

Distributed Systems

Introduction and Overview

Utah Tech University—Department of Computing

Spring 2025

Overview

What is a distributed system?

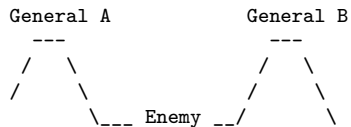
- Mostly an academic subject until the late 90s, now a standard part of building software
 - Mobile, web, and desktop apps
 - Servers
 - 3-tier apps, peer-to-peer, client-server, cluster

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable” - Leslie Lamport

What makes distributed systems interesting?

The Two Generals' Problem

Two generals are planning an attack on a city. They communicate by sending messengers who may be captured or delayed. How can they reach agreement on the time to attack?



RPC

A standard way of communicating between nodes is to use *Remote Procedure Calls* (RPC), which emulate regular procedure calls across the network

```

result = getAccount(node, user)
|
+--> getAccount client stub:
    * serialize arguments
    * send network message to node-->
        --
        .-, (  ), -.
        .-(          )-.
        '-(          ), -'
        '-.( ) .-'
        network
        <-----+
        * deserialize results
        * return from client stub
        |
result = ... <--+

        <-----+
        * deserialize results
        * return from client stub
        |
        +----> result = getAccount(user)
        |
        |
        |
        * serialize results <-----+
        * send network message back
        |
        .-, (  ), -.
        .-(          )-.
        '-(          ), -'
        '-.( ) .-'
        network
        <-----+
    
```

Distributed systems

Distributed systems are different from single-node systems in a few important ways. Consider the differences between an RPC and a regular function call:

- Latency
 - Data centers?
 - Across the world?
 - How much for local calls?
 - Can't we just wait for the network to get faster?
 - Case study: SQLite vs client-server databases
 - Requires rethinking call graph, but hard to miss and easy to plan for

Distributed systems

Distributed systems are different from single-node systems in a few important ways. Consider the differences between an RPC and a regular function call:

- Memory access
 - What does a pointer mean?
 - Why do we use pointers?
 - Efficiency (less copying)
 - Shared data structures (shared changes)
 - Recursive data structures (trees, graphs, loops)
 - Can we just make copies?
 - Concurrency
 - What about file handles, locks, etc.?
 - Requires rethinking data flow, but hard to ignore

Distributed systems

Distributed systems are different from single-node systems in a few important ways. Consider the differences between an RPC and a regular function call:

- Partial failure and concurrency
 - Fundamentally different from fail-stop model
 - Consider each step that can fail in a simple RPC
 - Do we know if it failed?
 - Is it just slow?
 - We often choose a distributed system for fault tolerance and availability
 - The defining problem of distributed systems
 - Easy to miss, hard to think about
 - Requires rethinking every part of the system
 - For many architectures, you must start with the distributed systems problems and then fill in everything else

Attendance, distractions, etc.

- Attendance is expected every day, and is required on paper discussion days
 - Excessive absense without making arrangements will result in failing (see the syllabus)
- You are responsible for what we talk about in class, and much of what we cover will *not* be available elsewhere
 - Assignment instructions, tips, etc.
 - If you miss class, you may not be able to complete the homework
- You are expected to take notes: bring pen and paper
- Laptops and mobile devices are not allowed in class unless specifically called for
 - Not even for notes or following along with demos
 - Exceptions need documentation
- Make-up policy for projects
 - no make-up for Go basics: DO NOT FALL BEHIND
 - no make-up for readings: must read and participate in discussions

CodeGrinder

You should have a Linux (including WSL) or Mac OS environment to work on

- We will use CodeGrinder for autograding many assignments, especially early ones
- I recommend installing Debian 12 (Bookworm) if using WSL
- First steps: install CodeGrinder and Go

Learning Go

We will spend the first few weeks doing Go practice exercises

Philosophy:

- To learn a language, you need practice
- You need to practice every day (sleep between)
- A bunch of short sessions is better than a long session

Plan to complete one CodeGrinder exercise every single day (except weekends). Each exercise is 2 or 3 problems.

We will still only touch on many important parts of the language. I recommend a book:

The Go Programming Language
by Alan Donovan and Brian Kernighan

It costs about \$30 and is well worth it. Start by reading the first few chapters, then plan to read an additional chapter every once in a while to deepen your understanding of a topic.

Reading papers

A major part of this class is reading research papers that focus on real systems

Reading research papers is hard work and takes a long time. Do not underestimate this part.

Papers are on Wednesdays:

- I will assign groups and send out discussion questions in advance
 - We will spend most of Wednesday discussing the paper—come prepared to discuss the entire paper and especially your assigned questions
 - No make up for papers—do not forget!
- The smart approach: study group before Wednesday to work through the big picture
- You will probably learn more from reading and discussing than from anything else we do
- Most of the projects will be based on implementing systems we read about (Paxos, Chord, MapReduce)

Hello, world

To set up Go and vim: see screencast on course page

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

Building and running:

- `go mod init`
- `go build`
- `go install`
- `go fmt` and `goimports`

Command-line arguments

```
// Echo1 prints its command-line arguments
package main

import (
    "fmt"
    "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

More about for loops

There are a few forms of `for` loops in Go:

```
// A C-style "for" loop  
for initialization; condition; post {  
    // zero or more statements  
}
```

```
// a "while" loop  
for condition {  
    // body  
}
```

```
// an infinite/"forever" loop  
for {  
    // body  
}
```

Range

// Echo2 prints its command-line arguments.

```
package main
```

```
import (  
    "fmt"  
    "os"  
)
```

```
func main() {  
    s, sep := "", "  
    for _, arg := range os.Args[1:] {  
        s += sep + arg  
        sep = " "  
    }  
    fmt.Println(s)  
}
```

// variations on declaring variables:

```
s := ""  
var s string  
var s = ""  
var s string = ""
```

// this can create a lot of garbage:

```
//  
// set += sep + arg
```

// a better way: use the standard library

```
func main() {  
    fmt.Println(strings.Join(os.Args[1:], " "))  
}
```