Track One Experiment

Distributed Systems

Johannes Arnold

January 25th, 2023

Leibniz Universität Hannover

Outline

Philosophy

Go Language Features & Jargon

Implementation Details

- The Client (Mapper)
- The Server (Reducer)

Questions?

Philosophy

Taco Bell Programming (2010)

Here's a concrete example: suppose you have millions of web pages that you want to download and save to disk for later processing. How do you do it? The cool-kids answer is to write a distributed crawler in Clojure and run it on EC2, handing out jobs with a message queue like SQS or ZeroMQ. (Taco Bell Programming)

“The Taco Bell answer? `xargs` and `wget`. In the rare case that you saturate the network connection, add some `split` and `rsync`. A ‘distributed crawler’ is really only like 10 lines of shell script.” (*Taco Bell Programming*)

My Interpretation

- Use the utilities already provided

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)
  - Programming Language Standard Libraries

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)
  - Programming Language Standard Libraries
- Scale by external means:

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)
  - Programming Language Standard Libraries
- Scale by external means:

  Cluster Management Software Kubernetes/Docker Swarm/HashiCorp Nomad

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)
  - Programming Language Standard Libraries
- Scale by external means:

  Cluster Management Software Kubernetes/Docker Swarm/HashiCorp Nomad

  Existing Standard Software e.g. NFS for distributing/collecting files

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)
  - Programming Language Standard Libraries
- Scale by external means:

  Cluster Management Software Kubernetes/Docker Swarm/HashiCorp Nomad

  Existing Standard Software e.g. NFS for distributing/collecting files
- Do not over-engineer a big workload that is in reality embarrassingly parallel

My Interpretation

- Use the utilities already provided
  - Shell Globbing (`*.txt`)
  - Programming Language Standard Libraries
- Scale by external means:

  Cluster Management Software Kubernetes/Docker Swarm/HashiCorp Nomad

  Existing Standard Software e.g. NFS for distributing/collecting files
- Do not over-engineer a big workload that is in reality embarrassingly parallel
  - No superfluous services

Go Language Features & Jargon

The Go Programming Language

The Go Programming Language

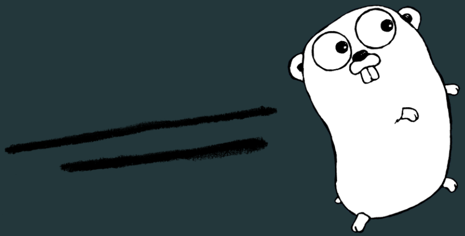

Figure 1: *The Go Gopher*

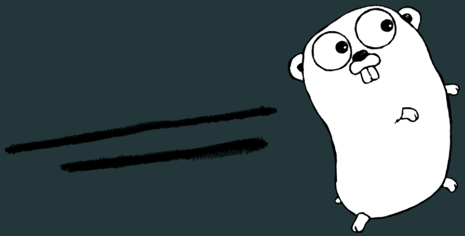

Figure 1: *The Go Gopher*

Why do I like Go?

The Go Programming Language

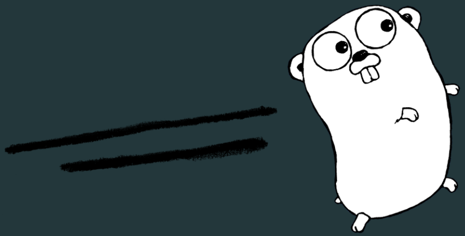

Figure 1: *The Go Gopher*

Why do I like Go?

- Simple syntax

The Go Programming Language

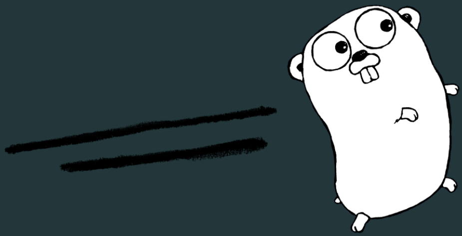

Figure 1: *The Go Gopher*

Why do I like Go?

- Simple syntax
- Fast

The Go Programming Language

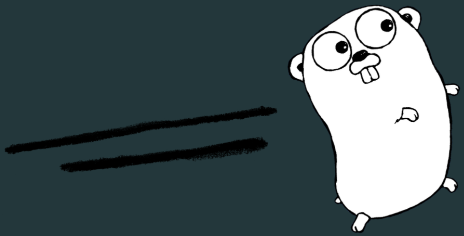

Figure 1: *The Go Gopher*

Why do I like Go?

- Simple syntax
- Fast
- Concurrency built-in

The Go Programming Language

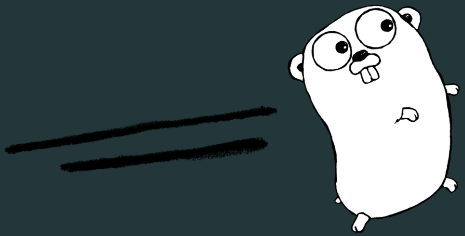

Figure 1: *The Go Gopher*

Why do I like Go?

- Simple syntax
- Fast
- Concurrency built-in
- Easy cross-compilation for a wide variety of operating systems and ISAs

Concurrency

“A *goroutine* is a lightweight thread managed by the Go runtime.” (*A Tour of Go*)

- Excellent for I/O

Concurrency

“A *goroutine* is a lightweight thread managed by the Go runtime.” (*A Tour of Go*)

- Excellent for I/O
- Can run for milliseconds or as long as the main program

Channels

“Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.” (*Go by Example*)

- Basically a form of IPC between goroutines

Channels

"Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine." (*Go by Example*)

- Basically a form of IPC between goroutines
- Channels can be *buffered* with a fixed size

Channels

"Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine." (*Go by Example*)

- Basically a form of IPC between goroutines
- Channels can be *buffered* with a fixed size
- This allows for them to be used as queues

Slices

"An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array. In practice, slices are much more common than arrays. ... The type *[ ]T* is a slice with elements of type *T*." (*A Tour of Go*)

- Resizing, copying, appending is very easy

Slices

"An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array. In practice, slices are much more common than arrays. ... The type *[ ]T* is a slice with elements of type *T*." (*A Tour of Go*)

- Resizing, copying, appending is very easy
- Can be pre-allocated with a specific capacity, yet appear to have a smaller size

Maps

Maps are Go's built-in associative data type (sometimes called hashes or dicts in other languages).

...

Set key/value pairs using typical `name[key] = val` *syntax. (A Tour of Go)*

Methods

"Go does not have classes. However, you can define methods on types." (*A Tour of Go*)

- For both my mapper and reducer, I implemented `struct`s which hold private slices, channels, etc. and are interacted with via methods

Implementation Details

Implementation Details

The Client (Mapper)

Mapping

1. Start a goroutine for each file

Mapping

1. Start a goroutine for each file
   1.1 Read each word in file with `bufio.NewScanner(f)` and configure the scanner to `scanner.Split(bufio.ScanWords)`

Mapping

1. Start a goroutine for each file
   1.1 Read each word in file with *bufio.NewScanner(f)* and configure the scanner to *scanner.Split(bufio.ScanWords)*
   1.2 In a loop: strip everything but Unicode letters with *[^\p{L}]** Regex

Mapping

1. Start a goroutine for each file
   1.1 Read each word in file with `bufio.NewScanner(f)` and configure the scanner to `scanner.Split(bufio.ScanWords)`
   1.2 In a loop: strip everything but Unicode letters with `[^\p{L}]*` Regex
   1.3 Calculate hash modulo the number of reducers

Mapping

1. Start a goroutine for each file
   1.1 Read each word in file with *`bufio.NewScanner(f)`* and configure the scanner to *`scanner.Split(bufio.ScanWords)`*
   1.2 In a loop: strip everything but Unicode letters with *`[^\p{L}]*`* Regex
   1.3 Calculate hash modulo the number of reducers
   1.4 Add to the specific reducer

Mapping

1. Start a goroutine for each file
   1.1 Read each word in file with *bufio.NewScanner(f)* and configure the scanner to *scanner.Split(bufio.ScanWords)*
   1.2 In a loop: strip everything but Unicode letters with *[^\p{L}]** Regex
   1.3 Calculate hash modulo the number of reducers
   1.4 Add to the specific reducer
2. Separate goroutine for each reducer reads from its input channel and adds it to a preallocated slice

Mapping

1. Start a goroutine for each file
    1.1 Read each word in file with *`bufio.NewScanner(f)`* and configure the scanner to *`scanner.Split(bufio.ScanWords)`*
    1.2 In a loop: strip everything but Unicode letters with *`[^\p{L}]*`* Regex
    1.3 Calculate hash modulo the number of reducers
    1.4 Add to the specific reducer
2. Separate goroutine for each reducer reads from its input channel and adds it to a preallocated slice
    - When the slice is full or the channel is closed:
      JSON encode collected words and POST to the Server (Reducer)

Mapping

1. Start a goroutine for each file
   1.1 Read each word in file with *bufio.NewScanner(f)* and configure the scanner to *scanner.Split(bufio.ScanWords)*
   1.2 In a loop: strip everything but Unicode letters with *[^\p{L}]** Regex
   1.3 Calculate hash modulo the number of reducers
   1.4 Add to the specific reducer
2. Separate goroutine for each reducer reads from its input channel and adds it to a preallocated slice
   - When the slice is full or the channel is closed:
     JSON encode collected words and POST to the Server (Reducer)
   - Failure tolerance: linear back-off with re-transmission in case of error

Diagram

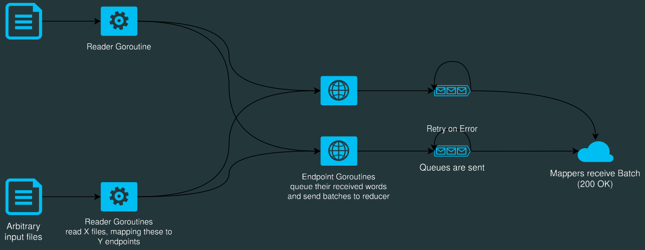

Figure 2: Data Paths in a single Mapping instance

Code Excerpt: Reading Files in Parallel

```
//...
var fileGroup sync.WaitGroup
// Read each file in parallel
for _, f := range r.files {
  fileGroup.Add(1)
  go func(f string) {
    defer fileGroup.Done()
    r.ProcessFile(f)
  }(f)
}
//...
```

Figure 3: `ProcessFile` splits each file into words, which are then mapped to their corresponding reducer queues

Code Excerpt: Hashing a Word

```go
func Hash(word string) uint64 {
  hash := uint64(5381)
  for _, r := range word {
    hash = ((hash << 5) + hash) ^ uint64(r)
  }
  return hash
}
```

Figure 4: XOR-Variant of *djb2 hash function*.

Code Excerpt: *Determinisically* Mapping Words to Reducers

```go
func (r *Reader) Map(word string) {
  // Pick an endpoint for this word
  index := Hash(word) % uint64(len(r.endpoints))
  r.endpoints[index].AddWord(word)
}
```

Figure 5: Mapping a word to its reducer. *AddWord* simply adds the word to the corresponding endpoint's channel.

Code Excerpt: Batching Words

```go
func (e *Endpoint) CollectWords(wg *sync.WaitGroup) {
  wg.Add(1)
  defer wg.Done()
  batch := make([]string, 0, len(e.wordQueue))
  for word := range e.wordQueue {
    batch = append(batch, word)
    // Send all words in the batch once the slice is full
    if len(batch) == cap(batch) {
      e.postWords(batch)
      batch = make([]string, 0, len(e.wordQueue))
    }
  }
  // The channel has been closed, it is time to finish up.
  e.postWords(batch)
}
```

Figure 6: Background goroutine continuously reads from a large buffered channel.

```
johannes@clay:~/Documents/project-ds-j0hax

stems-experiments/track_1/5_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/2_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/4_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/10_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/9_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/3_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/8_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/7_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/6_customer_trends.txt
2023/12/13 22:12:11 Finished reading /var/home/johannes/Documents/distributed-sy
stems-experiments/track_1/1_customer_trends.txt

real    0m5.111s
user    0m9.957s
sys     0m0.321s
johannes@clay:~/Documents/project-ds-j0hax$
```

Figure 7: A mapper after having read all sample files, executed with *`time(1)`*

Pros and Cons

Pros Can work with extremely large files: *`bufio`* only reads part of the file into memory

Pros and Cons

Pros Can work with extremely large files: *`bufio`* only reads part of the file into memory

The collecting channel and slice to encode are fixed size

Pros and Cons

Pros Can work with extremely large files: `bufio` only reads part of the file into memory

The collecting channel and slice to encode are fixed size

Can process a (nearly) arbitrary number of files in parallel

Pros and Cons

Pros	Can work with extremely large files: `bufio` only reads part of the file into memory
	The collecting channel and slice to encode are fixed size
	Can process a (nearly) arbitrary number of files in parallel
Cons	Not as fast as directly dumping all files into memory and sending at once

Pros and Cons

Pros Can work with extremely large files: *`bufio`* only reads part of the file into memory

The collecting channel and slice to encode are fixed size

Can process a (nearly) arbitrary number of files in parallel

Cons Not as fast as directly dumping all files into memory and sending at once

(Circa two seconds slower on my Core i5-4288U CPU @ 2.60GHz with Go 1.21.4)

Implementation Details

The Server (Reducer)

Reducing

1. Start a HTTP Handler

Reducing

1. Start a HTTP Handler
    1.1 Decode arriving JSON into `[]string`

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into `[]string`
   1.2 Load each string into a channel

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into *`[]string`*
   1.2 Load each string into a channel
2. Separate Goroutine counts words from channel

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into `[]string`
   1.2 Load each string into a channel
2. Separate Goroutine counts words from channel
   - Uses a `map[string]int`, reads a word and simply increments the value

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into `[]string`
   1.2 Load each string into a channel
2. Separate Goroutine counts words from channel
   - Uses a `map[string]int`, reads a word and simply increments the value
3. On `SIGINT` (^C):

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into `[]string`
   1.2 Load each string into a channel
2. Separate Goroutine counts words from channel
   - Uses a `map[string]int`, reads a word and simply increments the value
3. On `SIGINT` (^C):
   3.1 Sort all keys to the map

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into *`[]string`*
   1.2 Load each string into a channel
2. Separate Goroutine counts words from channel
   - Uses a *`map[string]int`*, reads a word and simply increments the value
3. On *`SIGINT`* (^C):
   3.1 Sort all keys to the map
   3.2 Loop through sorted list in order

Reducing

1. Start a HTTP Handler
   1.1 Decode arriving JSON into `[]string`
   1.2 Load each string into a channel
2. Separate Goroutine counts words from channel
   - Uses a `map[string]int`, reads a word and simply increments the value
3. On `SIGINT` (^C):
   3.1 Sort all keys to the map
   3.2 Loop through sorted list in order
   3.3 `Fprintf` each key with its corresponding `int` to a tempfile

Implementation with Mappers

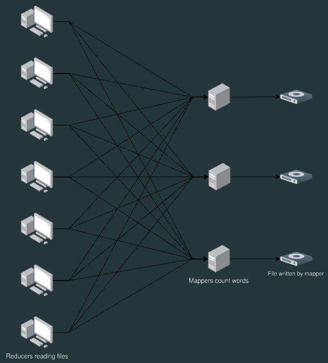

Figure 8: Many mappers typically connect to several reducers in this design

Code Excerpt: Running the HTTP Handler

```
func (w *Writer) Run() {
  var wg sync.WaitGroup
  sigs := make(chan os.Signal, 1)
  signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
  go func() {
    <-sigs
    close(w.incomingWords)
    wg.Wait()
    os.Exit(0)
  }()

  go w.countWords(&wg)
  http.HandleFunc(w.pattern, w.handleWords)
  log.Fatal(http.ListenAndServe(w.bindAddr, nil))
}
```

Figure 9: Handle HTTP requests and handle SIGINT

Code Excerpt: Continuously Counting incoming Words

```
func (w *Writer) handleWords(rw http.ResponseWriter, req *http.Request) {
  defer req.Body.Close()
  decoder := json.NewDecoder(req.Body)
  var wordsReceived []string
  err := decoder.Decode(&wordsReceived)
  if err != nil {
    panic(err)
  }

  for _, word := range wordsReceived {
    w.incomingWords <- word
  }
}
```

Figure 10: Custom HTTP Handler which decodes JSON and dumps it into channel

Code Excerpt: Continuously Counting incoming Words

```
func (w *Writer) countWords(wg *sync.WaitGroup) {
  wg.Add(1)
  defer wg.Done()
  for word := range w.incomingWords {
    w.wordCounts[word] += 1
  }
  w.saveFile()
}
```

Figure 11: Goroutine which keeps track of word counts and saves file as soon as the channel is closed

A note on implementation...

A note on implementation...

A note on implementation...

I *really* wanted use a data structure that sorts data as it arrives asynchronously, but...

[a] Go 1.21's new `slices.Sort()` uses pdqsort

A note on implementation...

I *really* wanted use a data structure that sorts data as it arrives asynchronously, but...

Heap Insert / Pop $\mathcal{O}(\log n)$

---

[a]Go 1.21's new `slices.Sort()` uses pdqsort

A note on implementation...

I *really* wanted use a data structure that sorts data as it arrives asynchronously, but...

Heap Insert / Pop $\mathcal{O}(\log n)$

Hashmap Insert $\mathcal{O}(1)$

[a] Go 1.21's new `slices.Sort()` uses pdqsort

A note on implementation...

I *really* wanted use a data structure that sorts data as it arrives asynchronously, but...

Heap Insert / Pop $\mathcal{O}(\log n)$

Hashmap Insert $\mathcal{O}(1)$

Hashmap Key Sort $\mathcal{O}(nk)$[a]

[a] Go 1.21's new `slices.Sort()` uses pdqsort

Code Excerpt: Sorting and Saving

```
keys := make([]string, 0, len(w.wordCounts))
for k := range w.wordCounts {
  keys = append(keys, k)
}
slices.Sort(keys)
file, err := os.CreateTemp("ds", "excercise")
if err != nil {
  log.Fatal(err)
}
defer file.Close()
for _, k := range keys {
  fmt.Fprintf(file, "%s %d\n", k, w.wordCounts[k])
}
```

Figure 12: Goroutine which keeps track of word counts and saves file as soon as the channel is closed

johannes@clay:~/Documents/project-ds-j0hax

```
johannes@clay:~/Documents/project-ds-j0hax$ ./project-ds-j0hax reduce
2023/12/13 22:11:17 Listening on :1831 under /words
2023/12/13 22:11:17 Press ^C (SIGINT) to save output file when done.
^C2023/12/13 22:12:35 Sorting 54 words
2023/12/13 22:12:35 Saved to /tmp/excercise3414276375
johannes@clay:~/Documents/project-ds-j0hax$
```

Figure 13: A reducer after receiving input and a *SIGINT*

Pros and Cons

Pros Can work with a large amount of parallel requests

Pros and Cons

Pros Can work with a large amount of parallel requests

No data conflicts: separate goroutine listening on one channel, modifying its own `map`

Pros and Cons

Pros Can work with a large amount of parallel requests

No data conflicts: separate goroutine listening on one channel, modifying its own `map`

HTTP is extremely flexible: data can be cached or redistributed/balanced by other programs

Pros and Cons

Pros	Can work with a large amount of parallel requests
	No data conflicts: separate goroutine listening on one channel, modifying its own `map`
	HTTP is extremely flexible: data can be cached or redistributed/balanced by other programs
Cons	Word counts are stored in memory: potential for exhaustion if there are millions of unique words & loss of data on crash

Wrapping it up

Wrapping it up

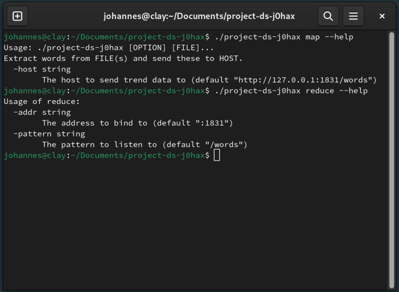

Figure 14: Outputs of *--help* flag for the mapper and reducer subcommand

Wrapping it up

```
johannes@clay:~/Documents/project-ds-j0hax$ ./project-ds-j0hax map --help
Usage: ./project-ds-j0hax [OPTION] [FILE]...
Extract words from FILE(s) and send these to HOST.
  -host string
        The host to send trend data to (default "http://127.0.0.1:1831/words")
johannes@clay:~/Documents/project-ds-j0hax$ ./project-ds-j0hax reduce --help
Usage of reduce:
  -addr string
        The address to bind to (default ":1831")
  -pattern string
        The pattern to listen to (default "/words")
johannes@clay:~/Documents/project-ds-j0hax$
```

Figure 14: Outputs of *--help* flag for the mapper and reducer subcommand

Portability:

Wrapping it up

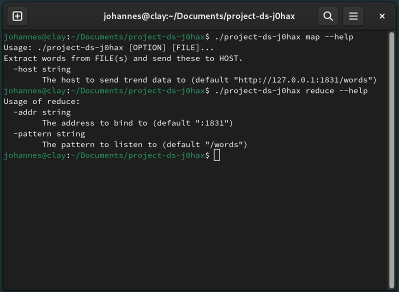

Figure 14: Outputs of *--help* flag for the mapper and reducer subcommand

Portability:

- Client and Server are implemented as standalone *structs*:

Wrapping it up

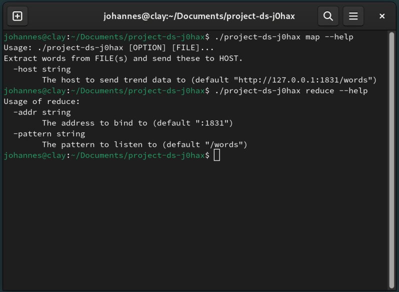

Figure 14: Outputs of `--help` flag for the mapper and reducer subcommand

Portability:

- Client and Server are implemented as standalone *`structs`*:
  - All-in-One executable with subcommands

Wrapping it up

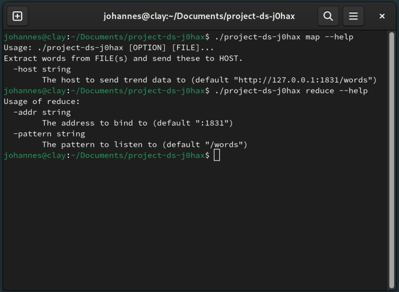

Figure 14: Outputs of `--help` flag for the mapper and reducer subcommand

Portability:

- Client and Server are implemented as standalone `structs`:
  - All-in-One executable with subcommands
  - Parameters can be passed as flags

Wrapping it up

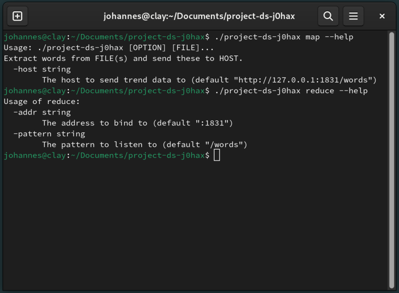

Figure 14: Outputs of `--help` flag for the mapper and reducer subcommand

Portability:

- Client and Server are implemented as standalone `structs`:
  - All-in-One executable with subcommands
  - Parameters can be passed as flags
- A service manager can `SIGINT` a running process at any time, then "convert" a mapper to a reducer or *vice versa*.

Outlook

Outlook

What I have learned:

Outlook

What I have learned:

1. Deeper insight into Go Standard library

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Outlook

Room for improvement:

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling
2. Tune parameters such as buffer size

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling
2. Tune parameters such as buffer size
3. Use environment variables to configure parameters

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling
2. Tune parameters such as buffer size
3. Use environment variables to configure parameters
4. Extend API:

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling
2. Tune parameters such as buffer size
3. Use environment variables to configure parameters
4. Extend API:
   4.1 Healthcheck (beyond checking if process has exited)

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling
2. Tune parameters such as buffer size
3. Use environment variables to configure parameters
4. Extend API:
   - 4.1 Healthcheck (beyond checking if process has exited)
   - 4.2 See statistics and progress for large files

Outlook

What I have learned:

1. Deeper insight into Go Standard library
2. How to work with data races and deadlocks
3. To accept that the most efficient solution may not be the most elegant ;)

Room for improvement:

1. Better error handling
2. Tune parameters such as buffer size
3. Use environment variables to configure parameters
4. Extend API:
    4.1 Healthcheck (beyond checking if process has exited)
    4.2 See statistics and progress for large files
    4.3 Control via API

Questions?

References i

[1] Daniel Julius Bernstein and Ozan Zigit. ***djb2 hash function.*** 2003. URL: *http://www.cse.yorku.ca/%7Eoz/hash.html* (visited on 01/24/2024).

[2] Go Programming Language Contributors. ***A Tour of Go.*** 2011. URL: *https://go.dev/tour* (visited on 12/16/2023).

[3] Ted Dziuba. ***Taco Bell Programming.*** 2011. URL: *http://widgetsandshit.com/teddziuba/2010/10/taco-bell-programming.html* (visited on 12/16/2023).

[4] Renée French. ***The Go Gopher.*** 2012. URL: *https://commons.wikimedia.org/wiki/File:Golang.png* (visited on 12/17/2023).

[5] Mark McGranaghan and Eli Bendersky. ***Go by Example.*** 2012. URL: *https://gobyexample.com/* (visited on 12/16/2023).

Further Acknowledgements

***smiling face with tear* Emoji** part of OpenMoji Project, Artist: Liz Bravo

Further Acknowledgements

***smiling face with tear* Emoji** part of OpenMoji Project, Artist: Liz Bravo

Draw.IO Diagram of Mapper data Flow created with Microsoft Azure icons

Further Acknowledgements

***smiling face with tear* Emoji** part of OpenMoji Project, Artist: Liz Bravo

Draw.IO Diagram of Mapper data Flow created with Microsoft Azure icons

Diagram of Network created with Veeam and Allied Telesis icons