mallocd: designing a garbage-free nosql data store

igor

2018-04-01

abstract

at bigcorp (tm) we have a lot of data. many jiggabytes worth of data. in order to get the data fast, we need to put it in memory. that is because memory is faster than other forms of storage. in this paper, we introduce a new design for a state-of-the-art, in-memory, garbage-free, cloud-native, nosql, bare metal, containerless, micro-service, data store.

the shift key on my keyboard is broken.

1. garbage

certain “professional” models of a particular brand of computing device have been said, by some, to be aestetically similar to trash cans. this comes as no surprise, indeed, if you look at many “desktop” screens at the office, you will see a similar pattern: the trash can icon is empty, and the entire desktop instead has been littered with garbage.

this is also quite similar to how modern software operates. as the old saying goes: when you leak the trash, you seek and thrash. garbage collectors have been at our disposal for many years now.

before the sunset, the coffee engineers at some microsystems corporation came across a paper about symbolic expressions and their computation by coffee machine. after a bit of small talk, they got to work, and eventually produced a device capable of briefly stopping time. the garbage collector.

this stopping of time occurs when too much garbage is produced. and when time stops in our data store, we cannot get the data very fast.

2. go

all cool new technology is written in go. for this very reason, we chose to use go as the foundation for our new data store.

all cool new technology written in go has a `.io` top-level domain. we tried to get approval from corporate to purchase `mallocd.io`, but the request was denied. as a result, we are currently seeking funding. if you are an investor with $30 bucks to spare, please get in touch.

go is as fast as c, because it was created by the fresh prince of bell labs. except when the garbage collector runs. go employs a “mark-and-weep” collector. this is sometimes also referred to as a “tracing” collector, named after the traces left by the tears rolling down the cheeks of those who are waiting.

n.b. this is also why distributed tracing systems are used in multi-tear architectures.

fun fact: there was a mistake in the original c programming language that ended up costing a billion dollars. but because the mistake was so iconic, they ended up including it in go as well! who knew?

3. allocation

in order to create garbage, things need to be put somewhere. in the context of programming, this process is called memory allocation.

the good old `stdlib.h` defines the following function signatures:

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

`malloc(3)` is used for creating garbage, and `free(3)` is used for collecting it. the reason you've never heard of these functions is because the garbage collector does the work for you.

freeing memory is important, because otherwise the kernel will stop the database process and you will get paged at an inconvenient time. as the old saying goes: when you free your memory, you also free your mind.

4. democratizing malloc

the `malloc(3)` machinery for direct memory allocation allows dealing with memory directly. most contemporary programming languages explicitly deny their users the opportunity to mess with arbitrary memory locations.

what if we had a mechanism that would allow *anyone* to use `malloc(3)`?

this is how we got the idea of `mallocd`. `mallocd` is a stateful micro-service written in go that provides direct memory access for high-performance data storage and retrieval.

the "d" in `mallocd` refers to "daemon", as this service allows its users to harness demonic powers.

the `mallocd` service exposes a udp socket, allowing *any* other process to allocate, access, modify, and free, memory.

5. client

clients for `mallocd` can be written in any language. you could use a command-line client that lets you allocate memory from the comfort of your couch.

here is an example of what that would look like:

```
$ mallocd-client malloc 5
842350568512
$ mallocd-client write 842350568512 5 hello
$ mallocd-client read 842350568512 4
hell
$ mallocd-client free 842350568512
```

don't forget to free your pointers!

6. protocol

`mallocd` uses a binary protocol. mainly for performance reasons, but also so that we could create one of those cool diagrams you see in rfcs.

the protocol exposes 4 methods: `malloc`, `free`, `read`, `write`.

6.1 malloc

the `malloc` request (`0x00`) is used to allocate memory of `len` bytes. it returns a 64-bit pointer to that memory `ptr`.

request:

```
  0 1 2 3 4 5 6 7
 +-+-+-+-+-+-+-+-+
 |      0x00     |
 +-+-+-+-+-+-+-+-+
 |      len      |
 +-+-+-+-+-+-+-+-+
```

reply:

```
  0 1 2 3 4 5 6 7
 +-+-+-+-+-+-+-+-+
 |      ptr      |
 +-+-+-+-+-+-+-+-+
```

these diagrams are not what they seem. what you thought were bits are in fact bytes. we figured, since we're sending huge 64-bit addresses around, we might as well make all fields 64 bits wide. it still looks really cool though.

6.2 free

the `free` request (`0x01`) is used to free the memory allocated at the memory location pointed at by the pointer `ptr`.

request:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|      0x01     |
+-+-+-+-+-+-+-+-+
|      ptr      |
+-+-+-+-+-+-+-+-+
```

this request has no reply. we simply assume the udp message was received and the memory was freed successfully.

this usually works.

6.3 read

the `read` request (`0x02`), not to be confused with the `read(2)` system call, allows reading an arbitrary chunk of memory of length `len` by de-referencing the pointer `ptr`.

request:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|      0x02     |
+-+-+-+-+-+-+-+-+
|      ptr      |
+-+-+-+-+-+-+-+-+
|      len      |
+-+-+-+-+-+-+-+-+
```

reply:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|               |
+  (len bytes)  +
|               |
+-+-+-+-+-+-+-+-+
```

this is mostly safe.

6.4 write

in order to write to any any address in the `mallocd` process, the `write` request (`0x03`) is used. just like `free`, this request does not have a reply.

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|      0x03     |
+-+-+-+-+-+-+-+-+
|      ptr      |
+-+-+-+-+-+-+-+-+
|      len      |
+-+-+-+-+-+-+-+-+
|               |
+  (len bytes)  +
|               |
+-+-+-+-+-+-+-+-+
```

while all of our bits are very significant, we transmit bytes in big-endian order, which is the one that just makes sense.

7. garbage-free computing

with the boring stuff out of the way, let's talk about garbage-free computing.

the go runtime can be provided with a `GODEBUG` environment variable that can print out information about how much garbage a program is producing.

in order to compute without garbage, one must pre-allocate all structs and buffers, and then re-use them. this is precisely what `mallocd` does.

once allocated, no garbage collector will touch those buffers ever again.

unfortunately the go standard library's udp networking code is not entirely garbage-free. receiving udp datagrams results in the allocation of two structs: `syscall.SockaddrInet4` and `net.UDPAddr`.

we can work around this by making syscalls directly:

```
r1, _, e := syscall.Syscall6(
  syscall.SYS_RECVFROM,
  fd,
  uintptr(unsafe.Pointer(&req[0])),
  uintptr(len(req)),
  0,
  uintptr(unsafe.Pointer(&addr)),
  uintptr(unsafe.Pointer(&addrSize))
)
```

now the only garbage produced in `mallocd` is memory allocated by `malloc` requests.

8. manually managing memory

to allocate memory, the `reflect` package can be used as usual:

```
t := reflect.ArrayOf(
  int(len),
  reflect.TypeOf(byte(0))
)
ptr := reflect.New(t).Pointer()
```

`ptr` can now be handed out for anyone to use freely.

however, that allocated memory is now at risk of being collected. in order to prevent that from happening, we keep a reference to it in a shared map:

```
p := unsafe.Pointer(uintptr(ptr))
refs[p] = nil
```

later, the memory can be freed by removing its reference from the map:

```
p := unsafe.Pointer(uintptr(ptr))
delete(refs, p)
```

this is a subtle way of instructing the garbage collector to run `free(3)` on that pointer and reclaim the memory.

don't forget to free your pointers!

9. results

we will be rolling out `mallocd` to production at bigcorp (tm) next week.

10. conclusion

`mallocd` is a next-gen, best-in-class, garbage-free, in-memory, nosql datastore written in go.

it democratizes `malloc` by allowing *anyone* to mess with memory in *any* language.

since it's just memory, it's easy to implement your own data structures on top of `mallocd`.

don't forget to free your pointers!

11. future work

other data stores provide support for scripting via lua stored procedures. `mallocd` provides the ability to write to arbitrary memory, it may be possible to write into the stack segment in order to create and run user-defined functions.

in order to simplify the adoption of `mallocd`, we want to develop a posix-compliant drop-in replacement for `malloc(2)` that can be loaded via `LD_PRELOAD`. it could use a `SIGSEGV` signal handler to intercept memory accesses, or if that doesn't work, we might just make a kernel module.

references

here are a few pointers:

- 0xc42001a440
- 0xc4200cb9e0
- 0xc4201355c0

don't forget to free them when you're done.