An empirical study of messaging passing concurrency in Go projects

Nicolas Dilley **Julien Lange**

University of Kent

ABCD Meeting — December 2018

Introduction

Go: an open source programming language that makes it easy to build simple, reliable, and efficient software [golang.org].

- Go has become a **key ingredient** of many modern software, e.g., main language of Docker and Kubernetes.
- Go offers **lightweight threads** and **channel-based communication**.
- These communication primitives are similar to synchronisation mechanisms in **process calculi**, e.g., CSP, CCS, and π-calculus.

Complex concurrency patterns: concurrent prime sieve

```
1  func worker(j int, x chan<- int, y <-chan int) {
2    for {
3      select {
4              case x <-j:          // send
5              case   <-y: return // receive
6      }
7    }}
8
```

Complex concurrency patterns: concurrent prime sieve

```
1  func worker(j int, x chan<- int, y <-chan int) {
2    for {
3      select {
4              case x <-j:          // send
5              case   <-y: return // receive
6      }
7    }}
8
9  func main() {
10   a := make(chan int, 5)
11   b := make(chan int)
12
```

Complex concurrency patterns: concurrent prime sieve

```
1  func worker(j int, x chan<- int, y <-chan int) {
2    for {
3      select {
4              case x <-j:         // send
5              case   <-y: return // receive
6      }
7    }}
8
9  func main() {
10   a := make(chan int, 5)
11   b := make(chan int)
12
13   for i := 0; i < 30; i++ {
14       go worker(i, a, b)
15   }
```

Complex concurrency patterns: concurrent prime sieve

```
1  func worker(j int, x chan<- int, y <-chan int) {
2    for {
3      select {
4            case x <-j:          // send
5            case   <-y: return // receive
6      }
7    }}
8
9  func main() {
10   a := make(chan int, 5)
11   b := make(chan int)
12
13   for i := 0; i < 30; i++ {
14       go worker(i, a, b)
15   }
16   for i := 0; i < 10; i++ {
17       k := <-a                       // receive
18       fmt.Println(k)
19   }
```

Complex concurrency patterns: concurrent prime sieve

```go
func worker(j int, x chan<- int, y <-chan int) {
  for {
    select {
           case x <-j:          // send
           case   <-y: return // receive
    }
  }}

func main() {
  a := make(chan int, 5)
  b := make(chan int)

  for i := 0; i < 30; i++ {
      go worker(i, a, b)
  }
  for i := 0; i < 10; i++ {
      k := <-a                        // receive
      fmt.Println(k)
  }
  close(b)
}
```

Context: verification of Go programs

Growing support for verification of Go programs.

Static verification:

- **Dingo-hunter:** multiparty compatibility [Ng & Yoshida; CC'16]
- **Gong:** (bounded) model checking [L, Ng, Toninho, Yoshida; POPL'17]
- **Godel:** mCRL2 model checker [L, Ng, Toninho, Yoshida; ICSE'18]

Context: verification of Go programs

Growing support for verification of Go programs.

Static verification:

- **Dingo-hunter:** multiparty compatibility [Ng & Yoshida; CC'16]
- **Gong:** (bounded) model checking [L, Ng, Toninho, Yoshida; POPL'17]
- **Godel:** mCRL2 model checker [L, Ng, Toninho, Yoshida; ICSE'18]
- **Gopherlyzer:** forkable regular expression [Stadtmüller, Sulzmann, Thieman; APLAS'16]
- **Nano-Go:** abstract interpretation [Midtgaard, Nielson, Nielson; SAS'18]

Context: verification of Go programs

Growing support for verification of Go programs.

Static verification:

- **Dingo-hunter:** multiparty compatibility [Ng & Yoshida; CC'16]
- **Gong:** (bounded) model checking [L, Ng, Toninho, Yoshida; POPL'17]
- **Godel:** mCRL2 model checker [L, Ng, Toninho, Yoshida; ICSE'18]
- **Gopherlyzer:** forkable regular expression [Stadtmüller, Sulzmann, Thieman; APLAS'16]
- **Nano-Go:** abstract interpretation [Midtgaard, Nielson, Nielson; SAS'18]

Runtime verification:

- **Gopherlyzer-GoScout:** [Sulzmann & Stadtmüller; PPDP'17] and [Sulzmann & Stadtmüller; HVC'17]

Challenges for the verification of message passing programs

Scalalibity (wrt. program size)

- Number of message passing primitives (send, receive, etc)
- Number of threads
- Size of channel bounds

Challenges for the verification of message passing programs

Scalalibity (wrt. program size)

- Number of message passing primitives (send, receive, etc)
- Number of threads
- Size of channel bounds

Expressivity (of the communication/synchronisation patterns)

- Spawning new threads within loops
- Creating new channels within loops
- Channel passing

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** *How is concurrency spread across Go projects?*

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** *How is concurrency spread across Go projects?*
  - Can a static analysis focus on specific parts of a codebase?

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** *How is concurrency spread across Go projects?*
  - Can a static analysis focus on specific parts of a codebase?
- **RQ3:** *How common is the usage of* **asynchronous** *message passing in Go projects?*

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** *How is concurrency spread across Go projects?*
  - Can a static analysis focus on specific parts of a codebase?
- **RQ3:** *How common is the usage of* **asynchronous** *message passing in Go projects?*
  - Is asynchrony a problem wrt. scalability?

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** *How is concurrency spread across Go projects?*
  - Can a static analysis focus on specific parts of a codebase?
- **RQ3:** *How common is the usage of* **asynchronous** *message passing in Go projects?*
  - Is asynchrony a problem wrt. scalability?
- **RQ4:** *What concurrent topologies are used in Go projects?*

Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** *How is concurrency spread across Go projects?*
  - Can a static analysis focus on specific parts of a codebase?
- **RQ3:** *How common is the usage of* **asynchronous** *message passing in Go projects?*
  - Is asynchrony a problem wrt. scalability?
- **RQ4:** *What concurrent topologies are used in Go projects?*
  - What sort of constructs should we focus on next?

Methodology

900 projects
Manual filter
865 app. projects
Git clone
Parsing & Metric extraction
35 other projects
HTML files
CSV files

- Selected the **top 900 Go projects** (wrt. number of stars)
- Manually selected 865 projects (35 million PLOC).
- Automatically analysed the AST of each .go in each project.
- Telemetry stored in **machine readable** CSV files and **human browsable** HTML files.

RQ1: *How often are messaging passing operations used in Go projects?*

How common is message passing in 865 projects?

Feature	projects	proportion
chan	661	**76%**
send	617	71%
receive	674	**78%**
select	576	66%
close	402	46%
range	228	26%

- 204 projects out of 865 ($\sim$ 24%) do not create any communication channels.
- the receive primitive is the most frequently used message passing operation.

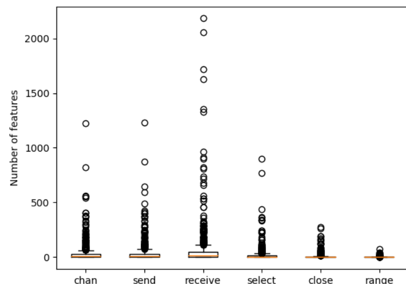
How common is message passing in 865 projects?

Feature	projects	proportion
chan	661	**76%**
send	617	71%
receive	674	**78%**
select	576	66%
close	402	46%
range	228	26%

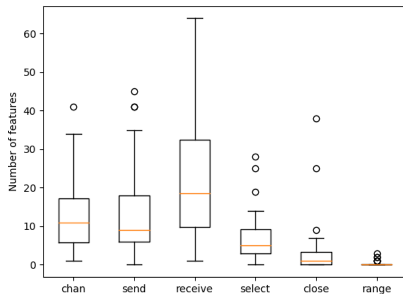
- 204 projects out of 865 ($\sim$ 24%) do not create any communication channels.
- the receive primitive is the most frequently used message passing operation.

NB: receive is also used for delay and timeouts.

Intensity of message passing: absolute measurements



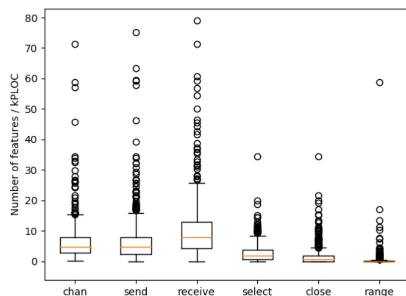
Occurrences in 661 projects



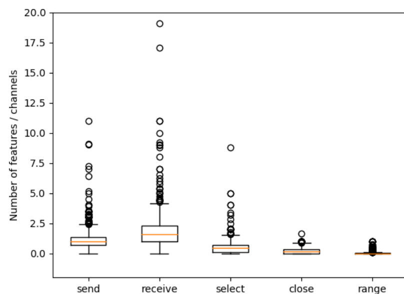
Occurrences in 32 projects

The 32 projects are those whose size falls within 10% of the median size (between 1.7 and 2.1 kPLOC).

Intensity of message passing: relative measurements



Occurrences wrt. size

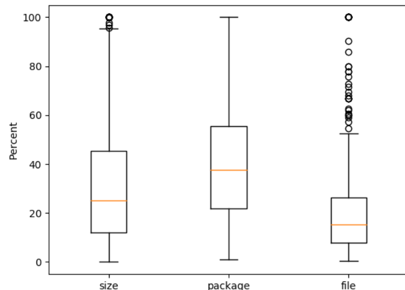


Occurrences wrt. # of channel

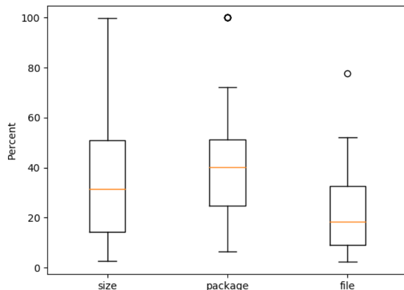
- 6.34 channels for every 1 kPLOC (median of 4.69) in concurrency-related files.
- Some clear outliers, e.g., `anaconda` with one channel creation every 18 PLOC.
- On average: **1.26 sends and 2.08 receives per channel**.

RQ2: *How is concurrency spread across Go projects?*

Concurrency spread



Concurrency spread in 661 projects



Concurrency spread in 32 projects

- **Size:** gives the ratio of **concurrent size** to the total number of physical lines of code.
- **Package:** ratio of number of **packages** featuring concurrency to the total number of packages.
- **File:** gives the ratio of number of **files** containing some concurrency features to the total number of files.

RQ3: *How common is the usage of* **asynchronous** *message passing in Go projects?*

Communication channels in 661 projects

Type	occurrences	proportion
All channels	22226	100%

Communication channels in 661 projects

Type	occurrences	proportion
All channels	22226	100%
Channels with known bounds	20868	94%
Synchronous channels	13639	61%
Asynchronous channels (known)	7229	33%
Channels with unknown bounds	1358	6%

- Asynchrony is much less common than synchrony (default).
- 3237/7229 (45%) asynchronous channels with statically **known bounds** were in **test files**.

Known sizes of asynchronous channels

	mean	std	min	25%	50%	75%	max
size	1193.62	29838.20	1	1	1	5	1,000,000

- Channel bounds are $\leq$ **5** in 75% of the cases.
- **Large bounds** tend to be used to **simulate unbounded asynchrony**.

RQ4: *What concurrent topologies are used in Go projects?*

Complex concurrency patterns: concurrent prime sieve

```go
func generate(ch chan<- int) {
	for i := 2; ; i++ {ch <-i}
}
```

Complex concurrency patterns: concurrent prime sieve

```
1  func generate(ch chan<- int) {
2       for i := 2; ; i++ {ch <-i}
3  }
4
5  func filter(in chan int, out chan int, p int) {
6    for {i := <-in
7      if i%p != 0 {out <-i}
8    }}
```

Complex concurrency patterns: concurrent prime sieve

```
1  func generate(ch chan<- int) {
2        for i := 2; ; i++ {ch <-i}
3  }
4
5  func filter(in chan int, out chan int, p int) {
6    for {i := <-in
7      if i%p != 0 {out <-i}
8    }}
9
10 func main() {
11   ch := make(chan int)
12   go generate(ch)
13   bound := readFromUser()
```

Complex concurrency patterns: concurrent prime sieve

```
1  func generate(ch chan<- int) {
2        for i := 2; ; i++ {ch <-i}
3  }
4
5  func filter(in chan int, out chan int, p int) {
6    for {i := <-in
7      if i%p != 0 {out <-i}
8    }}
9
10 func main() {
11   ch := make(chan int)
12   go generate(ch)
13   bound := readFromUser()
14   for i := 0; i < bound; i++ {
15     prime := <-ch
16     fmt.Println(prime)
17     ch1 := make(chan int)
18     go filter(ch, ch1, prime)
19     ch = ch1
20   }
21 }
```

Frequency of concurrency patterns in 865 projects

Feature	projects	proportion
`go`	711	82%
`go` **in (any)** `for`	**500**	**58%**
`go` in bounded `for`	172	20%
`go` in unknown `for`	474	55%

Frequency of concurrency patterns in 865 projects

Feature	projects	proportion
`go`	711	82%
`go` **in (any)** `for`	**500**	**58%**
`go` in bounded `for`	172	20%
`go` in unknown `for`	474	55%
`chan` in (any) `for`	111	13%
`chan` in bounded `for`	19	2%
`chan` in unknown `for`	103	12%
channel aliasing in `for`	14	2%

Frequency of concurrency patterns in 865 projects

Feature	projects	proportion
`go`	711	82%
`go` **in (any)** `for`	**500**	**58%**
`go` in bounded `for`	172	20%
`go` in unknown `for`	474	55%
`chan` in (any) `for`	111	13%
`chan` in bounded `for`	19	2%
`chan` in unknown `for`	103	12%
channel aliasing in `for`	14	2%
channel in slice	31	4%
channel in map	8	1%
channel of channels	49	6%

Frequency of concurrency patterns in 865 projects

Feature	projects	proportion
`go`	711	82%
`go` **in (any)** `for`	**500**	**58%**
`go` in bounded `for`	172	20%
`go` in unknown `for`	474	55%
`chan` in (any) `for`	111	13%
`chan` in bounded `for`	19	2%
`chan` in unknown `for`	103	12%
channel aliasing in `for`	14	2%
channel in slice	31	4%
channel in map	8	1%
channel of channels	49	6%

NB: 45% of channel as formal parameters had a specified direction.

Known bounds of `for` loops containing go

	mean	std	min	25%	50%	75%	max
bound	280.53	1957.50	1	5	10	100	50000

- 55% of projects use for loops with **unknown bounds**.
- 788/918 (86%) occurrences of a creation of a goroutine within a bounded `for` were located in a **test file**.
- Unfolding loops is probably not a good idea!

Conclusions

- **76%** of the projects use **communication channels**.
- The **number of primitives per channel is low**, suggesting that channels are used for simple synchronisation protocols.

Conclusions

- **76%** of the projects use **communication channels**.
- The **number of primitives per channel is low**, suggesting that channels are used for simple synchronisation protocols.
- On average, just under **half of the packages** of the Go projects we analysed **contain concurrency features**,
- around **20% of files contain concurrency**-related features.

Conclusions

- **76%** of the projects use **communication channels**.
- The **number of primitives per channel is low**, suggesting that channels are used for simple synchronisation protocols.
- On average, just under **half of the packages** of the Go projects we analysed **contain concurrency features**,
- around **20% of files contain concurrency**-related features.
- **Synchronous** channels are the **most commonly used** channels.

Conclusions

- **76%** of the projects use **communication channels**.
- The **number of primitives per channel is low**, suggesting that channels are used for simple synchronisation protocols.
- On average, just under **half of the packages** of the Go projects we analysed **contain concurrency features**,
- around **20% of files contain concurrency**-related features.
- **Synchronous** channels are the **most commonly used** channels.
- **58%** of the projects include **thread creations in `for` loops**.
- **Channel** creation in `for` loops is **uncommon**.

Thanks.

Any questions?