code (green for nearby, red for faraway) and download mp3 - Spotify: not on device just streaming Deaspara, Tour?- “you own your data” octal network to pass if you want to share, P2P instead d of Facebook

Tim Burtons Lee Wikipedia is client-server to edit pages etcP2P

P2P = out of admin zone clients, (not physically connected to architecture) I.e. data center is admin zone (specialized) so we connect to it. So even if data center looks like P2P, not count (its okay lots of gray areas) AZ = admin zone (see 18:19)

STRUCTURED vs UNSTRUCTURED (lecture 2) - structured: i.e. binary tree, ring, brick. For data indexing i.e. BitTorrent, Distrib. Data stores/databases - indexing to find correct nodes/files - “Distributed hash table” DHT: assigns ownership to particular node in system - Lives across network, replicated - Key (file name) to specific identifier hash to consistent hash alg (sha1), rendezvous - 22:09 see ex modulo etc - Breaks down in nodes that left but okay with new nodes - Possibly exit strategies: syncronizes with everyone and exits - Or data already replicated across nodes so fault tolerant, use one of the others - Not always necessary ie BitTorrent - only offers files already on your system, not absorbing from others. Only necessary in distrib. Data system. - Updates to table are broadcast over network so everyone updates their DHT - Layered is possible - looking for which layer data lives in - To retrieve files? - List of neighboring nodes - Get node closest to key to retrieve (i.e. B+ tree in database, or i-Sam) - Advertising: $$$$$ “advertise” myself and find new nodes, others constant probing for new nodes. - Discovery: “who can be my friend”? 30:00 - Routing to network? (Link to paper) - Can cord (ring) w/ shortcuts - Capestry - Pastry

UNSTRUCTURED P2P - Napster, Kuzzah, Nutella, file sharing apps - Auto ad hoc node joining (probabilistic existence) - Brute force NOT DHT only - Broadcast request to all neighbors - If yes, returns to requester - If no, forward to own neighbors - Then create connection directly back to requester - OR, propagate msg to forwarder (send to each node along path) - Drops if seen request before (to avoid cycles) - TTL: # max hops willing to find request, otherwise drop - Or, random walk. (Randomly pick a neighbor, which randomly picks a neighbor, etc until finds) - Sends msg back without random walk - Less network traffic w/ trade off for slower. if N neighbors each time, reduce speed by factor of n each hop - Random walks in parallel? Pick K random neighbors to start and - Good for others, Markov CHain and other complex problems (i.e. ML) - Separate overlay : “SUPER PEERS” more beefy nodes w/ bigger indexes - Send request to super peer first (subsystem) to find key and find lower layer - Then use flooding or random walk - Nested P2P essentially - BitTOrrent - If super peer dies, needs new one (promote weak one?) - Vote (Leader Election) algorithms i.e. Ralf

CONCURRENCY & GO

- Why Go? Not C++? Concurrency built in, threads and RPC, garbage collection
- Manual challenging problem to figure out if okay to dealloc
- Type safety mem safety
- Weird: OOP unlike Java/Python
- Compiled static strong typing so not worry crashes in threads
    - ORDER INDEPENDENT tasks: in overlapping period of time (within windows != parallelism)
    - Concurrency != parallelism
        - Concurrency: model of programming that DEALS simultaneously
            - EX: soup or salad (not and), one shared resource chef. He contact

switches between stir soup and chop salad.
        - EX: chop all the salad, stir all the soup. Less efficient/burning?
    - Parallelism: that DOES simultaneously
        - I.e. threads scheduled across multiple processors/cores
        - EX: Firefox and chrome unrelated tasks running, multiple subpieces of Job (subtask)
        - EX: each chef cooking subtasks simultaensously and then put all together
        - EX: one chef ONLY chops salad. Another ONLY stirs soup. Simultaneously. Is it concurrent? NO
        - EX: cook garnish (3 min) and cook duck (30 min).
            - Gordon cooks duck, 27 min later calls another chef to cook garnish, boom plate
            - Start same time, garnish burns

- each person cooking independent tasks (veggie crew, dessert crew, starter, etc.) = parallelism
- Costco: one cashier, two lines, toggles btwn lines on shared resource, not same time concurrency
- McDonald's multiple cashiers, parallelism not shared resource (exact same time)
    - course mostly concurrency

GO

THREAD NOT IMPLEMENTED AS OS THREAD, runtime inherent in code (i.e. JVE java virtual runtime) GO routine != threads but simpler to imagine as so Each thread has stack, address space shared, program counter, set of registers

Start: to start thread Exit: finish execute or crash Stop: - if thread blocked/waiting, pause thread so runtime schedules another while it waits - Resource contention to pause? Resume: resumes after stopping

Use of threads 1. IO concurrency (for Network/Disk IO) 2. Multi-core parallelism (if multiple cpus) 3. Convenience (book keeping, managing thread execution, logistics) 1. Fire separate thread to time the IO thread to sleep. 2. If second timeout thread exits first, first thread timed out No limit to GO threads, will execute whenever possible. - costs memory - Initializing overtime but less than OS threads

Examples

[[8cc4678b5a2389211377f9d35836ca67_MD5.jpeg|Open: Pasted image 20241020094110.png]] ![[8cc4678b5a2389211377f9d35836ca67_MD5.jpeg]] go some func; see squares.go iterate slice of integers

Thread challenges

Go Channels

for solving race condition: - Lock: for same memory location, one thread accesses at a time) - vs **Go Channel:** data stored in channel, thread requests and updates channel - if channel empty then pulling thread blocks - if channel full (1) then pushing thread blocks

```
package account

type Account struct {
    balance chan int // channel for a bank account
}
func NewAccount(init int) Account {
    a := Account{ balance: make(chan int, 1)} //inits account and channel
    a.balance <- init //pushes init to channel
    return a //returns new account
}

func (a *Account) CheckBalance() int { //method receiver to update a
    bal := <-a.balance //POPS balance from channel
    a.balance <- bal   //PUSHES same balance back to channel
    return bal
}
func (a *Account) Withdraw(v int) {
    bal := <-a.balance
    a.balance <- (bal - v)  // -$v
}
func (a *Account) Deposit(v int) {
    bal := <-a.balance
    a.balance <- (bal + v) // +$v
}
```

for coordination/communications between threads:

```
result := make(chan int, numWorkers)
    // or items

// Launch workers
for i := 0; i < numWorkers; i++ {
    go func() {
        doWork()
        completeFlag <- i //push something to channel to report completion
    }()
}

// Wait for all worker threads to finish
for i ;= 0; i < numWorkers; i++ { // pull completeFlag to check all done
    handleResult(<-result)
}
fmt.Println("Done!")
```

select: waiting multiple channels

- coordination: dependent threads
- ex: mom and dad channel

```
select { //non-blocking: default if no money in channel
    case money := <-dad:
        buySnacks(money) //if money in dad channel, pull and buy snacks
    case money := <-mom:
        buySnacks(money) //if money in mom channel, pull and buy snacks
    case default: //if nobody has money, starve.
        starve()
        time.Sleep(5 * time.Second)
}
```

vs no default: if neither has money, thread blocks until has money if both have money, randomly chooses one to execute first.

```
select { //non-blocking: default if no money in channel
    case money := <-dad:
        buySnacks(money) //if money in dad channel, pull and buy snacks
    case money := <-mom:
        buySnacks(money) //if money in mom channel, pull and buy snacks
}
```

can be inside for loop

timeout threads + select

two independent threads

```
// two channels
result := make(chan int)
timeout := make(chan bool)

// Ask server (Goroutine 1)
go func() {
    response := // ... ask, network, rpc, etc.
    result <- response // BLOCKS until get response
}()

// Start timer (Goroutine 2)
go func() {
    time.Sleep(5 * time.Second)
    timeout <- true // Pushes timeout flag to channel
}()

//main:
select {
    case res := <-result: //checks/stores result
        processResult(res)
    case <-timeout: // checks if something in timeout channel
        fmt.Println("Timeout!") //automatically exits main and kills
threads
}
```

deadlock:

- two threads waiting for each other

```
package main
import (
    "fmt"
)
func main() {
    c1 := make(chan int)
    fmt.Println("push c1: ")
    c1 <- 10 //pushes to channel, but nobody reads it so blocked
    g1 := <- c1
    fmt.Println("get g1: ", g1)
}
```

deadlock.go on BruinLearn:

```
func main() {
    //init channels
    ch1 := make(chan string)
    ch2 := make(chan string)

    //init and runs threads
    go func() {
        ch1 <- "Some data from Goroutine 1" //waits for someone to read
ch1
        msg := <-ch2
    }()
    go func() {
        ch2 <- "Some data from Goroutine 2" //waits for someone to read
ch2
        msg := <-ch1
    }()

    select {}
    // without select, main exits immediately and kills routines
    // even with just select, get deadlock
}
```

optional: BUFFERED channels (i.e. semaphones, etc.)

- not buffering means blocks if more than one pull/push

Vote nodes (i.e. leader election)

vote-count-1.go

```go
package main

import (
    "time"
    "math/rand"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    count := 0 // # votes for referendum
    finished := 0 //total # votes completed
    for i := 0; i < 10; i++ { // 10 goroutines
        go func() { //random order
            vote := requestVote() //yes/no
            if vote { count++ }
            finished++ //incremented successful vote
        }()
    }

    for count < 5 && finished != 10 {
        // wait until majority or all threads voted
    }

    if count >= 5 {
        println("received 5+ votes!")
    } else {
        println("lost")
    }
}

func requestVote() bool { //ret random yes/no
    time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
    return rand.Int() % 2 == 0
}
```