Going Infinite, handling 1M websockets connections in Go

Eran Yanay, Twistlock

The goal

Developing high-load Go server that is able to manage millions of concurrent connections

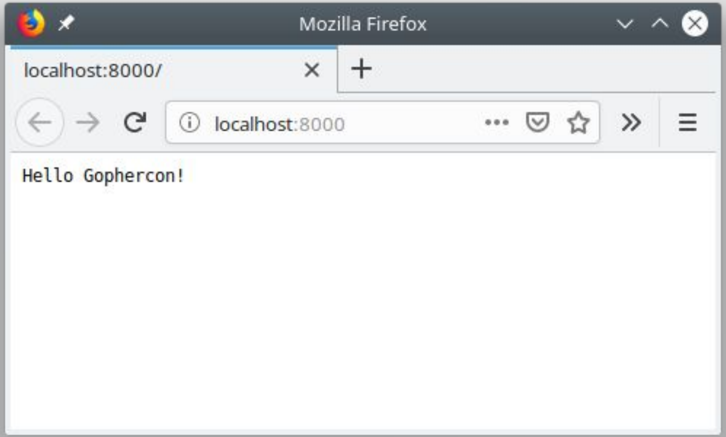
- How to write a webserver in Go?
- How to handle persistent connections?
- What limitations arise in scale?
- How to handle persistent connections efficiently?
  - OS limitations
  - Hardware limitations

How a Go web server works?

```go
package main
import (
    "io"
    "net/http"
)

func main() {
    http.HandleFunc("/", hello)
    http.ListenAndServe(":8000", nil)
}

func hello(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello Gophercon!")
}
```

How a Go web server works?

```go
package main
import (
    "io"
    "net/http"
)

func main() {
    http.HandleFunc("/", hello)
    http.ListenAndServe(":8000", nil)
}

func hello(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello Gophercon!")
}
```



How a Go web server works?

```go
// Serve accepts incoming connections on the Listener l, creating a
// new service goroutine for each. The service goroutines read requests and
// then call srv.Handler to reply to them.
func (srv *Server) Serve(l net.Listener) error {
    // ...
    for {
        rw, e := l.Accept()
        // ...
        c := srv.newConn(rw)
        c.setState(c.rwc, StateNew) // before Serve can return
        go c.serve(ctx)
    }
}
```

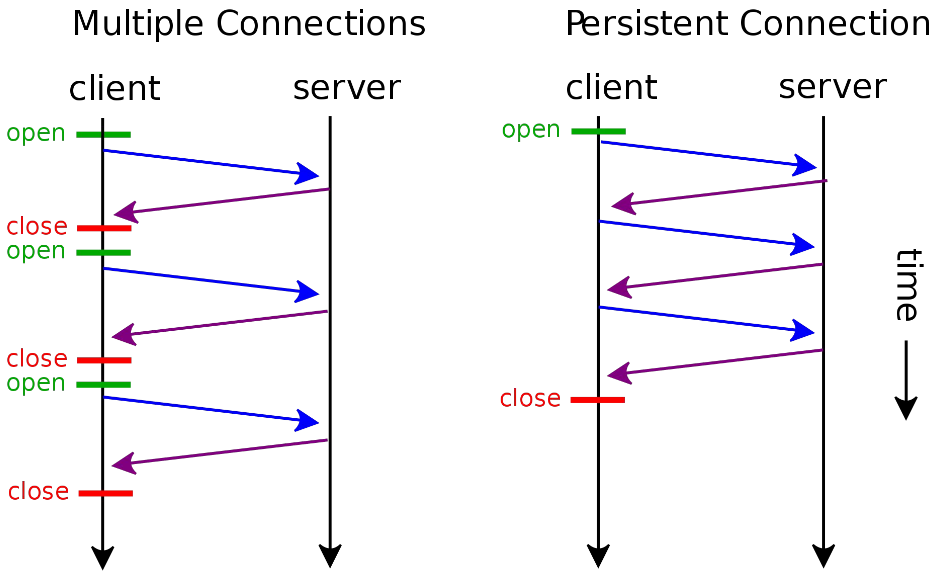
How a Go web server works?

```
// Serve accepts incoming connections on the Listener l, creating a
// new service goroutine for each. The service goroutines read requests and
// then call srv.Handler to reply to them.
func (srv *Server) Serve(l net.Listener) error {
    // ...
    for {
        rw, e := l.Accept()
        // ...
        c := srv.newConn(rw)
        c.setState(c.rwc, St
        go c.serve(ctx)
    }
}
```

```
func hello(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello Gophercon!")
}
```

The need for persistent connections

- Message queues
- Chat applications
- Notifications
- Social feeds
- Collaborative editing
- Location updates



What is a websocket?

WebSockets provide a way to maintain a full-duplex persistent connection between a client and server that both parties can start sending data at any time, with low overhead and latency

```
GET ws://websocket.example.com/ HTTP/1.1
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```


```
HTTP/1.1 101 WebSocket Protocol Handshake
Connection: Upgrade
Upgrade: WebSocket
```

Websockets in Go

net: golang.org/x/net/websocket — Index | Examples | Files

package websocket

```
import "golang.org/x/net/websocket"
```

Package websocket implements a client and server for the WebSocket protocol as specified in RFC 6455.

This package currently lacks some features found in an alternative and more actively maintained WebSocket package:

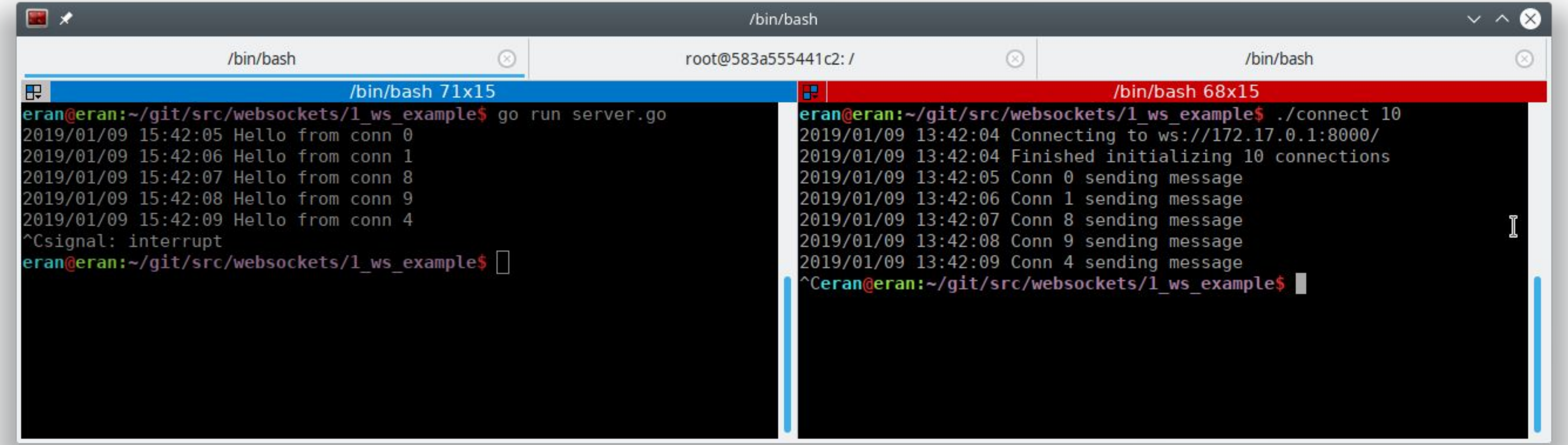
```
https://godoc.org/github.com/gorilla/websocket
```

Websockets in Go

```go
func ws(w http.ResponseWriter, r *http.Request) {
    // Upgrade connection
    upgrader := websocket.Upgrader{}
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        return
    }
    for {
        _, msg, err := conn.ReadMessage()
        if err != nil {
            log.Printf("Failed to read message %v", err)
            conn.Close()
            return
        }
        log.Println(string(msg))
    }
}
```

```go
func main() {
    http.HandleFunc("/", ws)
    http.ListenAndServe(":8000", nil)
}
```

Demo!



/bin/bash
/bin/bash
root@583a555441c2: /
/bin/bash
/bin/bash 71x15
eran@eran:~/git/src/websockets/1_ws_example$ go run server.go
2019/01/09 15:42:05 Hello from conn 0
2019/01/09 15:42:06 Hello from conn 1
2019/01/09 15:42:07 Hello from conn 8
2019/01/09 15:42:08 Hello from conn 9
2019/01/09 15:42:09 Hello from conn 4
^Csignal: interrupt
eran@eran:~/git/src/websockets/1_ws_example$
/bin/bash 68x15
eran@eran:~/git/src/websockets/1_ws_example$ ./connect 10
2019/01/09 13:42:04 Connecting to ws://172.17.0.1:8000/
2019/01/09 13:42:04 Finished initializing 10 connections
2019/01/09 13:42:05 Conn 0 sending message
2019/01/09 13:42:06 Conn 1 sending message
2019/01/09 13:42:07 Conn 8 sending message
2019/01/09 13:42:08 Conn 9 sending message
2019/01/09 13:42:09 Conn 4 sending message
^Ceran@eran:~/git/src/websockets/1_ws_example$

Demo! - Cont'd

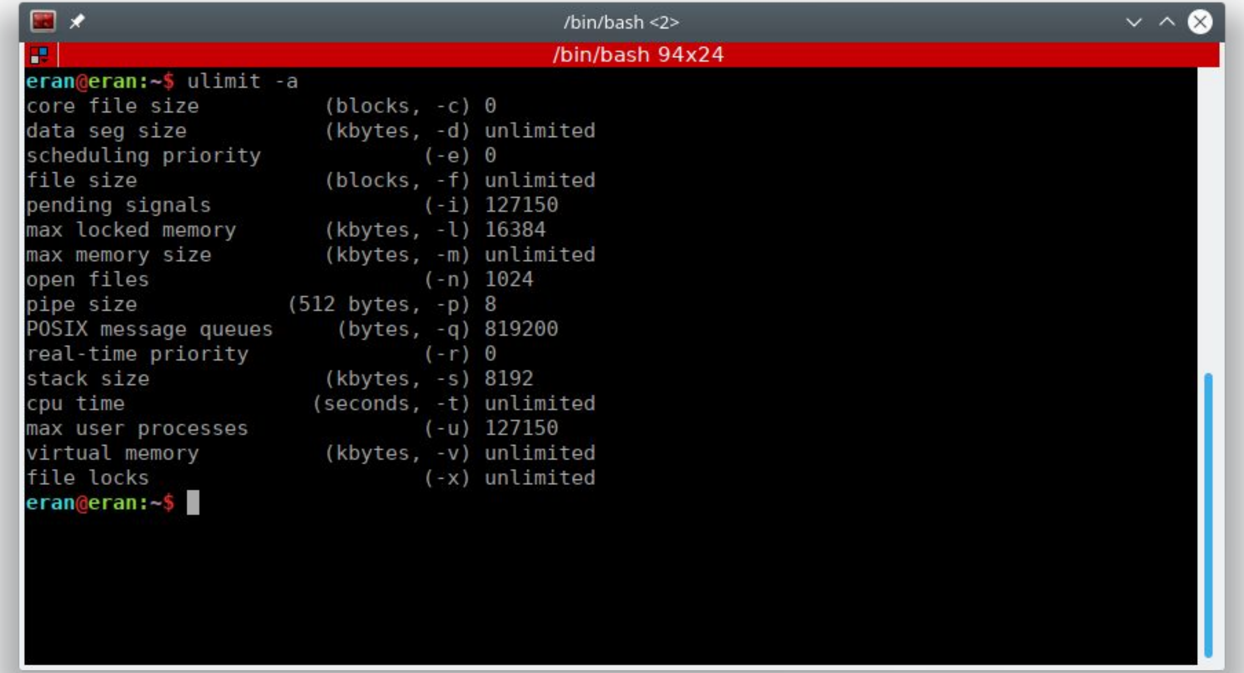
```
2019/01/09 15:55:36 Hello from conn 367
2019/01/09 15:55:36 Hello from conn 966
2019/01/09 15:55:36 Hello from conn 152
2019/01/09 15:55:36 Hello from conn 816
2019/01/09 15:55:36 Hello from conn 552
2019/01/09 15:55:36 Hello from conn 617
2019/01/09 15:55:36 Hello from conn 373
2019/01/09 15:55:36 Hello from conn 224
2019/01/09 15:55:36 Hello from conn 648
2019/01/09 15:55:36 Hello from conn 100
2019/01/09 15:55:36 Hello from conn 595
2019/01/09 15:55:36 Hello from conn 860
2019/01/09 15:55:36 Hello from conn 753
2019/01/09 15:55:36 Hello from conn 712
2019/01/09 15:55:36 Hello from conn 679
2019/01/09 15:55:36 Hello from conn 389
2019/01/09 15:55:36 Hello from conn 437
2019/01/09 15:55:36 Hello from conn 521
2019/01/09 15:55:37 Hello from conn 79
2019/01/09 15:55:37 Hello from conn 276
2019/01/09 15:55:37 Hello from conn 134
2019/01/09 15:55:37 Hello from conn 890
2019/01/09 15:55:37 Hello from conn 326
2019/01/09 15:55:37 Hello from conn 911
2019/01/09 15:55:37 Hello from conn 173
2019/01/09 15:55:37 Hello from conn 725
2019/01/09 15:55:37 Hello from conn 903
2019/01/09 15:55:37 Hello from conn 15
2019/01/09 15:55:37 Hello from conn 371
2019/01/09 15:55:37 Hello from conn 517
2019/01/09 15:55:37 Hello from conn 17
2019/01/09 15:55:37 http: Accept error: accept tcp [::]:8000: accept4: too many open files; retrying in 5ms
2019/01/09 15:55:37 http: Accept error: accept tcp [::]:8000: accept4: too many open files; retrying in 10ms
2019/01/09 15:55:37 http: Accept error: accept tcp [::]:8000: accept4: too many open files; retrying in 20ms
2019/01/09 15:55:37 http: Accept error: accept tcp [::]:8000: accept4: too many open files; retrying in 40ms
2019/01/09 15:55:37 Hello from conn 284
2019/01/09 15:55:37 http: Accept error: accept tcp [::]:8000: accept4: too many open files; retrying in 80ms
```

Too many open files

- Every socket is represented by a file descriptor
- The OS needs memory to manage each open file
- Memory is a limited resource
- Maximum number of open files can be changed via ulimits

Resources limit

Ulimit provides control over the resources available to processes



Resources limit

Ulimit provides control over the resources available to processes

- The kernel enforces the soft limit for the corresponding resource
- The hard limit acts as a ceiling for the soft limit
- Unprivileged process can only raise up to the hard limit
- Privileged process can make any arbitrary change
- *RLIMIT_NOFILE* is the resource enforcing max number of open files

Resources limit in Go

```go
func SetUlimit() error {
    var rLimit syscall.Rlimit
    if err := syscall.Getrlimit(syscall.RLIMIT_NOFILE, &rLimit); err !=nil {
        return err
    }
    rLimit.Cur = rLimit.Max
    return syscall.Setrlimit(syscall.RLIMIT_NOFILE, &rLimit)
}
```

Demo! (#2)

/bin/bash 122x26

```
2019/01/13 13:26:25 Hello from conn 7749
2019/01/13 13:26:25 Hello from conn 13670
2019/01/13 13:26:25 Hello from conn 13671
2019/01/13 13:26:25 Hello from conn 7750
2019/01/13 13:26:25 Hello from conn 7751
2019/01/13 13:26:25 Hello from conn 13672
2019/01/13 13:26:25 Hello from conn 7752
2019/01/13 13:26:25 Hello from conn 13673
2019/01/13 13:26:25 Hello from conn 7753
2019/01/13 13:26:25 Hello from conn 13674
2019/01/13 13:26:25 Hello from conn 7754
2019/01/13 13:26:25 Hello from conn 13675
2019/01/13 13:26:25 Hello from conn 7755
2019/01/13 13:26:25 Hello from conn 13676
2019/01/13 13:26:25 Hello from conn 7756
2019/01/13 13:26:25 Hello from conn 13677
2019/01/13 13:26:25 Hello from conn 7757
2019/01/13 13:26:25 Hello from conn 13678
2019/01/13 13:26:25 Hello from conn 7195
2019/01/13 13:26:25 Hello from conn 13679
2019/01/13 13:26:25 Hello from conn 7758
2019/01/13 13:26:25 Hello from conn 13680
2019/01/13 13:26:25 Hello from conn 7759
2019/01/13 13:26:25 Hello from conn 13681
2019/01/13 13:26:25 Hello from conn 7760
```

Memory consumption

```
/bin/bash 105x26
top - 13:43:02 up 1 day,  6:08,  2 users,  load average: 3.57, 4.78, 9.87
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  8.5 us,  6.4 sy,  0.0 ni, 83.6 id,  0.1 wa,  0.0 hi,  1.4 si,  0.0 st
KiB Mem : 32679576 total,   371020 free, 18873888 used, 13434668 buff/cache
KiB Swap: 31999996 total, 31993852 free,     6144 used. 12678268 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
24901 root      20   0 2503.5m 970.7m   7.2m S   9.3  3.0   2:05.45 server
```

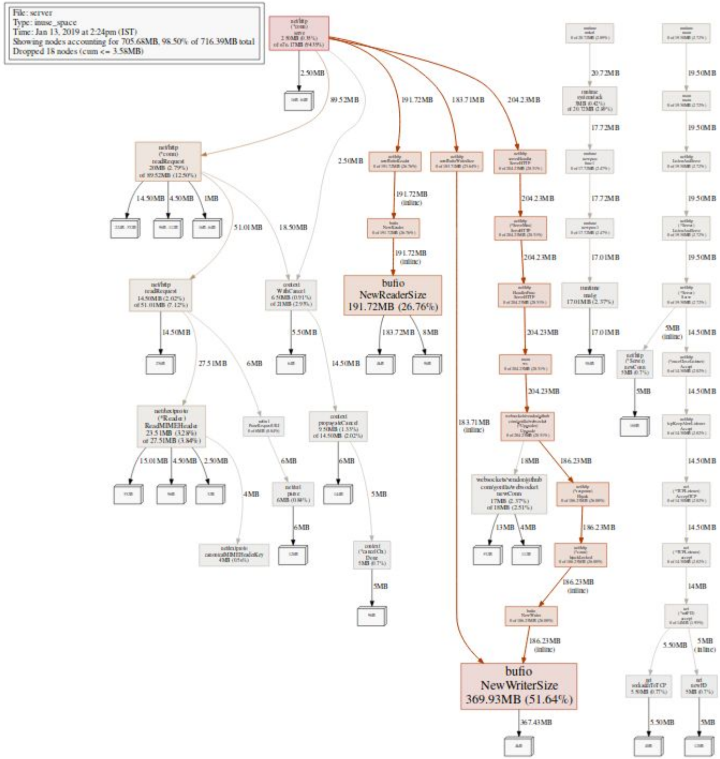
pprof

Package pprof serves via its HTTP server runtime profiling data in the format expected by the pprof visualization tool.

```
import _ "net/http/pprof"

go func() {
    if err := http.ListenAndServe("localhost:6060", nil); err != nil {
        log.Fatalf("Pprof failed: %v", err)
    }
}()
```

- Analyze heap memory: `go tool pprof http://localhost:6060/debug/pprof/heap`
- Analyze goroutines: `go tool pprof http://localhost:6060/debug/pprof/goroutine`

pprof - Demo!



File: server
Type: inuse_space
Time: Jan 13, 2019 at 2:24pm (IST)
Showing nodes accounting for 705.68MB, 98.50% of 716.39MB total
Dropped 18 nodes (cum <= 3.58MB)
bufio
NewReaderSize
191.72MB (26.76%)
bufio
NewWriterSize
369.93MB (51.64%)

Memory consumption

Each connection in the naive solution consumes ~20KB:

$$Mem = conns \cdot (goroutine + buf_{net/http} + buf_{gorilla/ws})$$

Memory consumption

Each connection in the naive solution consumes ~20KB:

$$Mem = conns \cdot (goroutine + buf_{net/http} + buf_{gorilla/ws})$$

```go
func ws(w http.ResponseWriter, r *http.Request) {
    // ...
}
```

Memory consumption

Each connection in the naive solution consumes ~20KB:

$$Mem = conns \cdot (goroutine + buf_{net/http} + buf_{gorilla/ws})$$

```
func ws(w http.ResponseWriter, r *http.Request) {
    // ...
}
```

```
upgrader := websocket.Upgrader{}
conn, err := upgrader.Upgrade(w, r, nil)
if err != nil {
    return
}
```

Memory consumption

Each connection in the naive solution consumes ~20KB:

$$Mem = conns \cdot (goroutine + buf_{net/http} + buf_{gorilla/ws})$$

```go
func ws(w http.ResponseWriter, r *http.Request) {
    // ...
}
```

```go
upgrader := websocket.Upgrader{}
conn, err := upgrader.Upgrade(w, r, nil)
if err != nil {
    return
}
```

Serving a million concurrent connections would consume over 20GB of RAM!

Optimizations

If we could…

- Optimize goroutines
- Optimize net/http objects allocations
- Reuse allocated buffers across websockets read/write

$$Mem \approx conns$$

Optimization #1: goroutines

Knowing when data exists on the wire would allow us to reuse goroutines and reduce memory footprint

- goroutines
- select / poll
- epoll

Optimization #1: goroutines

Knowing when data exists on the wire would allow us to reuse goroutines and reduce memory footprint

- **goroutines**
- select / poll
- epoll

```go
func ws(w http.ResponseWriter, r *http.Request) {
    // Upgrade connection …

    for {
        _, msg, err := conn.ReadMessage()
        if err != nil {
            log.Printf("Failed to read message %v", err)
            conn.Close()
            return
        }
        log.Println(string(msg))
    }
}
```

Optimization #1: goroutines

Knowing when data exists on the wire would allow us to reuse goroutines and reduce memory footprint

- goroutines
- **select / poll**
- epoll

```
t := &syscall.Timeval{ /* timeout for the call */ }
if _, err := syscall.Select(maxFD+1, fds, nil, nil, t); err != nil {
    return nil, err
}

for _, fd := range fds {
    if fdIsSet(fdset, fd) {
        // Consume data
    }
}
```

Optimization #1: goroutines

Knowing when data exists on the wire would allow us to reuse goroutines and reduce memory footprint

- goroutines
- select / poll
- **epoll**

```
epfd, _ := unix.EpollCreate1(0)

_ := unix.EpollCtl(epfd, syscall.EPOLL_CTL_ADD, fd,
        &unix.EpollEvent{Events: unix.POLLIN | unix.POLLHUP, Fd: fd})

events := make([]unix.EpollEvent, 100)
n, _ := unix.EpollWait(e.fd, events, 100)

for i := 0; i < n; i++ {
   // Consume data from connection who's fd is events[i].Fd
}
```

Epoll - Demo!

```
fd, err := unix.EpollCreate1(0)
if err != nil {
    return nil, err
}
fd := websocketFD(conn)
err := unix.EpollCtl(e.fd, syscall.EPOLL_CTL_ADD, fd, &unix.EpollEvent{Events: unix.POLLIN | unix.POLLHUP, Fd:
int32(fd)})
if err != nil {
    return err
}
```

Epoll - Results

$$Mem = conns \cdot buf_{gorilla/ws}$$

We managed to reduce the memory consumption by **~30%**

But..is it enough?

Optimization #2: buffers allocations

gorilla/websocket keeps a reference to the underlying buffers given by Hijack()

```
var br *bufio.Reader
if u.ReadBufferSize == 0 && bufioReaderSize(netConn, brw.Reader) > 256 {
    // Reuse hijacked buffered reader as connection reader.
    br = brw.Reader
}

buf := bufioWriterBuffer(netConn, brw.Writer)

var writeBuf []byte
if u.WriteBufferPool == nil && u.WriteBufferSize == 0 && len(buf) >= maxFrameHeaderSize+256 {
    // Reuse hijacked write buffer as connection buffer.
    writeBuf = buf
}
c := newConn(netConn, true, u.ReadBufferSize, u.WriteBufferSize, u.WriteBufferPool, br, writeBuf)
```

Optimization #2: buffers allocations

github.com/gobwas/ws - alternative websockets library for Go

- No intermediate allocations during I/O
- Low-level API which allows building logic of packet handling and buffers
- Zero-copy upgrades

```go
import "github.com/gobwas/ws"

func wsHandler(w http.ResponseWriter, r *http.Request) {
    conn, _, _, err := ws.UpgradeHTTP(r, w)
    if err != nil {
        return
    }
    // Add to epoll
}
```

```go
for {
    // Fetch ready connections with epoll logic
    msg, _, err := wsutil.ReadClientData(conn)
    if err == nil {
        log.Printf("msg: %s", string(msg))
    } else {
        // Close connection
    }
}
```

gobwas/ws - Demo!

```
ubuntu18: ~ 104x59
2019/02/03 16:14:41 Total number of connections: 95800
2019/02/03 16:14:41 Total number of connections: 95900
2019/02/03 16:14:41 Total number of connections: 96000
2019/02/03 16:14:41 Total number of connections: 96100
2019/02/03 16:14:41 Total number of connections: 96200
2019/02/03 16:14:41 Total number of connections: 96300
2019/02/03 16:14:41 Total number of connections: 96400
2019/02/03 16:14:42 Total number of connections: 96500
2019/02/03 16:14:42 Total number of connections: 96600
2019/02/03 16:14:42 Total number of connections: 96700
2019/02/03 16:14:42 Total number of connections: 96800
2019/02/03 16:14:42 Total number of connections: 96900
2019/02/03 16:14:42 Total number of connections: 97000
2019/02/03 16:14:42 Total number of connections: 97100
2019/02/03 16:14:42 msg: Hello from conn 0
2019/02/03 16:14:42 msg: Hello from conn 1
2019/02/03 16:14:42 msg: Hello from conn 2
2019/02/03 16:14:42 msg: Hello from conn 3
2019/02/03 16:14:42 msg: Hello from conn 4
2019/02/03 16:14:42 msg: Hello from conn 5
2019/02/03 16:14:42 msg: Hello from conn 6
2019/02/03 16:14:42 msg: Hello from conn 7
2019/02/03 16:14:43 msg: Hello from conn 8
2019/02/03 16:14:43 msg: Hello from conn 9
2019/02/03 16:14:43 msg: Hello from conn 10
```

Buffer allocations - Results

$$Mem \eqsim conns$$

We managed to reduce the memory usage by **97%**

Serving over a million connections is now reduced from **~20GB** to **~600MB**

Recap..

Premature optimization is the root of all evil, but if we must:

- Ulimit: Increase the cap of NOFILE resource
- Epoll (Async I/O): Reduce the high load of goroutines
- Gobwas - More performant ws library to reduce buffer allocations
- Conntrack table - Increase the cap of total concurrent connections in the OS

Thank you!

Code examples are available at https://github.com/eranyanay/1m-go-websockets

Questions?

Twistlock