Digo: distributed golang

Hanxiao Lu (hl3424), Yufan Chen(yc3858), Sida Huang(sh4081), Wenqian Yan(wy2249)

February 2, 2021

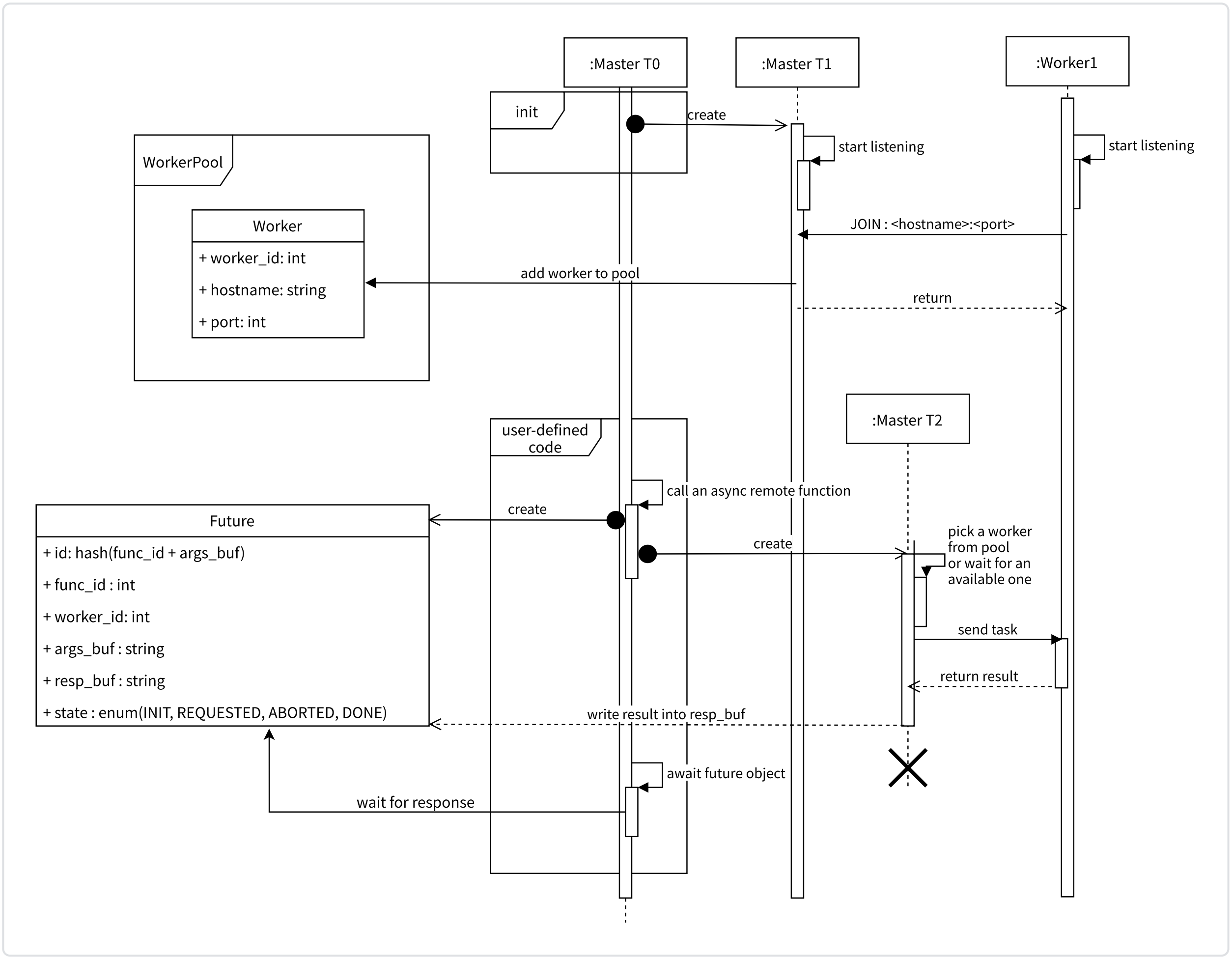
1 Language Overview

Digo is an imperative, statically typed, compiled programming language inspired by Golang, but with support for distributed routine. With Digo, we can easily create a master-worker model program that the master can distribute tasks to worker nodes.

To define a task, we use `async` as the keyword to create a new thread to execute a function. There are two types of async functions: `async func foo(...)` is a local task, and `async remote func foo()` is to define a task that is delivered to workers and execute remotely. We also designed `future` object to define a task.

To fit the master-worker distributed model, the compiled executable has two modes: master mode and worker mode, to specify when executing. An example to run with master mode: `./a.out -h 10.23.2.2 -p 80 -m master`. And to run code in the worker node, we need to specify its master's port and mode in args: `./a.out -h 10.23.2.2 -p 80 -m worker`.

The following plot exhibits the whole underlying process to make the complicated process clear:



As shown in the plot, the master keeps two major threads (T0 and T1): T0 is the main thread to execute the master function, and when it inits, it creates T1 to create and maintain a socket sitting there and waiting for any connections from newly-added workers. When a worker (worker1) is added and starts running, it firstly listens to a port and sends `<hostname, port>` to the master. After the master(T1) receives this information, it records `<worker_id, hostname, port>` in WorkerPool. The worker can close the connection after it recieves ACK from the master.

When executing the master function, everytime a future object is to be created by calling an async remote function, a new thread (T2 in the diagram) is created with it to:

a. run scheduler to find a proper worker in the WorkerPool maintained by the master. If there is no available worker, this thread will stuck in a loop until it gets one.
b. after it finds one worker to deliver the task to, it sends `<function_id + serialized parameters + future_object_id>` to the assigned worker.

Note that, by design, we only pass values instead of pointers to async remote functions and do not allow async tasks to access global variables.

When the worker receives task request from the master, it starts a new thread to look for the function specified by the `function_id` it recieved, and then execute the task. After it finishes the function, it sends a response containing `<result, future_object_id>` back to the master. After the thread (T2) in the master receives the response from the worker, it copies response to `resp_buf` in the corresponding future object. When the master executes `await` or `gather`, the underlying function enters a loop to check if the resp_buf in the future object is filled, otherwise it waits for its response in a blocking way.

An example of potential application is map-reduce model where tasks are computational intensive. Please refer to our word count sample program in the later part.

2 Language Details

2.1 Language Features

Type Scope	Static
Type Strengh	Strongly Typed
Garbage Collection	Automatic
Type System	Statically typed
Evaluation	Strict evaluation
Type Inference	Yes

2.2 Data Types

Data Type	Operation	Examples	Passed by value/reference
string	+, =, ==, >, <, >=, <=, !=	a := "helloworld"	reference
int	=, ==, +, -, *, /, >, <, >=, <=, %, +=, -=, !=, ++	a := 1	value
float	=, ==, +, -, *, /, >, <, >=, <=, %, +=, -=, !=,	a := 1.0	value

	++		
bool	=, ==, !=, !, &&, ||	a := true	value
future	await, =	a := await gcd()	reference

slice	append(), len(), [begin:end]	a := []int{1,2,3}
		append(a, 4)
		b := a[0:2]
			reference

2.3 Built-in Functions

Function Signature	Description
append(<slice>, <element>)	append a new element into a slice
len(<slice>)	get the length of a slice
gather(<future-slice>)	await all future objects in a slice. All future objects in the slice should return the same type of value.

2.4 Keywords

Go

```go
for, if, else, func, return, await, async, remote,
var, string, int, float, bool, continue, break
```

2.5 Control Flows

for loop

Go

```go
for i := 0; i < 100; i++ {
}

for {}

for i < 100 {
    i += 1
    continue
}
```

If-else

Go

```go
if a > b {
} else if a == b{
} else {
}
```

2.6 Functions

The normal function looks like Golang.

Go

```go
func foo() {
}
```

The function named `master` will be treated as the entry function of the master:

Go

```go
func master() {
}
```

Function with the `async` keyword will be treated as an async function to create a new thread to execute. When the master `await <async function>`, the master will check res_buf in future object and wait for the completion of this thread.

Go

```go
async func task() int {
}
```

Function with the `async remote` keyword will be treated as a remotely-executed async function. When the master `await <async remote function>`, internally the program will pick a worker and let this worker execute this function, and wait for it's response.

Go

```go
async remote func task() int {
}
```

2.7 Futures

The return result of an asynchronous function is of `future` type. When a future object is awaited, the program will blockingly wait for the response of the worker that actually executes the function.

The layout of a future object looks like:

C

```c
typedef enum {INIT, REQUESTED, ABORTED, DONE} state_t
typedef unsigned int id_t

struct future {
    id_t id // hash(func_id + args_buf)
    id_t func_id // id of the async function
    // id of the worker the scheduler chose.
    // If this is a local task, then worker_id = 0
    id_t worker_id
    string args_buf // a buffer to store the serialized arguments
    string resp_buf // a buffer to store the serialized return result.
    state_t state
}
```

2.8 Comments

Go

```go
// inline comment
/* comment block */
```

3 Sample Program

Go

```go
//   word-count

// The workers execute this `async remote` function.
// Calling an `async remote` function returns immediately
// after the job is dispatched to a worker. It does not
// wait for the job to finish.
// An `async remote` function does not return with return values
// defined in the function prototype.
// Instead, it returns a future object, on which
// user can call await/gather to get the actual return values.
async remote func worker_count_word(words []string) ([]string, []int) {
    // we count the words by first sorting them.
    Sort(words)
    resultWord := []string{}
    resultCount := []int{}
    if len(words) == 0 {
        return resultWord, resultCount
    }
    word := words[0]
    resultWord = append(resultWord, word)
    resultCount = append(resultCount, 1)
    for i := 1; i < len(words); i++ {
        if words[i] == word {
            resultCount[len(resultWord)-1]++
        } else {
            word = words[i]
            resultWord = append(resultWord, words[i])
            resultCount = append(resultCount, 1)
        }
    }
    return resultWord, resultCount
}
```

```

// The master executes this `async` function in a new thread.
// Calling an `async` function returns immediately with
// a implicit future object.
async func count_word(workerWords [][]string) map[string]int {
    futures := []future{}
    for i := 0; i < len(workerWords); i++ {
        // Calling the `async remote` worker_count_word function
        // will automatically send the task to a worker.
        // It does not block.
        futures = append(futures, worker_count_word(workerWords[i]))
    }
    var words map[string]int
    for i := 0 ; i < len(workerWords); i++ {
        // Here, we are explicity waiting for the remote task
        // worker_count_word(workerWords[i])
        // to finish.
        resultWord, resultCount := await futures[i]
        words[resultWord] += resultCount
    }
    return words
}

func split_file(file string, workerCount int) [][]string {
    reader := FileReader(file)
    workerWords := make([][]string, workerCount)
    rotate := 0
    for {
        // Reads until the first occurrence of delim ' ' or '\n'
        word, err := reader.ReadString([]string{" ", "\n"})
        if err != nil && err != EOF {
            panic("cannot read file")
        }
        // Dispatch words to workers evenly
        worker := rotate % workerCount
        workerWords[worker] = append(workerWords[worker], word)
        rotate++
        if err == EOF {
            break
        }
    }
    return workerWords
}

func word_count_entry() {
    futures := []future{}
    workerWords := split_file("book1.txt")
    futures = append(futures, count_word(workerWords))
    workerWords = split_file("book2.txt")
    futures = append(futures, count_word(workerWords))
    workerWords = split_file("book3.txt")
    futures = append(futures, count_word(workerWords))

    // Calling `async`/`async remote` functions does not block unless
    // we expicitly wait for them to finish by calling await/gather.
    // So word_count_entry() does not block until we `gather` here:
    results := gather(futures)
    // After gather returns, all tasks indicated by futures
    // are finished.
    print("Word Count result of book1.txt")
    print(results[0])
    print("Word Count result of book2.txt")
    print(results[1])
    print("Word Count result of book3.txt")
    print(results[2])
}

func master() {
    word_count_entry()
}
```

After having compiled this code, we will get an executable. In order to run this executable, we need to pass three required arguments:

Bash

```bash
-m [master|worker] specifies whether this is a master or a worker
-h <string> hostname of the master
-p <int> port of the master
```

4 Potential Application

Per the design of Digo, it can divide a complicated and computation-intensive task into separate parts and the master can assign each of them to available workers. Applications of our language include but do not restrict to the following examples in a ditributed way:

- Image processing: The master wants to know the information about a given picture, it commands several workers to convolve with different kernels to do object recognition, object classification and etc.
- Language processing: The master wants to parse different ngrams on a given text, it commands different workers to do different ngram tasks. First worker to parse unigram, Second worker to parse bigram and etc.
- Matrix Multiplication: Matrix multiplication is often expensive. But with our language, master could command each worker to calculate each cell of the output. If task of matrix A*B is given to master, master commands the first worker to calculate A[0,:] * B[:,0] , second worker to calculate A[0,:] * B[:,1] and etc