Blade : A Datacenter Garbage Collector

David Terei
CS244b Project

Abstract

Most systems software and distributed systems are still written in non-type-safe languages such as C or C++. The primary reason for this choice by developers is the use of garbage collection in nearly all type-safe languages. Unfourtuently, the latency impact of garbage collection on the slowest requests in a system is generally deemed unacceptable. We look to address this through a technique of garbage collection avoidance which we call Blade. Rather than continue to pursue an endless quest for faster collectors, we instead leverage the replication of data and compute in a distributed system to co-ordinate when collection occurs at an individual node such that the effects on latency for any request is bounded. We describe the API and implementation of Blade in the Go programming language. We also describe two different systems that utilize Blade, a HTTP load balancer, and a consensus algoirthm (Raft), although at this time we only have experimental results for Raft.

1 Blade

Blade attempts to improve the performance of garbage collectors in distributed systems by utilizing the availability of multiples nodes and redundant data sources that are present. Rather than treat collection as a problem isolated to one node, Blade treats it as a temporary failure condition that the system as a whole should model and handle. This allows developers to leverage existing code and expertise present in the system to effectively handle garbage collection. Distributed systems are also a domain where throughput and latency performance is often critical and developers go to lengths to optimize for.

Blade focuses on minimizing the latency effect on requests caused by a GC pause, which generally is the leading contributor to the tail latency [1] in system's written in garbage collected languages. Blade consists of an API between the run-time-system (RTS) and the application itself such that an application developer can customize how GC pauses are handled to their specific application and needs. We discuss some design patterns in Section 2.

The API for Blade consists of the following:

- `void register_gc_notify (handler)` – register a callback to be invoked when the GC has decided that it will need to perform a collection shortly. In this way, we still leave the policy of deciding when a collection should potentially begin up to the RTS.
- `bool notify_callback (gcid, remaining, allocd, gc_estimate)` – the function prototype that `register_gc_notify` callback functions must implement. The `gcid` parameter identifies this collection request from the RTS. The `remaining` parameter says how much memory is left for allocation (in bytes), the `allocd` parameter says how much memory has been allocated since the last collection, and the `gc_estimate` parameter gives an estimate of how long the next collection will take. For now `gc_estimate` is simply the time taken to perform the last collection. The callback returns a boolean, where true means that the RTS should collect immediately, while false means that the RTS should defer collection until the application invokes the `start_gc` function.
- `start_gc(gcid)` – a function that can be used to invoke a collection. Expected to be used by the application after it has been notified by the RTS that a collection will be needed soon. The function will return once the collection has completed. The same `gcid` that was given to the application through

[1] The latency of the slowest requests in the system, such as the 95th percentile or greater. This metric is used as SLA's for systems are described by such bounds.

the `notify_callgack` function should be passed to `start_gc`.

This API is simple enough that most garbage collected languages today already provide these hooks. For those that don't, the work to add it is straightforward. Adding support to Go was an addition of 112 lines of code for example.

In deciding on this API we considered the possibility of specifying the point for when one of the `register_gc_notify` callbacks should be invoked in time as opposed to bytes. We decided against this for two reasons, firstly, such an estimate can be hard to generate accurately for a collector and moves our API away from what is already available today in most RTS, and secondly, by using bytes we gain a more objective measure that can be sent to a controller for the cluster that is deciding when a node should collect and their ordering.

To implement a Blade policy, an application developer will write a `notify_callback` function handler and register that with the RTS. This function only does the initial check of a garbage collection request from the RTS to determine if it's cost is high enough to be handled in application specific manner implemented by the developer, or if it should just be allowed to collect as usual. This is why we use a boolean return argument. It is typical in a RTS design that garbage collection will be co-ordinated by the application (mutator) thread that requested the allocation that caused the collection. As such, the thread invoking the `notify_callback` may be holding locks in the application and as such care must be taken to avoid deadlock. We handle this by simply forking a new thread (in Go these are light-weight, user-level constructus) when the application decides to handle the collection.

As we are potentially delaying collection when the application decides to schedule it, we must decide both when to notify the application of a desired collection through the callback and what to do if the system is out of memory before the application schedules collection. In Go, the GC's policy is to invoke a collection once the amount allocated since the last collection is greater than some ratio of the amount of live data after the last collection. By default this ratio is 1:1. For Go, we simply notify the application of a desired collection once this ratio has been reached, which means we will likely allocate beyond that before the application schedules collection. This is fine as Go doesn't have any way of specifying a memory limit and operates with the assumption it can always grab more. If the RTS collection policy was one that had instead did have a memory limit then we'd want to allow the application to register a low-water mark of remaining memory at which the callback should be invoked.

Finally, the RTS may reach a point where it needs to collect to continue to run and we are still waiting on the application to schedule the requested collection. In this case, the RTS will simply collect and ignore any future call to `start_gc` with a `gcid` that's already been handled. We could also try to notify the application when this occurs so they can cancel any outstanding operations for scheduling the collection but as we expect this to be a rare failure case we avoid the extra complication.

2 Blade Cluster Designs

In this section, we look at applying Blade to a number of different distributed systems. We begin with the simplest case where nodes have no shared state and requests can be routed arbitrarily, we then look at a system with replicated state and finally at a system with state at nodes such that a request can only be routed to one particular node.

2.1 HTTP Proxy: No shared state

Perhaps the most natural application domain for Blade is a fully replicated service where any node can service a request. In particular, we consider a load balanced HTTP service where a single coordinating load balancer proxies client requests to many backend servers. Typically, all backend servers are identical and the load balancer uses simple round-robin to schedule requests. The load balancer can also detect when backend nodes fail by imposing a timeout on requests. However, since some HTTP requests might take a while to service, the load balancer cannot easily distinguish between a misbehaving node servicing a fast request and a properly behaving node servicing a slow request. As a result, timeouts are typically set high — for example, in the NGINX web server, the default timeout is 60 seconds.

The HTTP load balancing application has a few unique properties. First, each request can be routed to any of the replicas. Second, any mutable state is either stored externally (e.g. in a shared SQL database) or is not relevant for servicing client requests (e.g. performance/debugging metrics that are collected periodically). Third, the HTTP load balancer serves as a single, centralized coordinator for all requests [2]. These three properties make Blade easy to utilize for such an application.

The basic idea is that the coordinator treats a backend node garbage collection as a temporary failure. Unlike a network failure, though, the load balancer receives an explicit notification from the node that is about to fail.

[2] Some deployments have multiple HTTP load-balancers, themselves load balanced with DNS or IP load balancing, however, commonly each load balancer in this case manages a separate cluster anyway to make more effective load-balancing decisions.

```
func blade_gc(id) {
  // always replys yes, just need to block
  // on reply, if error, just collect anyway
  // as likely we aren't getting requests
  // from proxy
  _ = send_rpc(controller, can_gc)

  // wait for outstanding requests to finish, load
  // balancer won't send us any new work
  wait_for_trailers()
  start_gc(id)
  send_rpc(controller, done)
}

func handle_gc(id, left, allocd, pause) bool {
  if (below_threshold(left, allocd, pause)) {
    return false
  } else {
    go blade_gc(id)
    return true
  }
}

func init_blade() {
  register_gc_notify(handle_gc)
}
```

Figure 1: Go code for HTTP Cluster using Blade for garbage collection where each request can be routed to any node.

Moreover, the load balancer can decide to postpone the backend node's collection when it is necessary to maintain SLAs — i.e. when too many other node's have failed or are currently collecting.

The pseudo-code for how a backend node uses Blade can be seen in Figure 1. When a backend node begins to experience memory pressure, it immediately informs the load balancer, but continues operating normally. If the all (or a threshold number) of the other nodes are live, the load balancer instructs the backend node to begin garbage collection and marks it dead — i.e. it stops forwarding new requests to it. If too many nodes are off-line right now such that allowing the requesting node to collect would lower our capacity too much, then the load balancer simply queues the request along with any others in order of memory left at the node until some currently collecting nodes have finished. Once a backend node receives permission from the load balancer to collect, it first completes all outstanding requests, safe knowing it won't receive any new requests. Next, it starts the garbage collector, using a stop-the-world collector for maximum throughput. When garbage collection has completed, the backend node informs the load balancer, which marks the backend node live and starts forwarding requests to it.

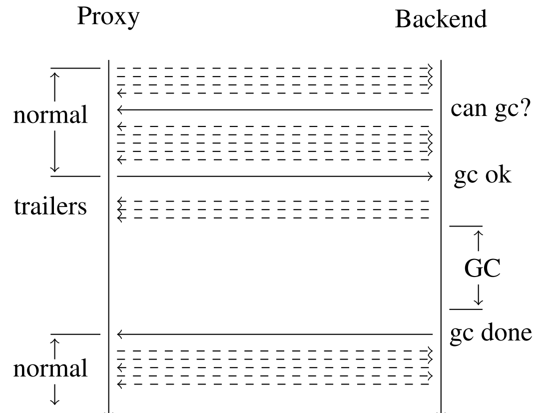

Figure 2: Timeline of Blade messages and backend operations for the HTTP load balancer and backend during a garbage collection. Dashed lines represent normal requests, while solid lines represent Blade messages.

This method allows us to trade capacity for latency, where in normal operating conditions no request should be blocked waiting for the garbage collector. We represent a timeline of the request and Blade messages sent between the load balancer and backend node in Figure 2. We further break this down into how much capacity is available at a backend node in Figure 3. From this figure we can model the total collection time of a backend node, T_B using Blade as $T_B = t_w + t_r + t_{gc} + t_d$ where t_w is the time a node must wait to be scheduled for collection by the load balancer, t_r is the time it takes for the backend to finish up trailing requests, t_{gc} is the time to actually garbage collect, and t_d is the time it takes to send a message to the load balancer that the node has finished collection (i.e., half the RTT). Thus, we can model the capacity downtime as T_B, and assuming enough capacity we can reasonably expect no impact on latency for any request during collection. Even when capacity is reduced enough to impact latency of a request, the impact will be evenly spread over all requests and not isolated and amplified to a few requests causing a far slower tail.

The cost for this though is that T_B is greater than without the use of Blade. In the non-Blade situation, the downtime for a node would simply be t_{gc}, while under blade it greater by $t_w + t_r + t_d$, which has a lower bound of $1.5RTT$, i.e., $T_B > t_{gc} + 1.5RTT$.

2.2 Raft: Strongly consistent replication

In the HTTP load balancer, because there is no shared mutable state on which a client request could depend, any node can service any request and, as a result, we can treat garbage collection events as temporary failures.

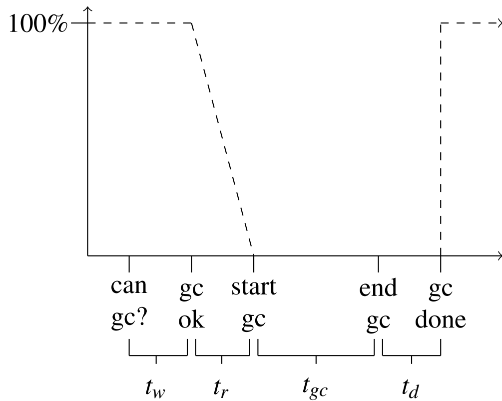

Figure 3: Capacity of a HTTP backend node during a garbage collection cycle with Blade. Total collection time, T_B is $T_B = t_w + t_r + t_{gc} + t_d$, where each t value is represented in the figure.

It turns out that the same is true when mutable state is consistently shared between all nodes, for example, as in a Paxos-like system that uses a consensus algorithm for strongly consistent replication. We consider how to use Blade in a consensus system with strong leadership, such as Raft.

In Raft, during steady-state, all write requests flow through a single leader node. Writes are committed within a single round-trip to a majority of the other nodes, leading to sub-millisecond writes in the common case [3]. Garbage collection pauses can hurt cluster performance in two cases. First, when the leader pauses, all requests must wait to be serviced until GC is complete. If GC pause time exceeds the leader timeout (typically 150ms), the remaining nodes will elect a new leader before GC completes. Second, if a majority of the non-leader nodes are paused for GC, no progress can be made until a majority are live again. The second case is worse, because if garbage collection pauses are very long, there is no built-in way for the system to make progress during this time. The probability of this occurring is also higher than may be expected, due to the replicated state machines executing on each node, the memory consumption is expected to be synchronized across the nodes as well.

We apply Blade to Raft as follows. For the situation where a node that wishes to collect is a follower in the cluster (i.e., not the leader), then we follow the same protocol as the HTTP load balanced cluster, the follower requests with the leader of the Raft cluster permission to start a collection. The leader replies when the follower can collect, which in the normal case will be immedi-

[3] In a low-latency network topology and persistent storage (such as flash drives).

```
// run in own thread
func blade_client() {
  reqInFlight := 0
  gcComplete := 0

  // loop on messages from other threads
  for {
    select {
    // handle a gc request
    case id := <- gc_req:
      reqInFlight = id
      send_rpc(leader, can_gc)

    // gc authorized by leader
    case id := <- gc_auth:
      if id <= gcComplete { continue }
      reqInFlight = 0
      gcComplete = id
      start_gc(id)
      send_rpc(leader, gc_done)

    // leader changed, resend request
    case lead := <- leader_change:
      if reqInFlight != 0 {
        send_rpc(leader, can_gc)
      }
    }
  }
}

func handle_gc(id, left, allocd, pause) bool {
  if (below_threshold(left, allocd, pause)) {
    return false
  } else {
    go func() { gc_req <- id }()
    return true
  }
}

func init_blade() {
  register_gc_notify(handle_gc)
}
```

Figure 4: Pseudo-code for Raft node using Blade for when the node isn't the leader.

ately. The leader may delay replying though if allowing this follower to collect would mean that a quorum could no longer be formed and so consensus not reached, delaying the whole system. An outline of the code for this can be seen in Figure 4, including the retry logic for sending requests to new the new leader if it changes over the course of a collection request. The code makes use of `channels`, a way in Go of passing messages between threads.

When the node that needs to collect is the leader, then we first need to take an aditional step of switching the leadership to a different node before the same code as before for Raft followers can run. This is achieved by having a leader keep a queue on requested collections from node. This queue will include requests from itself (the code from Figure 4 runs regardless of if the node is a leader or follower, it's essentialy a client). The leader keeps track of the number of nodes that can collect simultaneously such that forward progress can still be made by the Raft cluster and only allows a node to collect when by it doing so this won't be violated. When it encounters a request to collect from itself at the head of the queue, then it performs a fast-leadership switch to the node that has most recently garbage collected. If it doesn't have this information, then it chooses another node at random. After this, it resets it's cluster state back to nil and waits for it to become leader again. The blade client thread that sent the request to collect to itself will notice that leadership has changed and then resend the gc request. This time it will go to the newly elected leader though. Pseudo-code for this can be seen in Figure 5.

2.2.1 Raft: Performance

In the case that a node that wishes to collect is a follower, than we can service this without any impact on latency to the system and also no loss of throughput assuming the costs to bring a node up-to-date post collection are within the resource constraints of the system, a reasonable assumption with Raft. When the node that wishes to collect is the cluster leader, then we will take the cost of a fast leader election, delaying requests while that switch takes place. This means that a threshold for invoking Blade rather than just collecting locally without cluster communication should be set to the cost of a fast leader election.

To be precise, the costs are:

Follower

$Latency_F = 0$ (assuming enough capacity)

$GC_F = t_{gc}$

Leader

$Latency_L = t_{leaderchange} + t_{clientretry}$

$GC_L = t_{leaderchange} + GC_F$

```
// run in own thread
func blade_leader() {
  used := 0
  pending := list.New()
  last_collect := None

  // loop on messages from other threads
  for {
    select {
    case m := <- gc_follow_ask:
      if used < slots {
        used++
        if m.From == my_id {
          go switch_leader(last_collect)
        } else {
          send_rpc(m.From, can_gc)
        }
      } else {
        pending.End(m)
      }

    case m := <- gc_finished:
      last_collect = m.From
      if pending.Len() > 0 {
        m = pending.Front()
        if m.From == my_id {
          go switch_leader(last_collect)
        } else {
          send_rpc(m.From, can_gc)
        }
      } else {
        used--
      }

    case lead := <- leader_change:
      used = 0
      pending.Clear()
    }
  }
}
```

Figure 5: Pseudo-code for Raft node using Blade for the when the node is the leader / co-ordinator.

System	SLOC
Go RTS	112
Etcd fast-leader switch	214
Etcd blade support	349

Table 1: Source code changes needed to implement Blade for Go and utilize it with Etcd.

For a follower node, the costs under Blade are no different than if Blade wasn't in use, Blade only plays a role in scheduling when the collection will occur. However, for a leader, the costs without Blade would be:

Follower (without Blade)

$Latency_{F'} = 0 \parallel t_g c$

$GC_{F'} = t_{gc}$

Leader (without Blade)

$Latency_{L'} = min(t_{gc}, timeout_{leader} + t_n + Latency_L)$

$GC_{L'} = t_{gc}$

Where t_n is the cost for the old leader to notice a new leader has been elected when the collection pause is long enough to cause that switch. This gives us a bound for Blade: we should only elect to invoke Blade when the collector reports that the expected collection time is greater than $timeout_{leader} + t_{leaderchange} + t_{clientretry}$.

3 Evaluation

In evaluating Blade we implemented the two systems outlined in Section 2. At this stage though we have only evaluated the performance of the changes to Raft. For Raft, we used Etcd, which is both an implementation of Raft itself and a key-value store with a design very similar to ZooKeeper built on-top of Raft.

3.1 Code Changes

Implementing Blade and using it in Etcd involved modifying both the Go programming language to support Blade and Etcd to support a fast-leadership transfer mechanism, and finally using Blade itself for high-performance garbage collection. Table 1 shows the amount of code changes needed to achieve this.

3.2 Performance

For evaluating the performance we compared a three node Etcd cluster when using Blade, when running with the stock garbage collector, and when running with garbage collection disabled. All experiments were run on machines with dual socket Intel Xeon E5-2630 6 core CPU's, 64GB of RAM over a 10GbE network.

	GC Off	GC On	Blade
95th	1.10	1.10	1.20
99th	1.11	1.40	1.97
99.9th	3.94	83.23	1.63
99.99th	4.70	194.85	4.79
99.999th	13.66	215.26	16.23
99.9999th	13.66	215.26	16.23

Table 2: Various SLA measurements for different Etcd configurations. Timings are in milliseconds (ms).

We ran a single experiment were we evaluated the tail latency of Etcd under the three different configurations. In the experiment we first load 600,000 keys into Etcd and then send 100 requests per second for the next 10 minutes to the cluster, using a mixture of reads and writes in a $3:1$ ratio. We track the latency of each request after the initial load of keys.

We ran the experiment three times for each configuration and took the average of the three. In all configurations the standard deviation was less than 5% of the measured value. The results are presented in Table 2. As can be seen, the numbers for Blade are very close to that of Etcd without garbage collection and over a magnitude better than the stock garbage collector configuration. This is inline with the model for performance outlined in Section 2.2.1 where we are only approximately 2.6ms slower in the worst-case than the GC-Off configuration. The model predicts that latency should only be increased by the cost to switch leaders and to retry a request. On average it takes approximately $1ms$ to switch leaders but at the tail can get up to around $2ms$, while the cost to retry is the RTT of the network, which is $0.180ms$.

4 Conclusion

Blade is a new approach to garbage collection for a particular, but large and important class of programs, those dealing with distributed systems. Blade leverage the extra computing resources in this environment as well as the skill and expertise of distributed systems programmers to allow garbage collection to be scheduled in a system wide fashion that bounds the delay experienced by any single request. We have demonstrated the viabilty of this by investing applying it's design to two different systems and experimentally evaluating one of them, showing performance equal to that of non-garbage-collected language. Blade points to a different design space for collectors, where simple designs can be used that strive for predictability and throughput performance as the latency impact is instead dealt with by Blade.