Go Concurrency and Load Balancing Explained In Color

REVISION 1

Hawthorne-Press.com

GO Concurrency and Load Balancing Explained In Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

CONTENTS

GO CONCURRENCY AND LOAD BALANCING EXPLAINED IN COLOR

The code presented here is from a Google talk on "Go Programming - Google I/O 2010". The example code has been slightly modified to remove features unrelated to the current topic. Additionally, comments have been added to hopefully improve the understanding of the code. This and other documents are available at ***http://www.hawthorne-press.com/***

GENERAL OVERVIEW

The program we will explore is an idealized load balancer system. While it is a simplified version of the concept, it includes all primary unique **Go** features needed by such a program.

What is a Load Balancer? A Load Balancer Function distributes requests for work to a number of worker routines. It attempts to keep all the workers optimally loaded. That is, each doing approximately the same amount of work. This is accomplished by using a *priority queue*.

The general structure of the program, *balance.go,* is shown below:

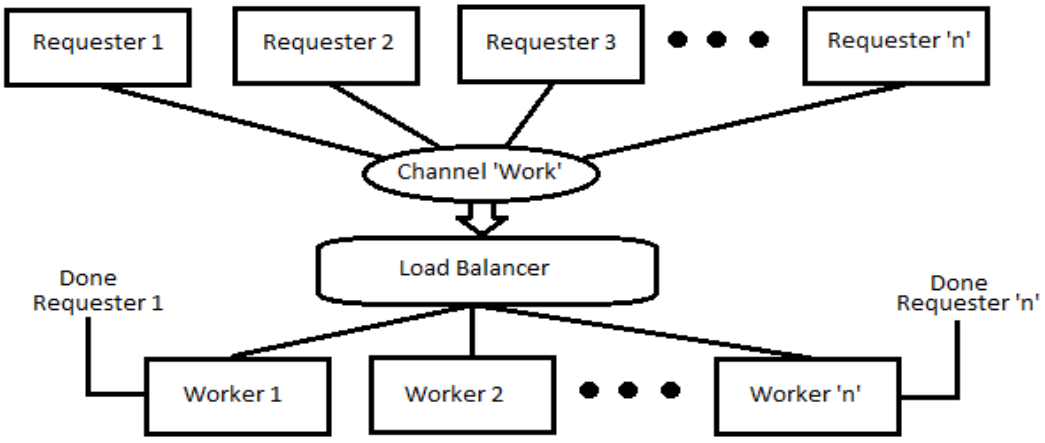

FIGURE 1 - LOAD BALANCER PROGRAM FLOW

Though not shown in the diagram above the priority queue is the heart of the **load balancer** function.

The priority queue data is stored in the pool of *Worker Structures (See: Data Structures)*. To organize and manage the pool we use the Heap Interface, which implements a priority queue. The priority queue is implemented as a balanced binary tree with the property that each sub-tree node has the minimal value for that sub-tree. The result is that the root node always has the minimum pending job count value for the entire tree.

To get a deeper understanding of the heap interface look at the package "container/heap" and also look up the theory of priority queues , which will greatly help in understanding how the heap works. Additionally, we will cover load balancer function and the priority queue code in a later section.

Though not having in part in the actual processing, this program prints balancing statistics each time a worker completes a task. It prints out the number of the pending requests per worker and the average pending requests and their variance. If the balancer is working correctly, the variance should be low, zero of course being best. See examples below:

```
1 1 1 1 1 1 1 1 1 1  1.00 0.00
0 1 1 1 1 1 1 1 1 1  0.90 0.09
1 1 1 1 1 1 2 1 1 2  1.20 0.16
```

In the following sections, we will examine the program **balance.go** in some detail. The entire program is available in the download section of ***http://www.hawthorne-press.com.***

DATA STRUCTURES

The following are all the Constants and Data Structures for the program **balance.go**.

```
const nRequester = 100      // Number of Request Goroutines
const nWorker = 10          // Number of Worker Goroutines

type Request struct {       // Request Structure
    fn func() int           // Work Function to call
    c  chan int             // Reply Channel(Tells Request Function 'work
done')
}

type Worker struct {        // Structure of the Pool Elements
    i        int            // Index into the Pool Structure
    requests chan Request   // Worker Request Value
    pending  int            // Pending Job Count Value
}

type Pool []*Worker       // Create A Slice of Pointers to Worker Structures

type Balancer struct {    // Balance "Object": Contains Pool of Worker
Structures
    pool Pool             // The "Pool":  Managed by the Priority Queue!
    done chan *Worker     // Done Channel for Request Goroutines
}
```

Main Function

The main function consists of four parts.

1. Create "Work" Channel for *Requests.*
2. Creating and Starting *Requester Goroutines.*
3. Creating a Pool of Worker Structures and launching the *Worker Goroutines*.
4. Launching the Balance Loop function.

The code is as follows:

```
func main() {
    work := make(chan Request) // Create the "Work" Channel
    for i := 0; i < nRequester; i++ {
        go requester(work) // Create and start request Goroutines
    }
    b := NewBalancer()     // Create Worker Pool & Start Workers Goroutines

    b.balance(work)        // Launches Balancer Loop
}
```

As shown in ***Figure 1***, there is only one "Work" Channel and all requests are funneled through that channel.

Requester Function Processing

The next step is to create all the Requester Goroutines. The main routine creates **nRequester** number of Requester Goroutines. These routines enter an **infinite** loop that performs the following:

1. Makes a reply channel 'c'.
2. Delays for a random amount of time to simulate request processing.
3. Creates a Request Structure with the appropriated information.
4. Sends the created Request to the "Work" Channel.
5. Waits for a "Done" Signal on the reply channel 'c'.

The Request contains a *Function Pointer* to the 'op' function. This function is the *simulated work function* executed by the Worker Goroutines. It returns a "done" signal on the reply channel.

```
// "Request" Goroutine
//    Infinite Loop - Wait ... Send Request ... Wait for done
func requester(work chan Request) {
    c := make(chan int) // Make Reply Channel
    for {               // Loop Forever
        time.Sleep(time.Duration(rand.Int63n(nWorker * 2e9))) // Random Wait
        work <- Request{op, c}   // Push Request into "Work" Channel
        <-c                      // Wait for "Done" Reply
    }
}

// Simulation of some work: just sleep for a while and report how long.
//
func op() int {         // Actual Simulated Work Function
    n := rand.Int63n(1e9)
    time.Sleep(time.Duration(nWorker * n)) // Sleep random amount
    return int(n)       // Return time slept(value not used)
}
```

NEWBALANCER FUNCTION

The newBalancer function creates *the Worker Structure Pool,* hereafter referred to as *Pool,* and launches the work processing Goroutines **(work(done chan *Worker**)).

This function also creates the "Object" Balancer.

The Pool contains **nWorker** pointers to Worker Structures. The Worker Structure is initialized with a buffered Request Channel **nRequester** in length. The *Pending Job* count is set to zero, and the index 'i' is initialized when it is pushed onto the heap (ie the Pool). The mechanics of this are explained later in the section on the heap and the priority queue.

```
// Create Pool and Start Work Goroutines
func NewBalancer() *Balancer {
    done := make(chan *Worker, nWorker)
    b := &Balancer{make(Pool, 0, nWorker), done}
    for i := 0; i < nWorker; i++ {  // For Each Worker
        // Create a Worker Structure and Point to it
        w := &Worker{requests: make(chan Request, nRequester)}
        heap.Push(&b.pool, w)    // PUSH Struture into Queue
        go w.work(b.done)        // Start work processing routine
    }
    return b        // Return pointer to Balancer Structure
}
```

Objects and Methods

While the Go Language does not have objects in the standard sense, those coming from a background in Object Orientated Programming will find this section informative.

In C++, at a basic level, Objects are essentially structures with methods. The same applies in Go. In that sense, **balance.go** contains three Objects, the *Worker,* Pool, and *Balancer*.

The Worker Structure has one method:

- **work(done chan *Worker)**.

The Pool Structure (ie *Worker) have four methods:

- **Len()**
- **Swap(I, j int)**
- **Push(x interface)**
- **Pop(interface)**

The balancer Structure also has four methods:

- **Balance(work chan Request)**
- **Print()**
- **Dispatch(req Request)**
- **Completed(w *Worker)**

As you will see, Balancer and Pool are closely related. The Balancer Structure holds the Pool Structure. In **Go** parlance, Pool is *embedded* in the Balancer Structure. This feature makes the Pool methods visible to users of the Balance Structure. However, this program accesses Worker Structures though the priority queue using a pointer and this feature of **Go** is not used in this manner.

The Pool

The Pool is a collection of pointers to Worker Structures. This Pool is organized into a *priority queue*.

All access to the Pool is through the *Heap Interface*. The heap functions have no knowledge of the actual structure of the Pool. The user must supply methods for the interface to manipulate the items in the queue. As stated previously, the priority queue is a binary tree, where each sub-tree node has the minimum value for the sub-tree. The following is a visual representation of the relationship between the various structures.

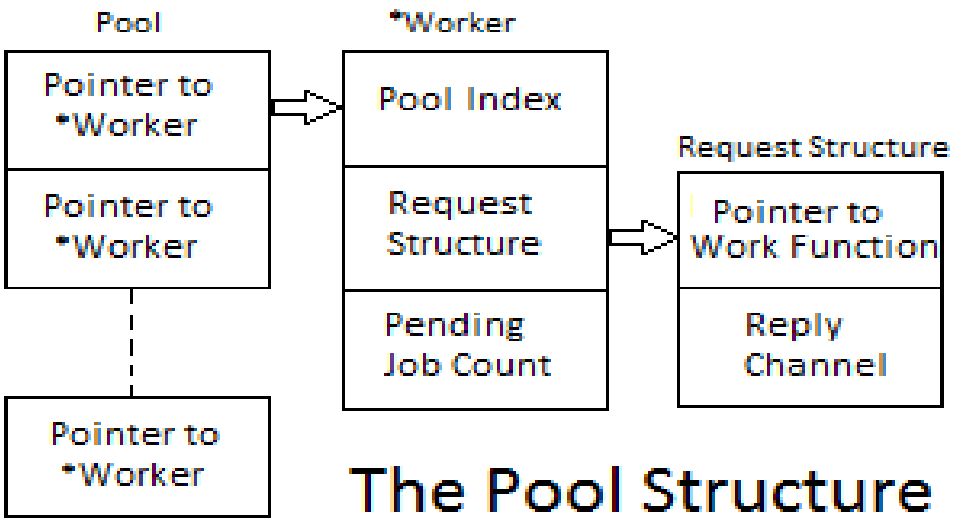

FIGURE 2 - POOL STRUCTURE

Since the Heap Interface does not know the actual structure of pool, this program must supply the *Pool methods* described in the section *Objects and Methods*. This program calls the *Heap methods* **Push**, **Pop**, and **Remove** from the package "container/heap". These methods, in turn, call the user's Pool methods ***Len***, ***Swap***, ***Push***, and ***Pop*** as needed.

```
func (p Pool) Len() int { return len(p) } // Return length of Pool

func (p Pool) Less(i, j int) bool {
    return p[i].pending < p[j].pending // Return Compare of pending values
}

func (p *Pool) Swap(i, j int) {
    a := *p
    a[i], a[j] = a[j], a[i] // Swap Worker Structures
    a[i].i = i              // Adjust Both Pool Indexes
    a[j].i = j
}

func (p *Pool) Push(x interface{}) {
    a := *p           // Get base of Pool Structure
    n := len(a)       // Len of input Slice
    a = a[0 : n+1]    // Create New Element at end+1 of slice
    w := x.(*Worker)  // w now equal *Worker Parameter
    a[n] = w          // Put Worker in new slot
    w.i = n           // Worker.i = position in Pool!
    *p = a            // Update Pool Slice!
}
```

```
func (p *Pool) Pop() interface{} {
    a := *p                 // Get base of Pool Structure
    *p = a[0 : len(a)-1] // Shorten the Pool by 1
    w := a[len(a)-1]      // Load Removed Element
    w.i = -1                // for safety (Non-existant Pool Index)
    return w                // Return Last Pool Value
}
```

A binary tree is represented in the *Pool* slice by storing it in level order traversal format. See correlation in figure 3 below:

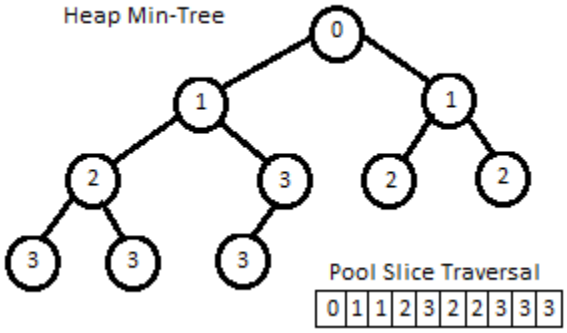

FIGURE 3 - BINARY TREE / TRAVERSAL FORMAT

If you wish to explore this representation, further see: ***https://en.wikipedia.org/wiki/Binary_heap***

The **heap.push** function adds the new worker at the end of the Pool. After the value is added, swaps of the elements are performed until the new element value is less than or equal to all elements in its sub-tree.

The **heap.pop** function removes the element at index zero and returns it. It also uses **Swap** to reorder the Pool.

The functions **Len** and **Swap** are straight forward. The **Less** function is used to determine which elements are *less than or equal* when compared.

BALANCE LOOP FUNCTION

The function ***balance(work)*** is the heart of the Load Balancing Operation. The Balancer Structure (Object) and its methods control the processing of work requests.

The Balance Structure has the Pool Structure embedded within it and a buffered "Done" channel of length **nWorker**. See *Data Structures* section.

The function has a **Select** statement inside an infinite loop. The Select statement waits for one of two conditions:

1. A work request from the "Work" Channel. The Request triggers the **Dispatch** method.
2. A "Done" signal from a Worker Goroutine. It triggers the **Completed** method.

```
// Balance Loop Processing
func (b *Balancer) balance(work chan Request) {
    for { // Infinite Loop
        select {     // Select on Channel
        case req := <-work: // Dispatch Requests
            b.dispatch(req)
        case w := <-b.done: // Process Completions
            b.completed(w)
        }
        b.print()     // Print Statistics
    }
}

func (b *Balancer) dispatch(req Request) {
    w := heap.Pop(&b.pool).(*Worker) // Get Item with least/equal low value
    w.requests <- req                // Update Request Buffer
    w.pending++                      // Advance Pending Count (+1)
    heap.Push(&b.pool, w)            // Update Value in Heap!
}

func (b *Balancer) completed(w *Worker) {
    w.pending--               // Update Pending Value (-1)
    heap.Remove(&b.pool, w.i) // Remove Item at work index in pool
    heap.Push(&b.pool, w)     // Push back Item with a new pending value
}
```

The code above forms the Load Balancer. The balance function retrieves requests from the "Work" Channel and executes the **dispatch** function. This function loads the Root Node Worker and gives it the request. The Pool is then reordered before the next request is processed. This reordering of the pool returns it to a valid priority queue state, insuring that a worker with the lowest pending job count is used on the next iteration.

The **completed** function responds to "done" message from the request reply channel, which indicates that the worker has completed a request. It then decrements the pending job count in the appropriate worker structure and again reorders the priority queue (ie the Pool).

Print Function

The function is a simple diagnostic to show the result of *Load Balancing*. It displays the average load and its statistical variance. It also displays the load on each worker Goroutine.

```
func (b *Balancer) print() { // Print Statistics
    sum := 0
    sumsq := 0
    for _, w := range b.pool { //Loop thru the Pool
        fmt.Printf("%d ", w.pending) // Print Pending Count
        sum += w.pending             // Compute Intermediates
        sumsq += w.pending * w.pending
    }
    // Compute and Print Average and Variance of Pending Counts
    avg := float64(sum) / float64(len(b.pool))
    variance := float64(sumsq)/float64(len(b.pool)) - avg*avg
    fmt.Printf(" %.2f %.2f\n", avg, variance)
}
```

A Final Word

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Revision History

Date	By	Section	Changes
03/13/2016	C.E. Thornton	All	Initial Document