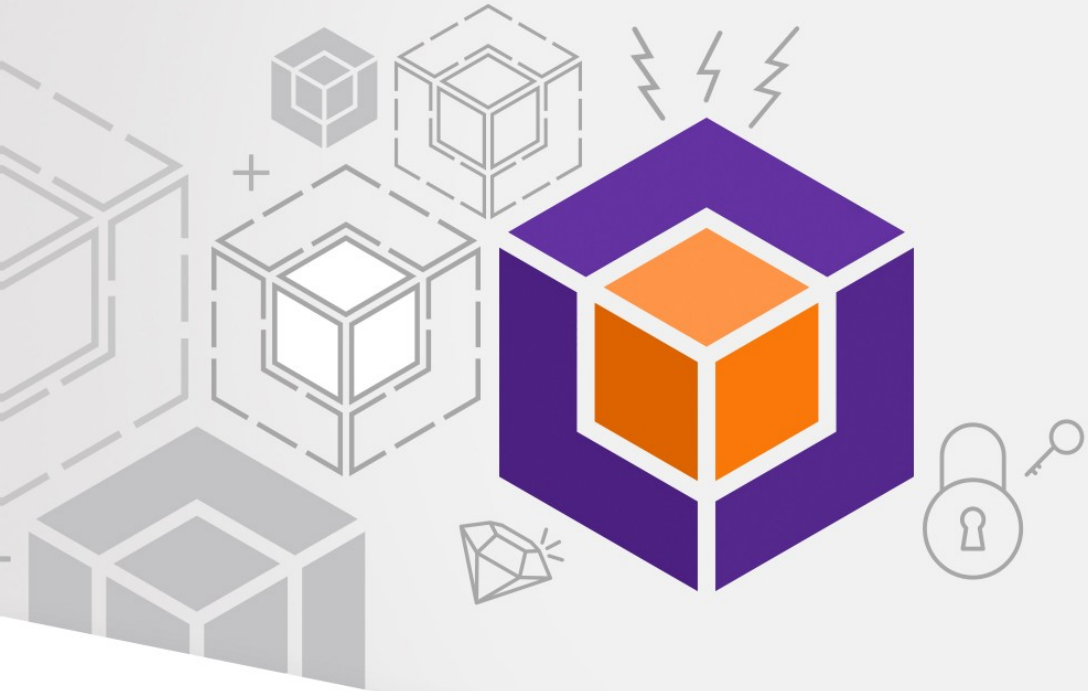
YottaDB

October 15, 2019



Hidden Dragons of CGO: Hard-learned Lessons from Writing Go Wrappers

Outline

YottaDB

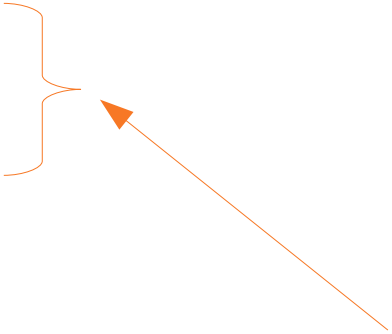
- Motivation & Background – YottaDB, C API, Go, CGO
- Parameter Passing
- Callbacks
- Garbage Collection (common thread)
- Questions & Answers

Key-Value Tuples

```
["Capital","Belgium","Brussels"]
["Capital","Thailand","Bangkok"]
["Capital","USA","Washington,DC"]
```

Key

Value



Always sorted – YottaDB means never having to say you're sorting!

Schemaless

```
["Capital","Belgium","Brussels"]
["Capital","Thailand","Bangkok"]
["Capital","USA","Washington,DC"]
["Population","Belgium",13670000]
["Population","Thailand",84140000]
["Population","USA",325737000]
```

Schema determined entirely by application – YottaDB assigns no meaning

Numbers and strings (blobs) can be freely intermixed in values and keys except first

Default order for each key:

- Empty string ("")
- Canonical numbers in numeric order
- Strings (blobs) in lexical order

Mix Key Sizes

YottaDB

```
["Capital","Belgium","Brussels"]
["Capital","Thailand","Bangkok"]
["Capital","USA","Washington,DC"]
["Population","Belgium",13670000]
["Population","Thailand",84140000]
["Population","USA",325737000]
["Population","USA",17900802,3929326]
["Population","USA",18000804,5308483]
…
["Population","USA",20100401,308745538]
```

"Population" + 1 more key means value is latest population

"Population" + 2 more keys means value is population on date represented by last key

yyyymmdd

Keys ⟷ Array References

YottaDB

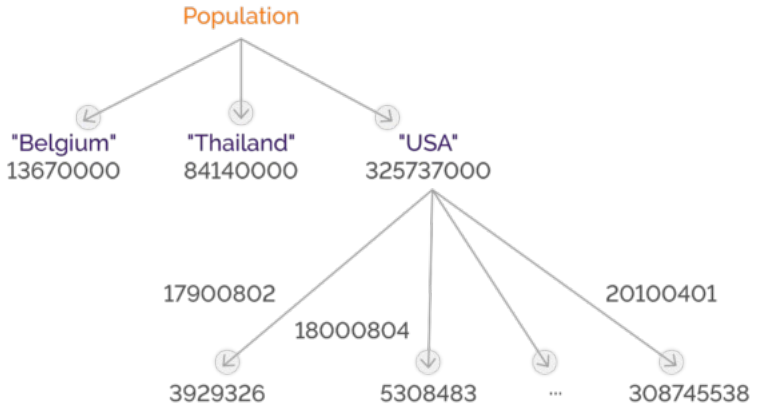
```
^Population("Belgium")=13670000
^Population("Thailand")=84140000
^Population("USA")=325737000
^Population("USA",17900802)=3929326
^Population("USA",18000804)=5308483
 …
^Population("USA",20100401)=308745538
```

First key is variable name

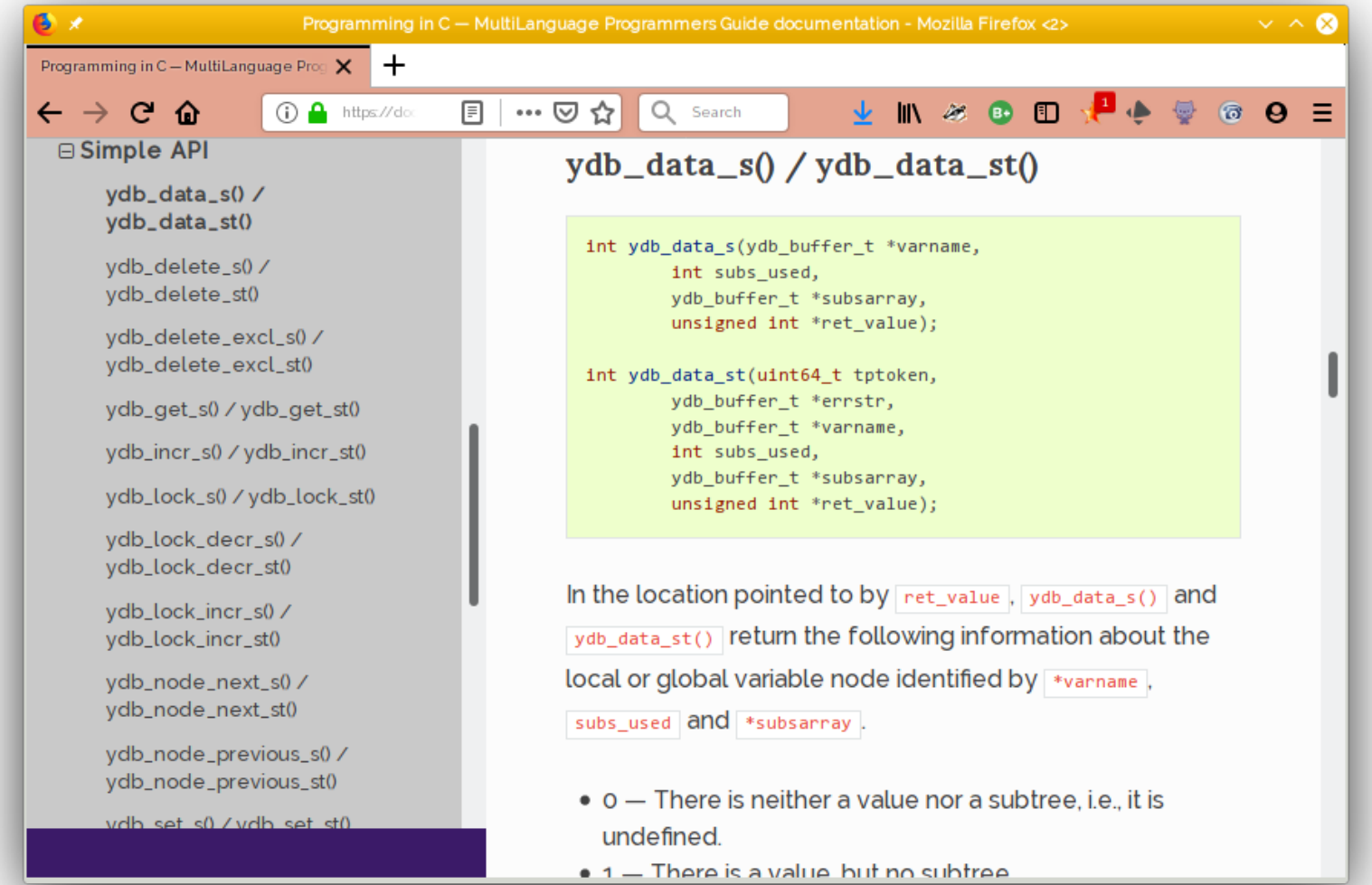
Other keys are subscripts

Array references are a familiar programming paradigm

POPULATION TREE



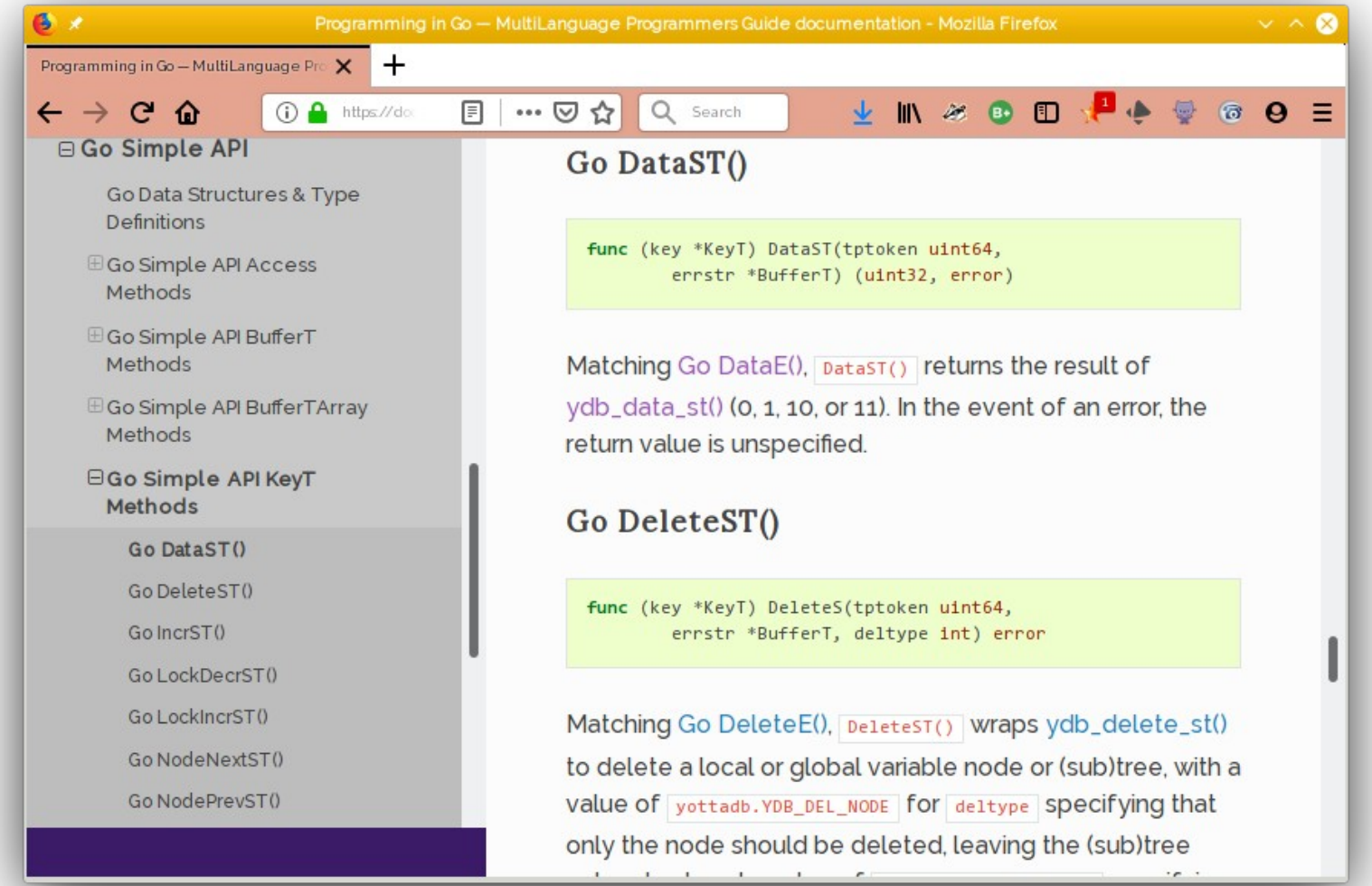
Any JSON structure is representable as a tree, but not vice versa


Programming in C — MultiLanguage Programmers Guide documentation - Mozilla Firefox <2>
Programming in C — MultiLanguage Prog
Search
Simple API
ydb_data_s() / ydb_data_st()
ydb_delete_s() / ydb_delete_st()
ydb_delete_excl_s() / ydb_delete_excl_st()
ydb_get_s() / ydb_get_st()
ydb_incr_s() / ydb_incr_st()
ydb_lock_s() / ydb_lock_st()
ydb_lock_decr_s() / ydb_lock_decr_st()
ydb_lock_incr_s() / ydb_lock_incr_st()
ydb_node_next_s() / ydb_node_next_st()
ydb_node_previous_s() / ydb_node_previous_st()
ydb_set_s() / ydb_set_st()
ydb_data_s() / ydb_data_st()
int ydb_data_s(ydb_buffer_t *varname,
int subs_used,
ydb_buffer_t *subsarray,
unsigned int *ret_value);
int ydb_data_st(uint64_t tptoken,
ydb_buffer_t *errstr,
ydb_buffer_t *varname,
int subs_used,
ydb_buffer_t *subsarray,
unsigned int *ret_value);
In the location pointed to by ret_value, ydb_data_s() and ydb_data_st() return the following information about the local or global variable node identified by *varname, subs_used and *subsarray.
0 — There is neither a value nor a subtree, i.e., it is undefined.
1 — There is a value, but no subtree

C API

How about a Go API to YottaDB?

- Everything we do is 100% free / open source
- We make money from support contracts and funded enhancements
- What could be simpler than a Go wrapper to the C API as a funded enhancement?


Programming in Go — MultiLanguage Programmers Guide documentation - Mozilla Firefox
Programming in Go — MultiLanguage Pro
Search
Go Simple API
Go Data Structures & Type Definitions
Go Simple API Access Methods
Go Simple API BufferT Methods
Go Simple API BufferTArray Methods
Go Simple API KeyT Methods
Go DataST()
Go DeleteST()
Go IncrST()
Go LockDecrST()
Go LockIncrST()
Go NodeNextST()
Go NodePrevST()
Go DataST()
func (key *KeyT) DataST(tptoken uint64,
errstr *BufferT) (uint32, error)
Matching Go DataE(), DataST() returns the result of ydb_data_st() (0, 1, 10, or 11). In the event of an error, the return value is unspecified.
Go DeleteST()
func (key *KeyT) DeleteS(tptoken uint64,
errstr *BufferT, deltype int) error
Matching Go DeleteE(), DeleteST() wraps ydb_delete_st() to delete a local or global variable node or (sub)tree, with a value of yottadb.YDB_DEL_NODE for deltype specifying that only the node should be deleted, leaving the (sub)tree

Go API

Finoskov [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)]

Yes, indeed, what could be simpler?

CGO

YottaDB

- https://golang.org/cmd/cgo/
  - A lot of code is written in C (GUI libraries, databases, math libraries, …); calling into that code opens a whole new world of possibilities
  - C is not garbage collected; every C program is responsible for its own memory allocations (malloc) and frees; CGO attempts to work around this by creating rules

Passing Go Pointers to C

YottaDB

"Go code may pass a Go pointer to C provided the Go memory to which it points does not contain any Go pointers. The C code must preserve this property: it must not store any Go pointers in Go memory, even temporarily. When passing a pointer to a field in a struct, the Go memory in question is the memory occupied by the field, not the entire struct. When passing a pointer to an element in an array or slice, the Go memory in question is the entire array or the entire backing array of the slice. "

- https://golang.org/cmd/cgo/#hdr-Passing_pointers

Integers are OK

YottaDB

```
package main
import "fmt"
// int myFunc(int r1) {
//  return r1 + 42;
// }
import "C"
func main() {
  fmt.Printf("%v", C.myFunc(22))
}
```

C char is an Integer

YottaDB

```
package main
import "fmt"
// int myFunc(char c1) {
//  return c1 + 42;
// }
import "C"
func main() {
  fmt.Printf("%v", C.myFunc(22))
}
```

C Strings are not Go Strings

YottaDB

- Compilation error – a Go string is a descriptor unlike a C string

```
package main
// #include <stdio.h>
// void myFunc(char *c1) {
//  printf("Hello %s!\n", c1);
// }
import "C"
func main() {
  C.myFunc("world!")
}
```

Passing a Reference to 1st Element …1

YottaDB

- Functionally correct
- Protected from garbage collection

```
package main
import "unsafe"
// #include <stdio.h>
// void myFunc(void *c1) {
//  printf("Hello %s!\n", (char*)c1);
// }
import "C"
func main() {
  msg := ([]byte)("world!")
  C.myFunc(unsafe.Pointer(&msg[0]))
}
```

Passing a Reference to 1st Element …2

YottaDB

- Also functionally correct
- Also protected from garbage collection

```
package main
import "unsafe"
// #include <stdio.h>
// void myFunc(char *c1) {
//  printf("Hello %s!\n", c1);
// }
import "C"
type BufferStruct struct {
    buff []byte
}
func main() {
  myBuff := BufferStruct{([]byte)("world!")}
  C.myFunc((*C.char)(unsafe.Pointer(&myBuff.buff[0])))

}
```

Cannot Pass a Structure with a Pointer …1

YottaDB

- Compilation error

```
package main
// #include <stdio.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//  printf("Hello %s!\n", strct->msg);
// }
import "C"
func main() {
  msg := C.myStruct{"world!"}
  C.myFunc(&msg)
}
```

Cannot Pass a Structure with a Pointer …2

YottaDB

- Compiles
- Runtime error

```
package main
import "unsafe"
// #include <stdio.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//  printf("Hello %s!\n", strct->msg);
// }
import "C"
func main() {
  val := ([]byte)("world!")
  msg := C.myStruct{(*C.char)(unsafe.Pointer(&val[0]))}
  C.myFunc(&msg)
}
```

Go Structures with Pointers to C structs …1

YottaDB

- Compiles
- Runs
- But leaks memory

```
package main
// #include <stdio.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//  printf("Hello %s!\n", strct->msg);
// }
import "C"
func main() {
  msg := C.myStruct{C.CString("world")}
  C.myFunc(&msg)
}
```

Go Structures with Pointers to C structs …2

YottaDB

- Compiles
- Runs
- Can use explicit free
  - Unfriendly
  - Error prone

```
package main
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//  printf("Hello %s!\n", strct->msg);
// }
import "C"
func main() {
  msg := C.myStruct{C.CString("world")}
  defer C.free(msg.msg)
  C.myFunc(&msg)
}
```

Go Structures with Pointers to C structs …3

YottaDB

- Finalizers
  - Annoys Gophers
- Can still cause segmentation violations!

```
package main
import "unsafe"
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//  printf("Hello %s!\n", strct->msg);
// }
import "C"
func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(&msg)
}
```

Go Structures with Pointers to C structs

YottaDB

- Finalizers
- KeepAlive to protect from garbage collection

```
package main
import "unsafe"
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//  printf("Hello %s!\n", strct->msg);
// }
import "C"
func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(&msg)
  runtime.KeepAlive(&msg)
}
```

Callbacks

- Common in C, especially with UI work
- YottaDB uses callbacks for ACID transactions

Callbacks – “Obvious approach”

YottaDB

- Does not compile

```
package main
// void do_callback(void (*cb)()) {
//  cb();
// }
import "C"
func callback() {
  fmt.Printf("Here!\n")
}
func main() {
  C.do_callback(callback)
}
```

Callbacks – “Obvious” Solution

- Split into two files
- C func to return callback

```
package main
// void do_callback(void (*cb)())
{
//  cb();
// }
// extern void callback();
// void (*get_callback())() {
//  return callback;
// }
import "C"
func main() {
  C.do_callback(C.get_callback())
}
```

```
package main
import "fmt"
//export callback
import "C"
func callback() {
  fmt.Printf("Here!\n")
}
```

Callbacks – “Obvious” Drawbacks

YottaDB

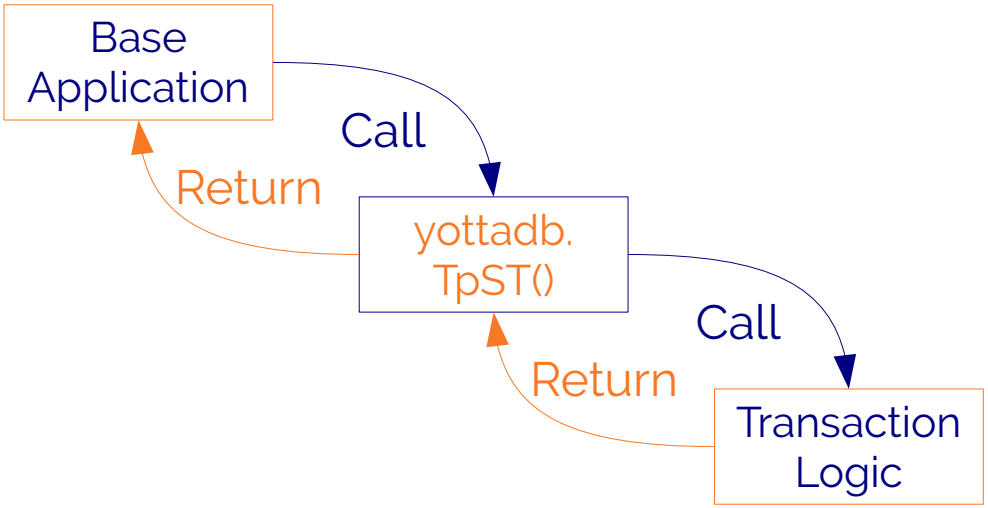
- Error prone
- Makes testing hard
  - Can’t import C in test code
- Feels kludgy

Callbacks – Alternative Approach

YottaDB

- Write a callback function to pass back an index to a hashmap that stores the Golang callback; use closure for passing parameters
- Less error prone, with small performance hit
- Still feels kludgy – but not so much to gophers

Callback for Transaction Processing



Needed for optimistic concurrency control

Closure

YottaDB

```
func helloWorld(i int) {
  (func() {
    fmt.Printf("i: %v", i)
  })()
}
```

Callbacks – Alternative Code Fragment

YottaDB

```
var tpIndex uint64
var tpMap sync.Map
// YdbTpStWrapper is private callback to wrap calls to a Go closure for TpST
//export ydbTpStWrapper
func ydbTpStWrapper(tptoken uint64, errstr *C.ydb_buffer_t, tpfnparm
unsafe.Pointer) int32 {
    var errbuff BufferT
    index := *((*uint64)(tpfnparm))
    v, ok := tpMap.Load(index)
    if !ok {
        panic("YDB: Could not find callback routine")
    }
    errbuff.BufferTFromPtr((unsafe.Pointer)(errstr))
    return (v.(func(uint64, *BufferT) int32))(tptoken, &errbuff)
}
```

Taming the CGO Dragons

YottaDB

- Issues may not show up until garbage collection is occurs
  - Sometimes only after days of intensive use
  - And even then only at random

Poking the CGO Dragons

YottaDB

- Trigger more garbage collection
  - export GOGC=1
- Catch for Go pointers being passed
  - export GODEBUG="cgocheck=2"
- Horrible. Mean. Ugly. Tests.

CGO Dragons are Slumbering, not Slain

YottaDB

- Go still believes it owns the process
  - Go developers know best because they develop Go primarily for their own use
- Another dragon not discussed here – signal handlers
- The cost of sharing a cave with a dragon is eternal vigilance
  - Test continuously, and then test some more


CARL
LUND
GREN
1963

YottaDB

Thank You!

yottadb.com