MENDER.io

Deploy Software Updates for Linux Devices

Develop your Embedded Applications Faster: Comparing C and Golang

Marcin Pasinski
Mender.io

My view on C vs Go

- I think Go is great and very productive programming language
- It excels when developing networking code
- I'm not considering it a replacement or competitor for C
- Among the other things garbage collection alone ensures that

Agenda

- What is Go
- Why did we choose go
- Go basics
- Code samples
- Demo

Who am I?

- Marcin Pasinski
    - 10+ years in software development
    - M. Sc., Electronics and Telecommunication
    - marcin.pasinski@northern.tech

Northern.tech

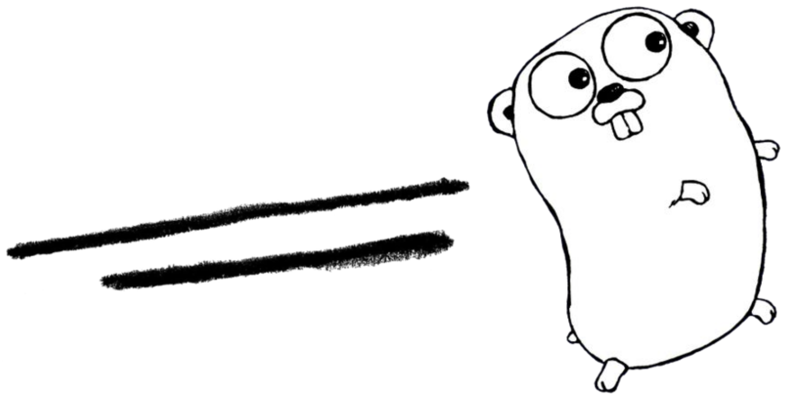
MENDER.io

- OTA updater for Linux devices
- Integrated with Yocto
- Open source (Apache v2 license)
- Written in Go

CFEngine™

- Configuration management tool
- Open source (GPL v3 license)
- Written in C

What is Go: timelines



- Robert Griesemer, Rob Pike and Ken Thompson started sketching — September 21, 2007
- Ian Taylor started GCC front end — May 2008
- Public open source — November 10, 2009
- Go v1 released — March 28, 2012
- Go v1.9 — August 24, 2017

What is Go?

- “Go was born out of frustration with existing languages and environments for **systems programming**.”
- “One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language.”

https://golang.org/doc/faq

Language requirements

1. "External impact"
    - Size requirements on device
    - Setup requirement in Yocto Project
    - Possibility to compile for multiple platforms

2. "Internal considerations"
    - Competences in the company
    - Code share/reuse
    - Development speed
    - Access to common libraries (JSON, SSL, HTTP)
    - "Automatic memory management"
    - "Security enablers" (buffer overflow protection, etc.)

Language comparison

	C	C++	Go
Size requirements in devices	Lowest	Low (1.8MB more)	Low (2.1 MB more, however will increase with more binaries)
Setup requirements in Yocto	None	None	Requires 1 layer (golang)*
Competence in the company	Good	Have some long time users	Only couple of people know it
Buffer under/overflow protection	None	Little	Yes
Code reuse/sharing from CFEngine	Good	Easy (full backwards compatibility)	Can import C API
Automatic memory management	No	Available, but not enforced	Yes
Standard data containers	No	Yes	Yes
JSON	json-c	jsoncpp	Built-in
HTTP library	curl	curl	Built-in
SSL	OpenSSL	OpenSSL	Built-in

* Go is natively supported by Yocto Project from Pyro release (Yocto 2.3)

Yocto build comparison

	C	C++	C++/Qt	Go	...
Pure image size	8.4MB	10.2MB	20.8MB*	14.6MB	
Size with network stack	13.4MB (curl)	15.2MB (curl)	20.8MB*	14.6MB	
Shared dependencies	Yes	Yes	Yes	No/Maybe	
Extra Yocto layer needed	No	No	Yes	Yes**	
Deployment complexity	Binary	Binary	Binary + Qt	Binary	

* Required some changes to upstream Yocto layer
** Go is natively supported by Yocto from Pyro release (Yocto 2.3)

Why did we pick up Go?

1. Golang has lots of core language features and libraries that allows much faster development of applications.
2. The learning curve from C to Golang is very low, given the similarities in the language structure.
3. As it is a compiled language, Golang runs natively on embedded devices.
4. Go is statically linked into a single binary, with no dependencies or libraries required at the device (*note that this is true for applications compiled with CGO_ENABLED=0*).
5. Go provides wide platform coverage for cross-compilation to support different architectures
6. Similar in size with static C binaries, Go binaries continue to get smaller as their compilers get optimized.
7. Both the client and the backend are written in the same language

Go vs C: size

```
package main
func main() {
    println("hello world")
}
```

- $ go build
  - 938K
- $ go build -ldflags '-s -w'
  - **682K**
- $ go build & strip
  - 623K

```
package main
import "fmt"
func main() {
    fmt.Println("hello world")
}
```

- $ go build
  - **1,5M**

```
#include <stdio.h>

int main(void)
{
  printf("hello world\n");
  return 0;
}
```

- gcc main.c
  - 8,5K
- ldd a.out
  - linux-vdso.so.1
  - libc.so.6
  - /lib64/ld-linux-x86-64.so.2
- gcc -static main.c
  - 892K
- gcc -static main.c & strip
  - **821K**

Go vs C: speed

1. Go is fully garbage-collected
2. Go declaration syntax says nothing about stack and heap allocations making those implementation dependant ($ go build -gcflags -m;)
3. Fast compilation
4. Go provides support for concurrent execution and communication
5. The speed of developer is most important in most cases and Go really excels here

The Computer Language Benchmarks Game

https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=go&lang2=gcc

Go basic features

- Standard library
- Tooling
- Compilation
- Concurrency
- Linking with C and C++
- Code samples

Standard library

- Standard library (https://golang.org/pkg/)
  - io/ioutil/os
  - flag
  - net (http, rpc, smtp)
  - encoding (JSON, xml, hex, csv, binary, ...)
  - compress and archive (tar, zip, gzip, bzip2, zlib, lzw, ...)
  - crypto (aes, des, ecdsa, hmac, md5, rsa, sha1, sha256, sha512, tls, x509, ...)
  - database (sql)
  - regexp
  - sync and atomic
  - unsafe and syscall

Tools

- fmt
- test
- cover
- pprof
- doc
- get
- vet
- race detector
- and many more

Compilation

- Compilers
  - The original **gc**, the Go compiler, was written in C
  - As of Go 1.5 the compiler is written in Go with a recursive descent parser and uses a custom loader, based on the Plan 9 loader
  - **gccgo** (frontend for GCC; https://golang.org/doc/install/gccgo)
    - gcc 7 supports Go 1.8.1

- Compilation
  - fast (large modules compiled within seconds)
  - single binary file (no dependencies, no virtual machines)
    - from Go 1.5 possible to create shared libraries and dynamic linking but only on x86 architecture
  - makefile (https://github.com/mendersoftware/mender/blob/master/Makefile)

Cross compilation (https://golang.org/doc/install/source#environment)

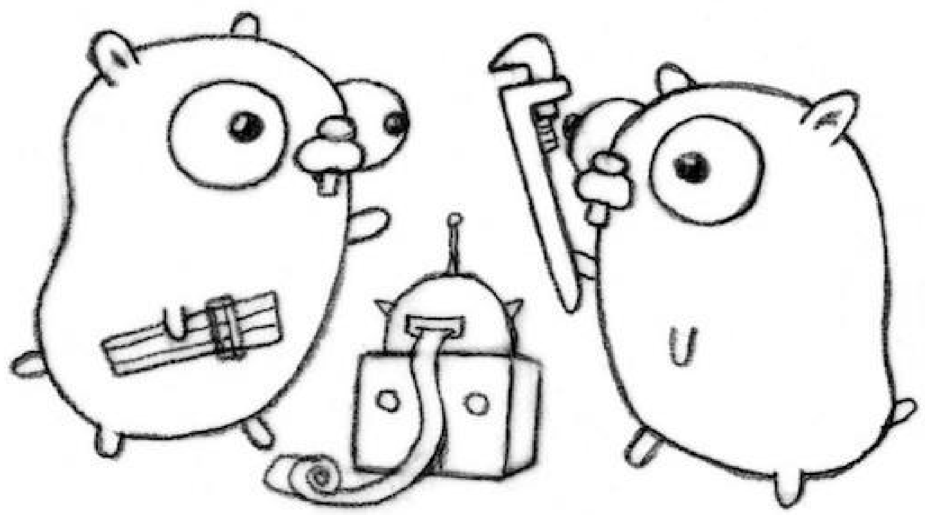
$GOOS / $GOARCH	amd64	386	arm	arm64	ppc64le	ppc64	mips64le	mips64	mipsle	mips
android			X							
darwin	X		X	X						
dragonfly	X									
freebsd	X	X	X							
linux	X	X	X	X	X	X	X	X	X	X
netbsd	X	X	X							
openbsd	X	X	X							
plan9	X	X								
solaris	X									
windows	X	X								

Debugging

- Gdb
- Delve (https://github.com/derekparker/delve)

Testing

- Unit tests
- Benchmarks
- All you need:
  - add "_test" to filename
  - add "Test" to function
  - import "testing"



Variables

- Variable declarations

```
package main
var e, l, c bool
func main() {
    var prague int
    var elc string = "linux"
    var a, s, d = true, false, "data"
    f := 1
}
```

- Basic types
    - bool
    - string
    - int, int8, int16, int32, int64
    - uint, uint8, uint16, uint32, uint64
    - byte //alias for uint8
    - rune //represents a Unicode point; alias for int32
    - float, float64
    - complex64, complex128

Functions

- Functions
    - take zero or more arguments
    - arguments pass by value
    - multiple return values

```
func div(x, y int) (int, error) {
    if y == 0 {
      return 0, errors.New("div by 0")
    }
    return x / y, nil
}

func main() {
    fmt.Println(div(4, 0))
}
```

Structures and methods

- Structs
  - Struct is collection of fields
- Methods
  - Functions with receiver argument
  - Can be declared on non-struct objects

```
type Point struct {
  X int
  Y int
}

type Square struct {
  Vertex Point
  Size int
}

func (s Square) area() int {
  return s.Size * s.Size
}

func (s *Square) setPoint(p Point) {
  s.Vertex = p
}
```

Interfaces

- Interfaces
  - Set of method signatures
  - Implemented implicitly
    - no explicit declaration
    - no “implements”
- Decoupled definition and implementation
- Empty interface *interface{}*

```
type Printer interface {
  Print() (string, error)
}

type myType int
func (mt myType) Print() (string, error) {
  return “this is my int”, nil
}

main() {
  var p Printer = myType(1)
  i.Print()
}
```

Concurrency

- Goroutines
  - Functions that run concurrently with other functions
  - Only few kB initial stack size (2kB)
  - Multiplexed onto OS threads as required
- Channels
  - Used for sending messages and synchronization
  - Sends and receives block by default
  - Can be unbuffered or buffered

Concurrency cont'd

- Goroutines
  - *go func()*

- Channels
  - *c := make(chan int)*

```
package main

func main() {
  messages := make(chan string)
  go func() { messages <- "ping" }()

  select {
     case msg := <- messages:
       fmt.Println(msg)
     case <- time.After(time.Second):
       fmt.Println("timeout")
     default:
       fmt.Println("no activity")
       time.Sleep(50 * time.Millisecond)
  }
}
```

C code inside Go

- CGO (https://golang.org/cmd/cgo/)
  - allows Go to access C library functions and global variables
  - imported C functions are available under virtual C package
  - CGO_ENABLED
  - There is a cost associated with calling C APIs (~150ns on Xeon processor)

```
/*
#cgo LDFLAGS: -lpcap
#include <stdlib.h>
#include <pcap.h>
*/
import "C"

func getDevice() (string, error) {
    var errMsg string
    cerr := C.CString(errMsg)
    defer C.free(unsafe.Pointer(cerr))

    cdev := C.pcap_lookupdev(cerr)
    dev := C.GoString(cdev)
        return dev, nil
}
```

C++ code inside go

- SWIG
  - Simplified Wrapper and Interface Generator
  - Used to create wrapper code to connect C and C++ to other languages
  - http://www.swig.org/Doc2.0/Go.html

```
// helloclass.cpp
std::string HelloClass::hello(){
    return "world";
}

// helloclass.h
class HelloClass
{
public:
      std::string hello();
}

// mylib.swig
%module mylib
%{
#include "helloclass.h"
%}
```

Shared Go libraries

- Possible from Go 1.5
  - *-buildmode* argument
    - archive
    - c-archive
    - c-shared
    - shared
    - exe
- ~ go build -buildmode=shared -o myshared
- ~ go build -linkshared -o app myshared

```go
// package name: mygolib
package main

import "C"
import "fmt"

//export SayHiElc
func SayHiElc(name string) {
  fmt.Printf("Hello ELC: %s!\n", name)
}

func main() {
  // We need the main for Go to
  // compile C shared library
}
```

Shared C libraries

- ~ go build -buildmode=c-shared -o mygolib.a mygolib.go
- ~ gcc -o myapp myapp.c mygolib.a

```c
// mygolib.h
typedef signed char GoInt8;
typedef struct { char *p; GoInt n; }
GoString;


extern void SayHiElc(GoString p0);


// myapp.c
#include "mygolib.h"
#include <stdio.h>

int main() {
  printf("Go from C app.\n");
  GoString name = {"Prague", 6};
  SayHiElc(name);
  return 0;
}
```

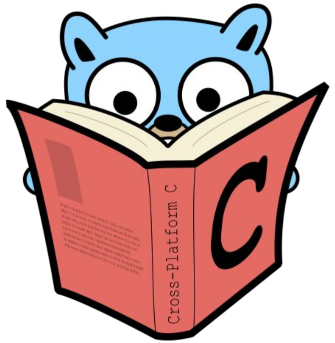
Embedded Go

- Heap vs stack
  - go build -gcflags -m
  - *./main.go:17: msg escapes to heap*
- Unsafe code
  - C: **(uint8_t*)0x1111 = 0xFF;*
  - Manipulating hardware directly is possible with GO, but it has been made intentionally cumbersome.

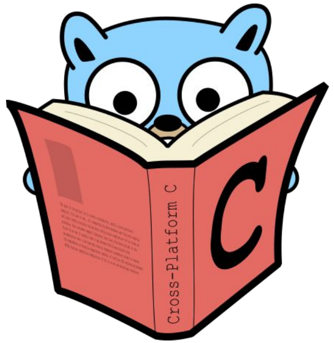
```
file, _ := os.OpenFile("/dev/gpiomem",
      os.O_RDWR|os.O_SYNC, 0);

mem, _ := syscall.Mmap(int(file.Fd()),
    0x20000000, 4096,
    syscall.PROT_READ|syscall.PROT_WRITE,
    syscall.MAP_SHARED)

header :=
*(*reflect.SliceHeader)(unsafe.Pointer(&mem))

memory =
*(*[]uint32)(unsafe.Pointer(&header))
```

Our experience with Go: cons

1. Messy vendoring of 3rd party libraries
2. Quality of community libraries varies a lot
3. Some issues with Yocto Go layer at the beginning
   - all gone after recent efforts of integrating Go with Yocto
4. While using cgo all the easiness of cross-compiling is gone



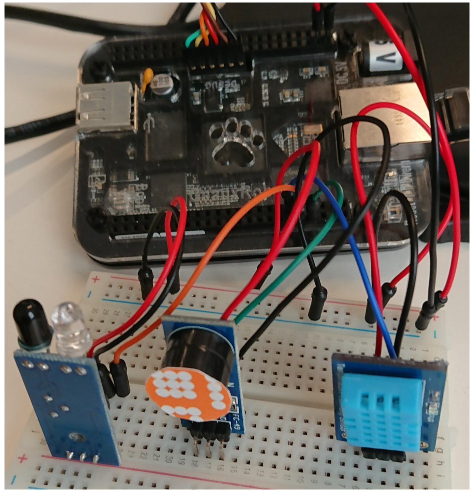
Our experience with Go: pros



1. Easy transition from C/Python (took couple of days to be productive in Go)
2. Very nice tooling and standard library
3. Some tasks exchange between backend and client teams happened, but we've been able to share lot of tools (CI, code coverage)
4. We can share some code between the client and the backend
5. Really productive language (especially when developing some kind of network communication)
6. Forced coding standard so all the code looks the same and is easy to read

Demo

- Yocto
- Mender.io
- ThermoStat ™
    - https://github.com/mendersoftware/thermostat



Q&A