Erlang and Go

(CS262a, Berkeley Spring 2018)

Philipp Moritz

The Problem

Distributed computation is hard!

- State
  - Hard to do recovery, dependency on order of execution
- Concurrency and Synchronization
  - Hard to reason about, deadlocks
- Fault handling
  - Error origin and handling in different parts of the system
- Complexity of coupled components
  - Makes it hard to develop software in large teams

This lecture is about languages/tools that make it easier to write distributed programs

What makes a good API?

- Easy to use
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements

How to Design a Good API and Why it Matters

Erlang

- Developed at Ericson as a **proprietary language** to improve development of **telephony applications**
- High availability of **nine “9”s** (30 ms downtime per year)
- 1986: initial version in **Prolog**
- 1992: BEAM (High performance VM)
- since 1998: **Open source**

named after **Agner Krarup Erlang,** mathematician and inventor of queuing theory

Erlang: Requirements

- Designed for telecommunication systems
- Hard requirements:
  - High degree of concurrency
  - Distributed
  - Soft real-time capabilities
  - 100% availability
- Soft requirements:
  - Hot swapping code

Erlang: Philosophy

Requirements

- 100% availability

Decision

Erlang: Philosophy

Requirements

- 100% availability

Decision

- strong fault recovery
  - hierarchy of tasks for recovery
  - isolation between tasks
  - dynamic code upgrade

Erlang: Philosophy

Requirements

- 100% availability
- **portability (e.g. to embedded devices)**

Decision

- strong fault recovery
  - hierarchy of tasks for recovery
  - isolation between tasks
  - dynamic code upgrade

Erlang: Philosophy

Requirements

- 100% availability
- **portability (e.g. to embedded devices)**

Decision

- strong fault recovery
  - hierarchy of tasks for recovery
  - isolation between tasks
  - dynamic code upgrade
- **agnostic to OS**
  - **green processes**
  - **doesn’t use OS services**

Erlang: Philosophy

Requirements

- 100% availability
- portability (e.g. to embedded devices)
- **high concurrency**
  - **soft real-time**

Decision

- strong fault recovery
  - hierarchy of tasks for recovery
  - isolation between tasks
  - dynamic code upgrade
- agnostic to OS
  - green processes
  - doesn’t use OS services

Erlang: Philosophy

Requirements

- 100% availability
- portability (e.g. to embedded devices)
- **high concurrency**
  - **soft real-time**

Decision

- strong fault recovery
  - hierarchy of tasks for recovery
  - isolation between tasks
  - dynamic code upgrade
- agnostic to OS
  - green processes
  - doesn’t use OS services
- **very lightweight processes, communicate via channels**
  - **share nothing**
  - **asynchronous calls**

Design Tradeoffs

Performance **Security/Isolation**

Erlang is a safe language (cf. SPIN)

- fast IPC (same address space)
- isolation via language semantics

Concurrency **Maintainability**

Decoupling components with “Share nothing” semantics

Erlang: Error handling

- Crash early
- Let some other process do the error recovery
- Do not program defensively
  - If you cannot handle the error, don’t try to recover

Erlang: Concurrency

- **Distributed actor model** (asynchronous message passing)
- Exposed via spawning processes and asynchronous message passing between them

```
% invoke web:start_server(Port, MaxConns)
 ServerProcess = spawn(web, start_server, [Port,
MaxConns]),

 % invoke web:start_server on machine RemoteNode
 RemoteProcess = spawn(RemoteNode, web, start_server,
[Port, MaxConns]),

 % Send a message to ServerProcess (asynchronously).
 ServerProcess ! {pause, 10},

 % Receive messages sent to this process
 receive
         a_message -> do_something;
         {data, DataContent} -> handle(DataContent);
         {hello, Text} -> io:format(...);
         {goodbye, Text} -> io:format(...)
 end.
```

Erlang: Example

Server client example

Erlang: Implementation

- Green processes (can launch millions of them)
    - mapped to OS threads
    - Support priorities
- Preemptive scheduler (every ~2000 function calls)
    - native C code needs to be instrumented to pass control to VM
    - IO threads to handle blocking IO
- robust and well tested
    - has been used in critical infrastructure by multiple companies
    - minimal dependence on OS
- https://github.com/erlang/otp

https://hamidreza-s.github.io/erlang/scheduling/real-time/preemptive/migration/2016/02/09/erlang-scheduler-details.html

Does Erlang achieve its goals?

Erlang: Impact

- Highly commercially successful in telecom industry
  - Ericson
  - Nortel
  - T-Mobile
- WhatsApp
- Facebook chat (200 Mio users)
- Elixir
- RabbitMQ

Go: History

- Started as an experiment at Google to design a language that would solve challenges that come up in large scale software development
- First appeared 2009, first stable release in 2011

Rob Pike, co-creator of Go

Go: Motivation

- Developing large software components with large team is hard
  - Slow builds
  - Dependencies and libraries
  - Complex languages, everybody uses a different subset
- Developing distributed software is even harder
  - Concurrency not natively supported by many existing languages

Influence from Plan 9

- Plan 9 from Bell Labs:
  - **Everything** is a file
  - Special C dialect:
    - No recursive #includes
    - Unicode
  - Distributed (byte oriented protocol 9P to exchange data between nodes)
- Compiler infrastructure shared

Standardization and Tooling

- **One** way to do things (cf. Python)
- Standardized tooling:
  - **go get**: Package manager integrated with the language and github
  - **go fmt**: Put code into a standard format
  - **go test:** Unit testing and microbenchmarks
  - **go vet**: Static analysis and linting
  - **go fix**: Automatically update APIs and language constructs
- Statically linked binaries

Simplicity

- **Few concepts that are orthogonal and composable:**
  - Concurrency
    - Goroutines (execution)
    - Channels (communication)
    - Select (coordination)
  - Object oriented programming
    - Interfaces (contracts)
    - structs (data)
    - functions (code)
- No Templates/Generics (instead: interface {})
- No exceptions
- One type of loop

https://talks.golang.org/2015/simplicity-is-complicated.slide

Go Concurrency

- Goroutines are **lightweight threads** that **share the same address space**
- Communication happens over channels
- More permissive than Erlangs: Can pass pointers over channels

```
func sum(s []int, c chan int) {
	sum := 0
	for _, v := range s {
		sum += v
	}
	c <- sum // send sum to c
}

func main() {
	s := []int{7, 2, 8, -9, 4, 0}

	c := make(chan int)
	go sum(s[:len(s)/2], c)
	go sum(s[len(s)/2:], c)
	x, y := <-c, <-c // receive from c
	fmt.Println(x, y, x+y)
}
```

Go Interfaces

```go
type geometry interface {
    area() float64
    perim() float64
}
```

```go
type rect struct {
    width, height float64
}

func (r rect) area() float64 {
    return r.width * r.height
}

func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
```

```go
type circle struct {
    radius float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Go Interfaces

- **interface {}**
- Reader implements Read
- Writer implements Write
- Stringer implements String
- Formatter implements Format

There are lots of interfaces in the standard library and in external libraries

http://sweetohm.net/article/go-interfaces.en.html

Discussion: Does Go achieve its goals?

Go success stories

- Docker
- Kubernetes
- etcd
- Google: components of youtube.com and also dl.google.com
- Many companies are using it for distributed applications:
  - Uber
  - Dropbox
  - Netflix

Conclusion: Good “APIs” for a distributed system?

- Distributed Shared Memory
  - Mismatch between physical and logical performance models
  - Fault tolerance very hard
- Message Passing (MPI)
  - Low level, hard to program (deadlocks, fault tolerance)
  - Can be very high performance
- Remote Procedure Calls (GRPC)
  - More structured than Message Passing
  - Can be hard to reason about state and implement fault tolerance
- Task Systems with Immutable Data (Go, Spark, Ray)
  - Easy to program, very high performance, transparent fault tolerance
- Actor Systems (Erlang, Orleans, Ray)
  - Easy to program, support for state