



Published in FAUN Publication



Kevin Wan

Follow

Aug 22, 2021 · 7 min read · [Listen](#)



Save



# Adaptive Load Balancing Algorithm and Implementation

## Background

When choosing a load balancing algorithm, we want to meet the following requirements.

1. scheduling affinity for data centers and different zones
  - select the node with the lowest load
  - select the fastest responsive node as much as possible
1. no need to manually intervene in the failed nodes
  - when a node fails, the load balancing algorithm can automatically isolate the node
  - when the failed node recovers, it can automatically resume traffic distribution to that node

Based on these considerations, `go-zero` chooses the `p2c+EWMA` algorithm to implement it.

## The core idea of the algorithm

## **p2c**

p2c (Pick Of 2 Choices) Two Choices: Randomly select two nodes among multiple nodes.

The ones in go-zero will be randomly selected 3 times, and if the health condition of one of the selected nodes satisfies the requirement, the selection is interrupted and the two nodes are adopted.

## **EWMA**

EWMA (Exponentially Weighted Moving-Average) Exponentially moving-weighted average: The weighting factor of each value decreases exponentially with time, and the closer the value is to the current moment, the larger the weighting factor is, reflecting the average value in the most recent period.

- Equation:

$$V_t = \beta * V_{t-1} + (1 - \beta)\theta_t$$

- Variables explained:
- $V_t$ : represents the EWMA value of the  $t$  request
- $V_{t-1}$ : represents the EWMA value of the  $t-1$  request
- $\beta$ : is a constant

## **Advantages of EWMA algorithm**

1. Compared to the common computed average algorithm, EWMA does not need to save all the past values, which significantly reduces the computation and also the storage resources.
2. Traditional computational averaging algorithms are insensitive to network time

consumption, while EWMA can adjust  $\beta$  by frequent requests to quickly monitor network burrs or to better reflect the overall average.

- When the requests are more frequent, it means that the node network load has increased, and we want to monitor the time taken by the node to process the requests (which reflects the node load), we adjust  $\beta$  down accordingly. The smaller the  $\beta$ , the closer the EWMA value is to the current time consumption, and then we can quickly monitor the network burr;
- When the requests are less frequent, we adjust the  $\beta$  value relatively larger. This way the calculated EWMA value is closer to the average value

### **$\beta$ calculation**

go-zero uses the decay function model of Newton's cooling law to calculate the  $\beta$  value in the EWMA algorithm:

$$\beta = \frac{1}{e^{k * \Delta t}}$$

where  $\Delta t$  is the interval between two requests,  $e$ ,  $k$  are constants

### **Implementing a custom load balancer in gRPC**

1. First we need to implement the `google.golang.org/grpc/balancer/base/base.go` /PickerBuilder interface, which is the Build method that will be called when a service node is updated

```
type PickerBuilder interface {  
    // Build returns a picker that will be used by gRPC to pick a  
    SubConn.  
    Build(info PickerBuildInfo) balancer.Picker  
}
```

1. also implement the `google.golang.org/grpc/balancer/balancer.go/Picker` interface. This interface mainly implements load balancing, picking a node for requests

```
type Picker interface {  
    Pick(info PickInfo) (PickResult, error)  
}
```

1. Finally, register our implemented load balancer with the load balancing `map`

The main logic of go-zero's load balancing implementation

1. In each node update, `gRPC` will call the `Build` method, which is implemented in `Build` to save all the node information.
2. `gRPC` calls the `Pick` method to fetch the node when it gets a node to process the request. `go-zero` implements the `p2c` algorithm in the `Pick` method to pick the node and calculate the load from the node's `EWMA` value and return the node with low load for `gRPC` to use.
3. At the end of the request `gRPC` calls the `PickResult.Done` method, in which `go-zero` implements the method to store the information about the time spent on this request, and calculates the `EWMA` value and saves it for the next request to calculate the load and so on.

## Load Balancing Code Analysis

1. Save all node information of the service
2. We need to store the time spent by the nodes to process this request, `EWMA` and other information, `go-zero` has designed the following structure for each node.

```
type subConn struct {  
    addr resolver.Address  
    conn balancer.SubConn
```

```

    lag uint64 // used to hold ewma values
    inflight int64 // used to store the total number of requests
being processed by the current node
    success uint64 // used to identify the health status of this
connection over time
    requests int64 // Used to store the total number of requests
    last int64 // used to save the last request time, used to
calculate the ewma value
    pick int64 // to save the last picked point in time
}

```

1. `p2cPicker` implements the `balancer.Picker` interface, and `conns` holds information about all nodes of the service

```

type p2cPicker struct {
    conns []*subConn // holds information about all nodes
    r *rand.
    stamp *syncx.AtomicDuration
    lock sync.Mutex
}

```

1. `gRPC` calls the `Build` method when a node is updated, passing in all the node information, where we save each node information in a `subConn` structure. We save each node information here with `subConn` structure and merge them together with `p2cPicker` structure

```

func (b *p2cPickerBuilder) Build(info base.PickerBuildInfo)
balancer.Picker {
    .....
    var conns []*subConn
    for conn, connInfo := range readySCs {
        conns = append(conns, &subConn{
            addr: connInfo.Address,
            conn: conn,
            success: initSuccess,
        })
    }
    return &p2cPicker{
        conns: conns,
        r: rand.New(rand.NewSource(time.Now().UnixNano()))),
    }
}

```

```

    stamp: syncx.NewAtomicDuration(),
  }
}

```

1. randomly picking node information, where there are three cases:
2. there is only one service node, which is returned directly for use by gRPC .
3. there are two service nodes, the load is calculated by EWMA value , and the node with the lower load is returned for gRPC use
4. With multiple service nodes, two nodes are selected by the p2c algorithm, the load is compared, and the node with the lower load is returned for use by gRPC .

The main implementation code is as follows.

```

switch len(p.conns) {
  case 0:// no node, return error
    return emptyPickResult, balancer.ErrNoSubConnAvailable
  case 1:// there is a node, return this node directly
    chosen = p.choose(p.conns[0], nil)
  case 2:// there are two nodes, calculate the load and return the
node with the lower load
    chosen = p.choose(p.conns[0], p.conns[1])
  default:// there are multiple nodes, p2c picks two nodes,
compares the load of these two nodes, and returns the node with the
lower load
    var node1, node2 *subConn
    // select two nodes at random 3 times
    for i := 0; i < pickTimes; i++ {
      a := p.r.Intn(len(p.conns))
      b := p.r.Intn(len(p.conns) - 1)
      if b >= a {
        b++
      }
      node1 = p.conns[a]
      node2 = p.conns[b]
      // If the selected node meets the health requirement this
time, break the selection
      if node1.healthy() && node2.healthy() {
        break
      }
    }
    // Compare the load of the two nodes and choose the one with

```

```

the lower load
    chosen = p.choose(node1, node2)
}

```

1. `load` calculates the load of the node

The `choose` method above will call the `load` method to calculate the node load.

The formula for calculating the load is:  $\text{load} = \text{ewma} * \text{inflight}$

Here is a brief explanation: `ewma` is the average request time, `inflight` is the number of requests being processed by the current node, and multiplying them together roughly calculates the network load of the current node.

```

func (c *subConn) load() int64 {
    // Calculate the load of the node by EWMA; add 1 to avoid the 0
    case
    lag := int64(math.Sqrt(float64(atomic.LoadUint64(&c.lag) + 1)))
    load := lag * (atomic.LoadInt64(&c.inflight) + 1)
    if load == 0 {
        return penalty
    }
    return load
}

```

1. end the request, update the node's `EWMA` and other information
2. subtract 1 from the total number of requests being processed by the node
3. save the time point when the request is finished, use it to calculate the difference between the last request processed by the node, and calculate the  $\beta$  value in `EWMA`
4. calculate the time taken for this request and calculate the `EWMA` value to be saved in the `lag` property of the node
5. calculate the health status of the node and save it to the `success` property of the node

```

func (p *p2cPicker) buildDoneFunc(c *subConn) func(info
balancer.DoneInfo) {
    start := int64(timex.Now())
    return func(info balancer.DoneInfo) {
        // Subtract 1 from the number of requests being processed
        atomic.AddInt64(&c.inflight, -1)
        now := timex.Now()
        // Save the time point at the end of this request and take out
the time point at the last request
        last := atomic.SwapInt64(&c.last, int64(now))
        td := int64(now) - last
        if td < 0 {
            td = 0
        }
        // Calculate the beta in the EWMA algorithm using the decay
function model from Newton's cooling law
        w := math.Exp(float64(-td) / float64(decayTime))
        // Save the elapsed time of this request
        lag := int64(now) - start
        if lag < 0 {
            lag = 0
        }
        olag := atomic.LoadUint64(&c.lag)
        if olag == 0 {
            w = 0
        }
        // Calculate the EWMA value
        atomic.StoreUint64(&c.lag, uint64(float64(olag)*w+float64(lag)*
(1-w)))
        success := initSuccess
        if info.Err != nil && !codes.Acceptable(info.Err) {
            success = 0
        }
        osucc := atomic.LoadUint64(&c.success)
        atomic.StoreUint64(&c.success,
uint64(float64(osucc)*w+float64(success)*(1-w)))

        stamp := p.stamp.Load()
        if now-stamp >= logInterval {
            if p.stamp.CompareAndSwap(stamp, now) {
                p.logStats()
            }
        }
    }
}

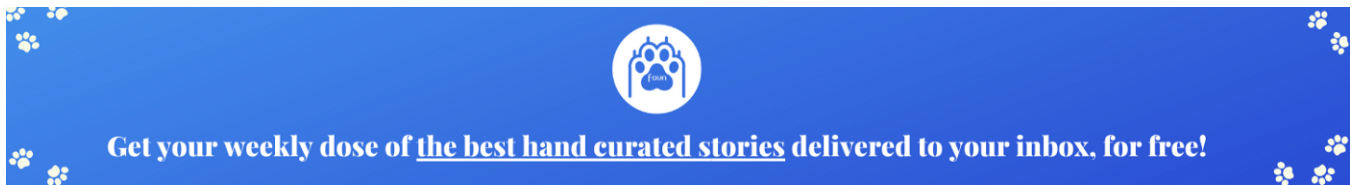
```

**Project address**



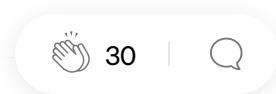
<https://github.com/tal-tech/go-zero>

Welcome to use `go-zero` and give a **star**!



Join FAUN: [Website](#)  [Podcast](#)  [Twitter](#)  [Facebook](#)  [Instagram](#)  [Facebook Group](#)  [Linkedin Group](#)  [Slack](#)  [Cloud Native News](#)  [More](#).

If this post was helpful, please click the clap 🖐️ button below a few times to show your support for the author 🙌



---

## Sign up for FAUN

By FAUN Publication

Medium's largest and most followed independent DevOps publication. Join thousands of developers and DevOps enthusiasts. [Take a look](#).

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.



Get this newsletter