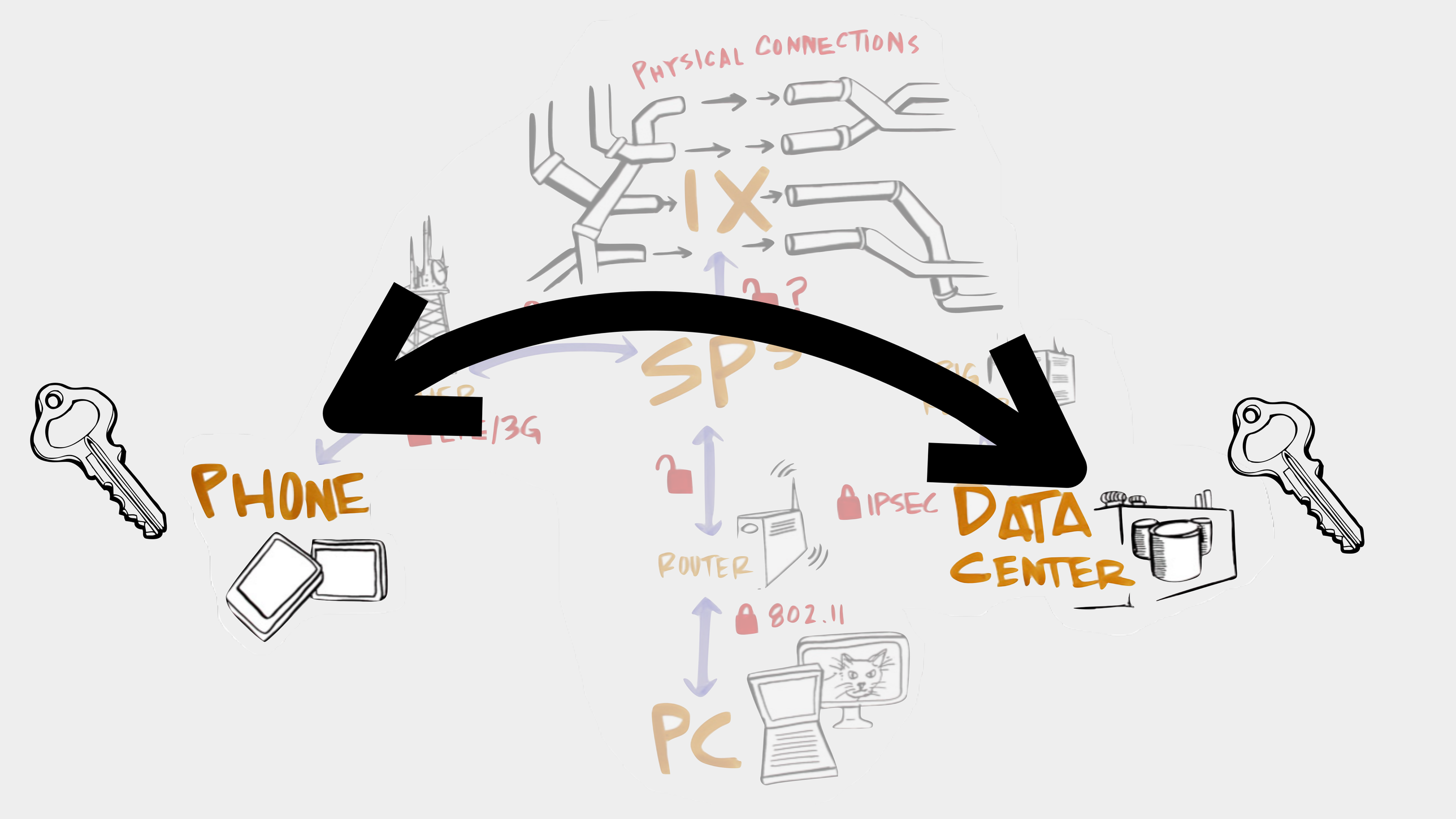
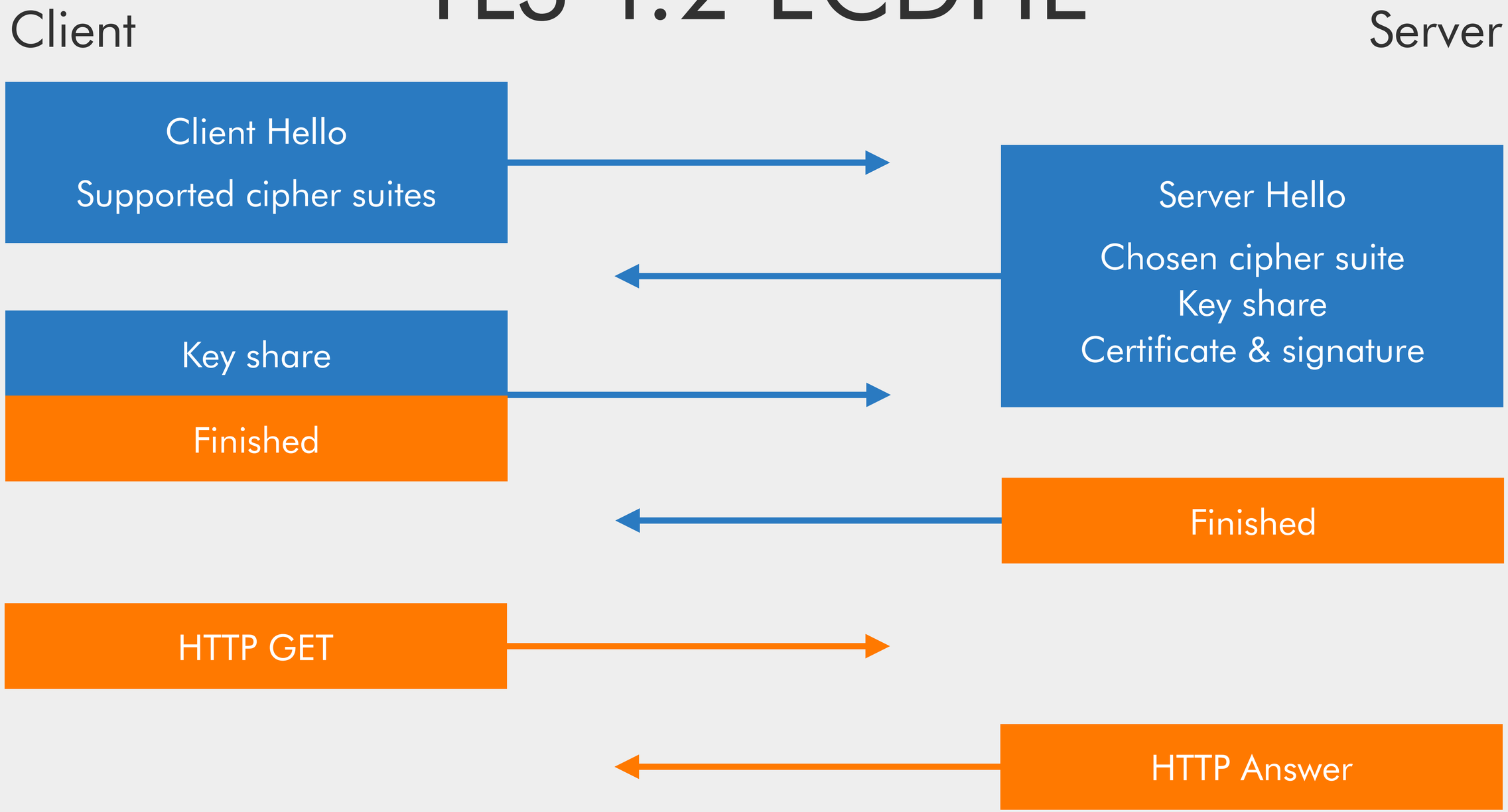
Encrypting the Internet with Go

Filippo Valsorda


PHYSICAL CONNECTIONS
IX
SP
LTE/3G
PHONE
ROUTER
IPSEC
DATA CENTER
802.11
PC

TLS 1.2 ECDHE

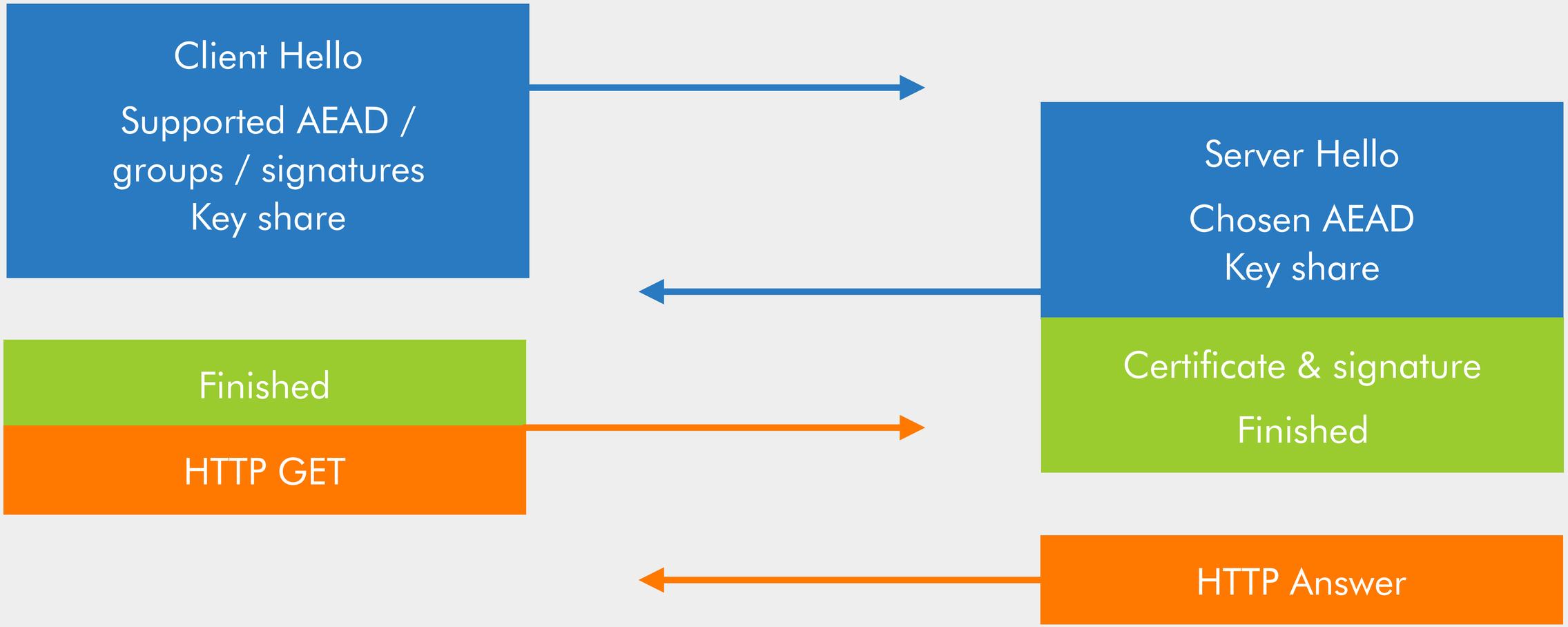

Client
Server
Client Hello
Supported cipher suites
Server Hello
Chosen cipher suite
Key share
Certificate & signature
Key share
Finished
Finished
HTTP GET
HTTP Answer

TLS ~~2~~ ~~2017~~ 1.3

~~Vista~~ ~~4~~

Client

Server



0-RTT


Client
Server
Client Hello
Session Ticket (PSK)
Key share
HTTP GET
Server Hello
Key share
Finished
HTTP Answer
Finished

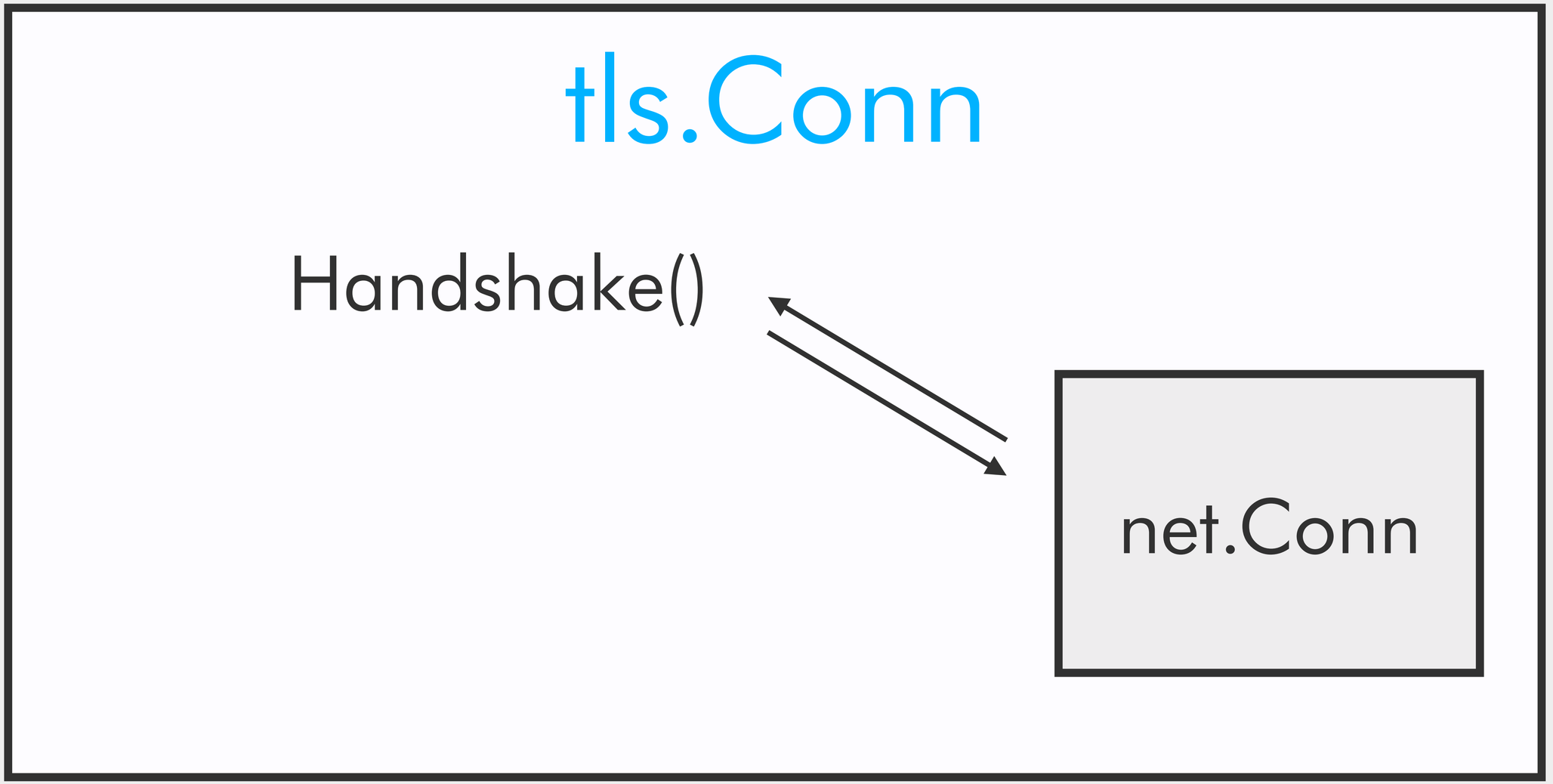
crypto/tls += TLS 1.3

```
package tls // import "crypto/tls"


type Conn struct {
        // Has unexported fields.
}
    A Conn represents a secured connection. It implements
    the net.Conn interface.


func Client(conn net.Conn, config *Config) *Conn

func Server(conn net.Conn, config *Config) *Conn
```

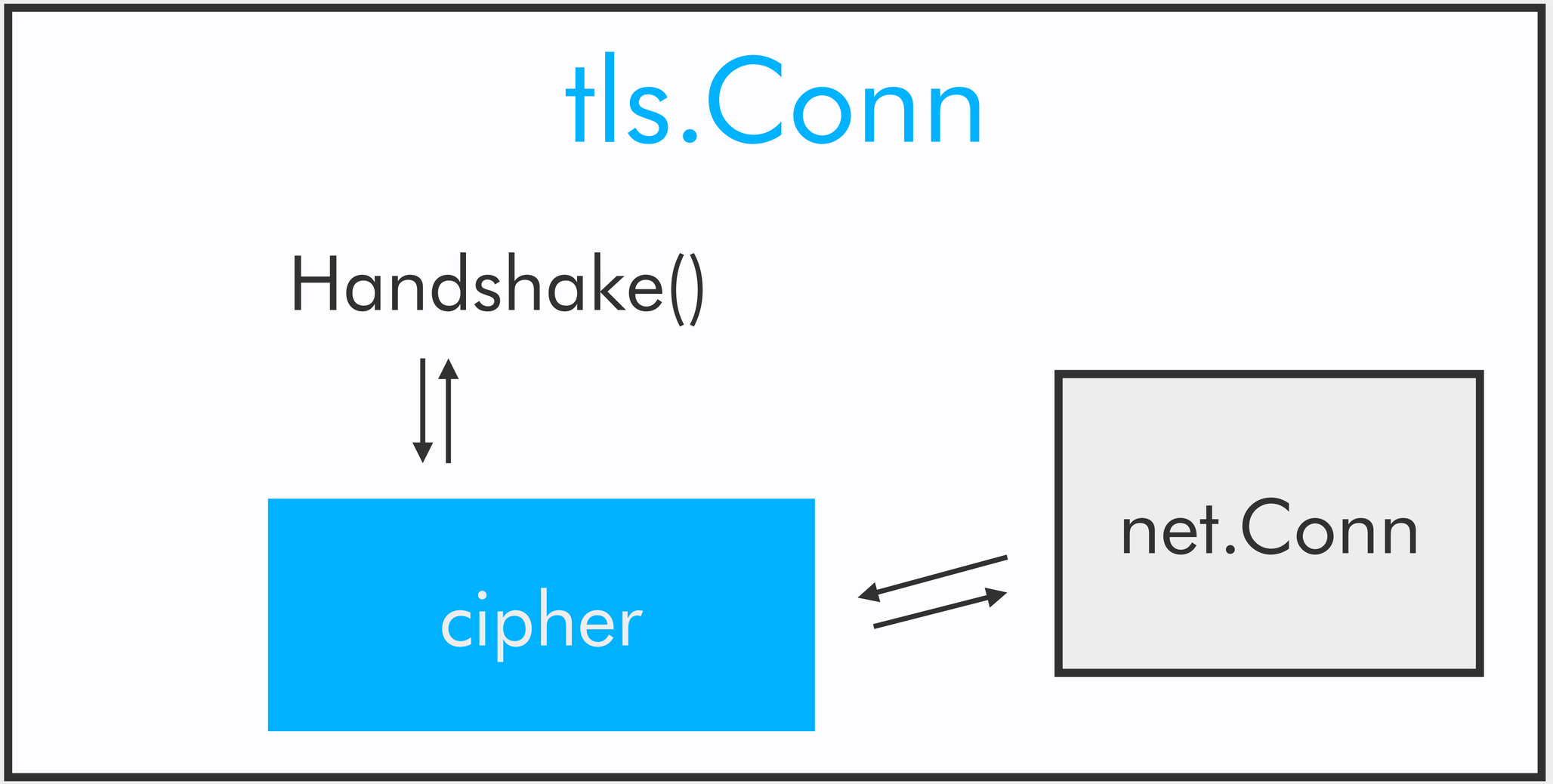
```
func (c *Conn) Handshake() error
```

```
func (c *Conn) Read(b []byte) (n int, err error) {
    if err = c.Handshake(); err != nil {
        return
    }
```

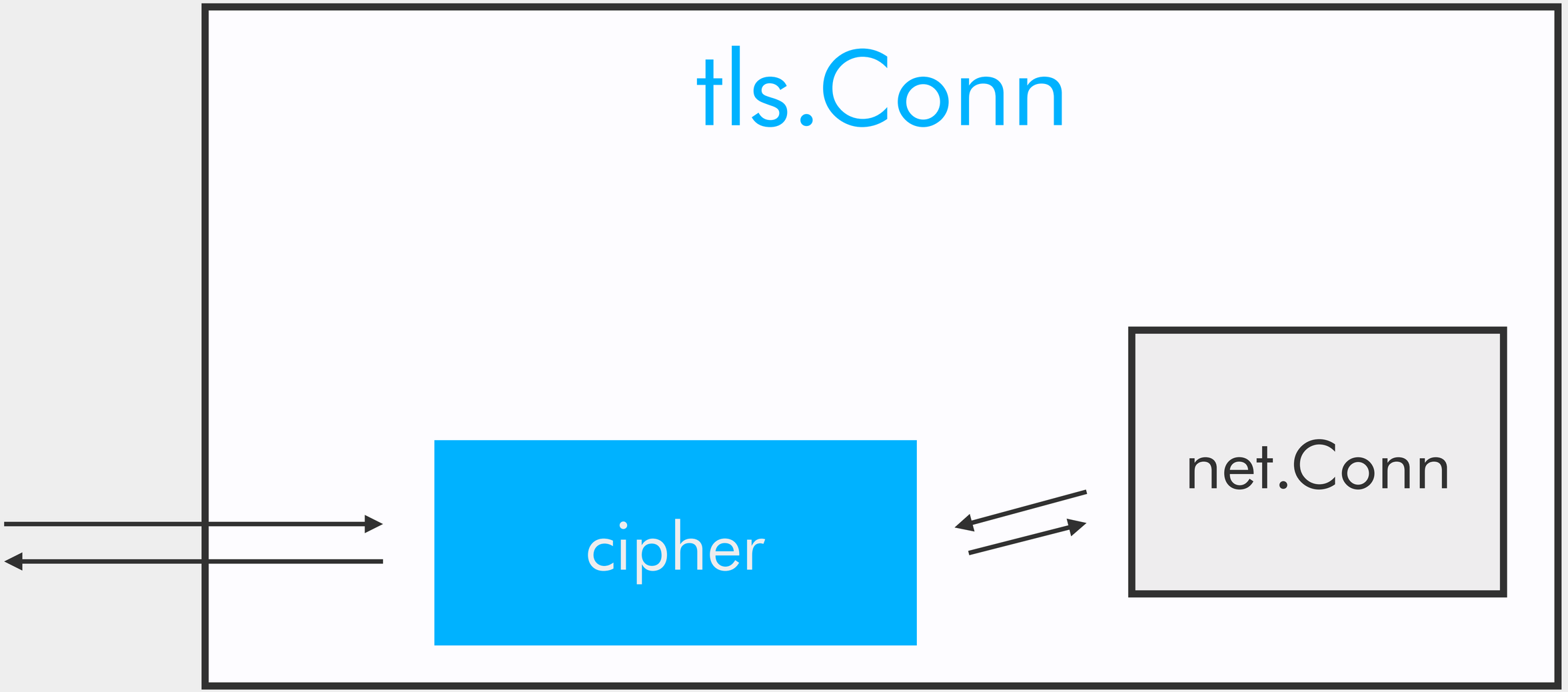
Initial Handshake



Late Handshake



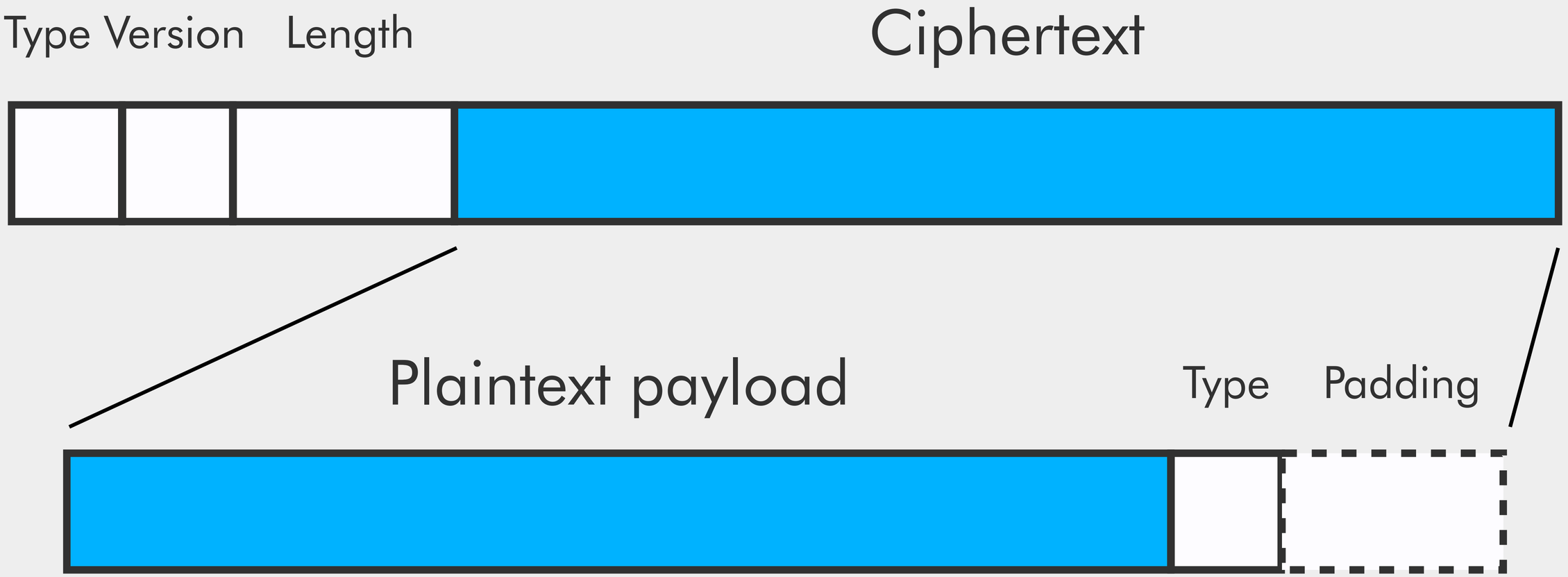
Handshake Complete



```
func (c *Conn) serverHandshake() error {

    msg, err := c.readHandshake()
    if err != nil {
        return false, err
    }
```

```
func (c *Conn) readHandshake() (interface{}, error) {

    for c.hand.Len() < 4 {
        if err := c.readRecord(recordTypeHandshake); err != nil {
            return nil, err
        }
    }
    data := c.hand.Bytes()
    n := int(data[1])<<16 | int(data[2])<<8 | int(data[3])
    for c.hand.Len() < 4+n {
        if err := c.readRecord(recordTypeHandshake); err != nil {
            return nil, err
        }
    }
```

TLS record layer



```
func (hc *halfConn) decrypt(b *block) (ok bool) {
    payload := b.data[recordHeaderLen:]

    switch c := hc.cipher.(type) {
    case cipher.Stream:
        c.XORKeyStream(payload, payload)
    case aead:
        explicitIVLen := c.explicitNonceLen()
        nonce := payload[:explicitIVLen]
        payload = payload[explicitIVLen:]
        payload, err = c.Open(payload[:0], nonce, payload, nil)
        if err != nil {
            return false, 0, alertBadRecordMAC
        }
        b.resize(recordHeaderLen + len(payload))
    }
```

```
func (c *Conn) readRecord(want recordType) error {

    switch typ {
    case recordTypeAlert:
        c.in.setErrorLocked(&net.OpError{
            Op: "remote error", Err: alert(data[1])})

    case recordTypeApplicationData:
        c.input = b
        b = nil

    case recordTypeHandshake:
        c.hand.Write(data)
    }
```

```
func (c *Conn) readHandshake() (interface{}, error) {

var m handshakeMessage
switch data[0] {
case typeClientHello:
    m = new(clientHelloMsg)
case typeServerHello:
    m = new(serverHelloMsg)
case typeNewSessionTicket:
    m = new(newSessionTicketMsg)
case typeCertificate:
    m = new(certificateMsg)
...

m.unmarshal(data)
```

```
func (c *Conn) serverHandshake() error {

    clientFinished, ok := msg.(*finishedMsg)
    if !ok {
        c.sendAlert(alertUnexpectedMessage)
        return unexpectedMessageErr(clientFinished, msg)
    }
```

TLS 1.3 handshake states

```
type handshakeStatus int

const (
    handshakeRunning handshakeStatus = iota
    discardingEarlyData
    readingEarlyData
    waitingClientFinished
    readingClientFinished
    handshakeConfirmed
)
```

0-RTT API

0-RTT data is different, because it might be replayed.

The server needs:

- To know what part of the data is 0-RTT
- To know when it's safe to use the 0-RTT data

Option 1: just a Config knob

```
type Config struct {
    Accept0RTTData bool
```

No way of knowing which part is the 0-RTT data.

Option 2: a separate function

```
func (*Conn) HandshakeWith0RTT() (io.Reader, error)
```

Breaks io.Reader!

Imagine wrapping a tls.Conn in gzip.Reader.

Option 3: a check function

```
type Config struct {
    IsORTTSafe func(io.Reader) bool
```

Subtly breaks io.Reader.

You'd have to manually parse HTTP.

Option 4: a ConnectionState field

```
func (c *Conn) ConnectionState() ConnectionState

type ConnectionState struct {
    HandshakeConfirmed bool
```

Ok, but how do I wait for it to be “confirmed”?

Option 5: ConfirmHandshake

```
func (c *Conn) ConfirmHandshake() error
```

You don't actually know which part exactly is 0-RTT, but you know if what you read so far is safe, and can wait otherwise. (Sadly, it must buffer in the slow path.)

Exposing it to the HTTP handler

The HTTP handler needs access to the live ConnectionState and the ConfirmHandshake method.

```
TLSConnContextKey = &contextKey{"tls-conn"}
```

Like ServerContextKey and LocalAddrContextKey.

Other API changes

None.

TLS 1.3 is about sane defaults, so I'd like to see how far we get with just that.

Full diff: https://gist.github.com/FiloSottile/37d6516af411582e2aa35a981bf12102

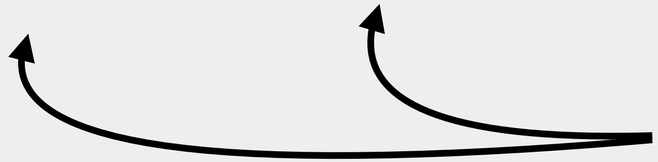
Read() Write() Handshake() ConfirmHandshake() Close()

Read() Write() Handshake() ConfirmHandshake() Close()

c.in.Mutex c.out.Mutex c.handshakeMutex

Read() Write() Handshake() ConfirmHandshake() Close()

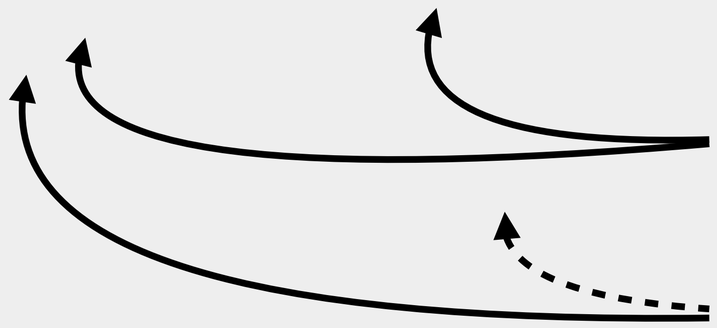
c.in.Mutex c.out.Mutex c.handshakeMutex



blocks

Read() Write() Handshake() ConfirmHandshake() Close()

c.in.Mutex c.out.Mutex c.handshakeMutex



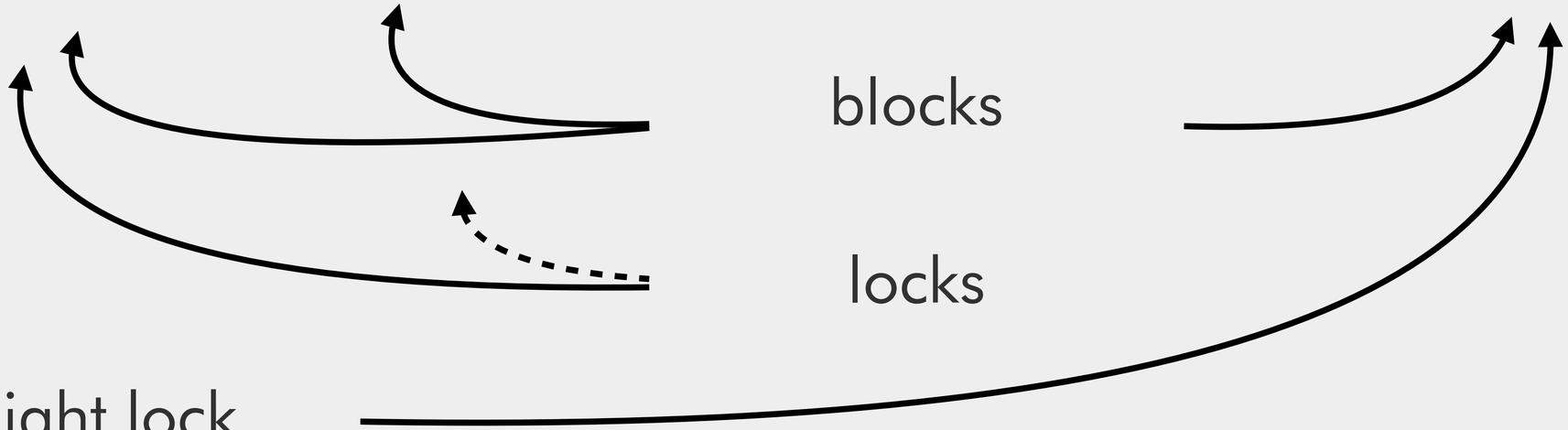
blocks

locks


Read()
Write()
Handshake()
ConfirmHandshake()
Close()
c.in.Mutex
c.out.Mutex
c.handshakeMutex
blocks
locks

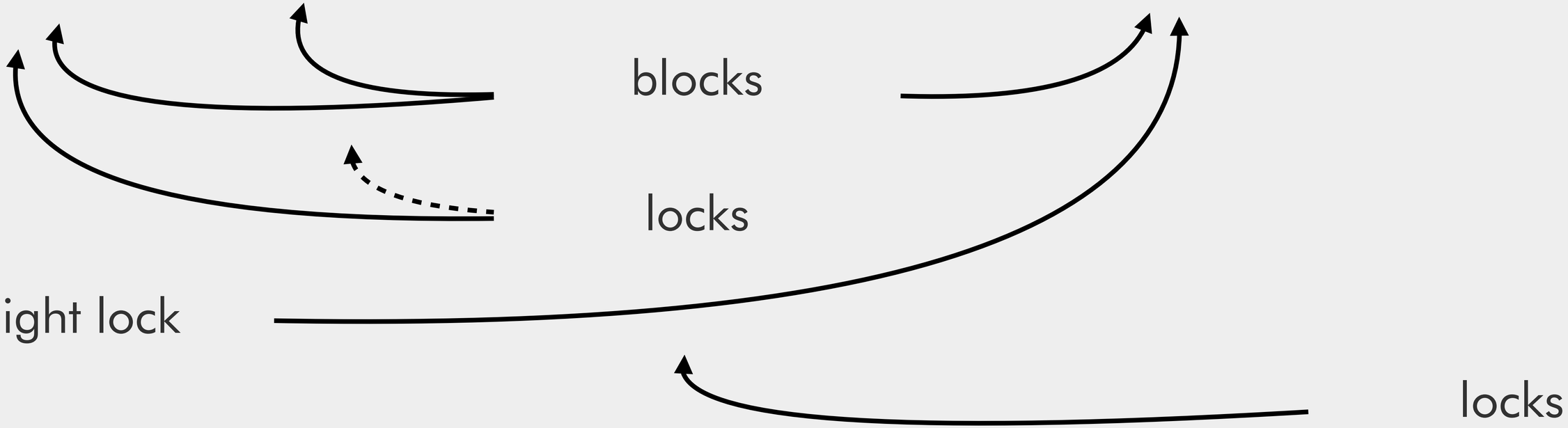
Read() Write() Handshake() ConfirmHandshake() Close()

c.in.Mutex c.out.Mutex c.handshakeMutex c.confirmMutex

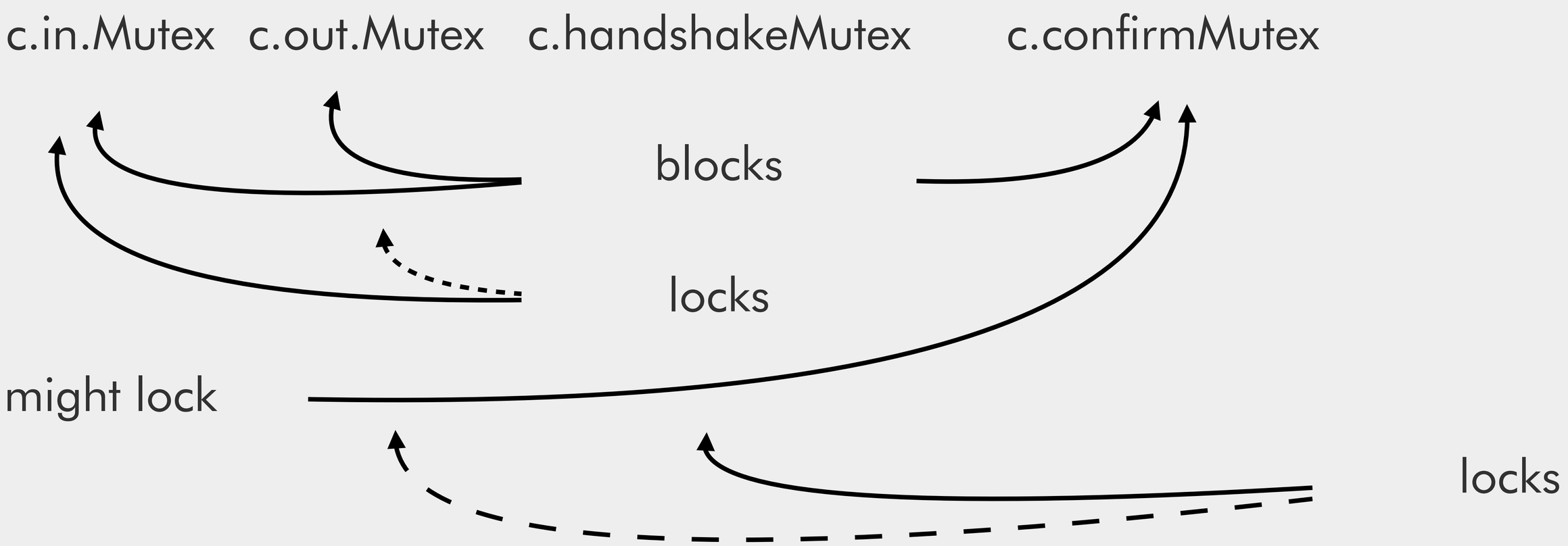


Read() Write() Handshake() ConfirmHandshake() Close()

c.in.Mutex c.out.Mutex c.handshakeMutex c.confirmMutex



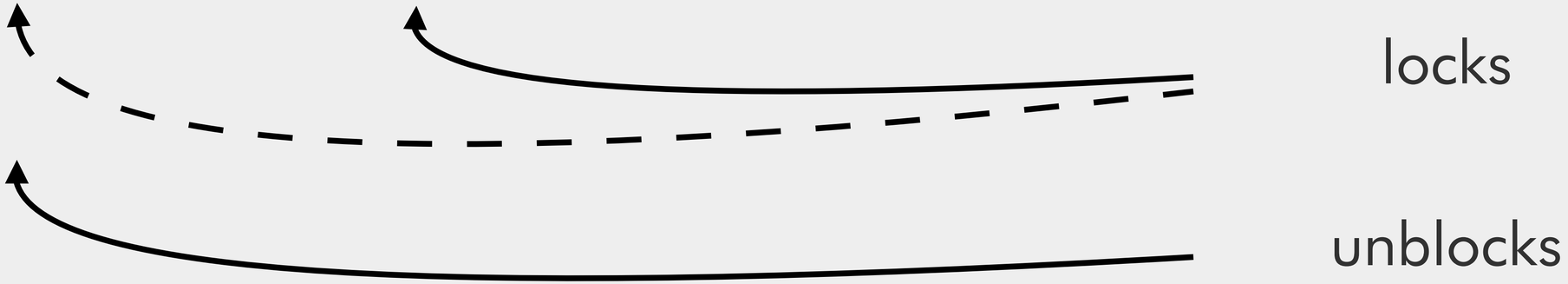
Read() Write() Handshake() ConfirmHandshake() Close()



Read() Write() Handshake() ConfirmHandshake() Close()

c.in.Mutex c.out.Mutex c.handshakeMutex c.confirmMutex

might lock



Read() Write() Handshake() ConfirmHandshake() Close()

Build Jobs

✓	# 206.1	Go: 1.7	MODE=interop CLIENT=boring	5 min 26 sec
✓	# 206.2	Go: 1.7	MODE=interop CLIENT=bogo	4 min 9 sec
✓	# 206.3	Go: 1.7	MODE=interop CLIENT=tstclnt	6 min 49 sec
✓	# 206.4	Go: 1.7	MODE=interop CLIENT=picotls ZRTT=1	3 min 28 sec
✓	# 206.5	Go: 1.7	MODE=interop CLIENT=mint	2 min 57 sec
✓	# 206.6	Go: 1.7	MODE=gotest	3 min 3 sec

Allowed Failures ?

✗	# 206.7	Go: 1.7	MODE=interop CLIENT=tstclnt ZRTT=1	6 min 21 sec
✓	# 206.8	Go: 1.7	MODE=interop CLIENT=boring REVISION=	5 min 17 sec
✗	# 206.9	Go: 1.7	MODE=interop CLIENT=tstclnt REVISION=	6 min 56 sec

Interoperability testing

```
IP=$(docker inspect -f '{{ .NetworkSettings.IPAddress }}' tris-localserver)

docker run --rm tls-tris:$2 $IP:3443 | tee output.txt # rejecting 0-RTT
grep "Hello TLS 1.3" output.txt | grep "resumed" | grep -v "0-RTT"

docker run --rm tls-tris:$2 $IP:4443 | tee output.txt # accepting 0-RTT
grep "Hello TLS 1.3" output.txt | grep "resumed" | grep "0-RTT"

docker run --rm tls-tris:$2 $IP:5443 | tee output.txt # confirming 0-RTT
grep "Hello TLS 1.3" output.txt | grep "resumed" | grep "0-RTT confirmed”
```

Patching the standard library

```
.PHONY: GOROOT
GOROOT: GOROOT/$(GOENV)/.ok_$(GOROOTINFO)
    @rm -f GOROOT/$(GOENV)/pkg/*/crypto/tls.a

GOROOT/$(GOENV)/.ok_$(GOROOTINFO):
    rm -rf GOROOT/$(GOENV)
    cp -r "$(shell $(GO) env GOROOT)/src" GOROOT/$(GOENV)/src
    cp -r "$(shell $(GO) env GOROOT)/pkg/include" GOROOT/$(GOENV)/pkg/include
    cp -r "$(shell $(GO) env GOROOT)/pkg/tool" GOROOT/$(GOENV)/pkg/tool
    rm -r GOROOT/$(GOENV)/src/crypto/tls
    ln -s ../../../../.. GOROOT/$(GOENV)/src/crypto/tls
    GOROOT="$(CURDIR)/GOROOT/$(GOENV)" $(GO) install -v std
    @touch "$@"
```

Patching the standard library

```
GO ?= go
GOENV := $(shell $(GO) env GOHOSTOS)_$(shell $(GO) env GOHOSTARCH)
GOROOTINFO := $(shell $(GO) version | cut -d' ' -f 3)_$(GOENV)

.PHONY: GOROOT
GOROOT: GOROOT/$(GOENV)/.ok_$(GOROOTINFO)
    @rm -f GOROOT/$(GOENV)/pkg/*/crypto/tls.a

GOROOT/$(GOENV)/.ok_$(GOROOTINFO):
    rm -rf GOROOT/$(GOENV)
    mkdir -p GOROOT/$(GOENV)/pkg
    cp -r "$(shell $(GO) env GOROOT)/src" GOROOT/$(GOENV)/src
    cp -r "$(shell $(GO) env GOROOT)/pkg/include" GOROOT/$(GOENV)/pkg/include
    cp -r "$(shell $(GO) env GOROOT)/pkg/tool" GOROOT/$(GOENV)/pkg/tool
    rm -r GOROOT/$(GOENV)/src/crypto/tls
    ln -s ../../../../.. GOROOT/$(GOENV)/src/crypto/tls
    GOROOT="$(CURDIR)/GOROOT/$(GOENV)" $(GO) install -v std
ifeq ($(shell go env CGO_ENABLED),1)
    GOROOT="$(CURDIR)/GOROOT/$(GOENV)" $(GO) install -race -v std
endif
    @touch "$@"

# Note: when changing this, if it doesn't change the Go version
# (it should), you need to run make clean.
GO_COMMIT := 98882e950cbb48ce7aad4b5b9972601d3438fbe5

.PHONY: go
go: go/.ok_$(GO_COMMIT)_$(GOENV)

go/.ok_$(GO_COMMIT)_$(GOENV):
    rm -rf go/.ok_*_$(GOENV) go/$(GOENV)
    mkdir -p go
    git clone --branch 1.8 --single-branch --depth 25 https://github.com/cloudflare/go go/$(GOENV)
    cd go/$(GOENV) && git checkout $(GO_COMMIT)
    cd go/$(GOENV)/src && GOROOT_BOOTSTRAP="$(shell $(GO) env GOROOT)" ./make.bash
    @touch "$@"

.PHONY: clean
clean:
    rm -rf GOROOT go
```

Patching the standard library

```
#!/usr/bin/env bash
set -e

BASEDIR=$(cd "$(dirname "$0")" && pwd)

make --quiet -C "$BASEDIR" go >&2
GOENV="$(go env GOHOSTOS)_$(go env GOHOSTARCH)"

export GOROOT="$BASEDIR/go/$GOENV"
make --quiet -C "$BASEDIR" GOROOT GO="$BASEDIR/go/$GOENV/bin/go" >&2
export GOROOT="$BASEDIR/GOROOT/$GOENV"

exec $BASEDIR/go/$GOENV/bin/go "$@"
```

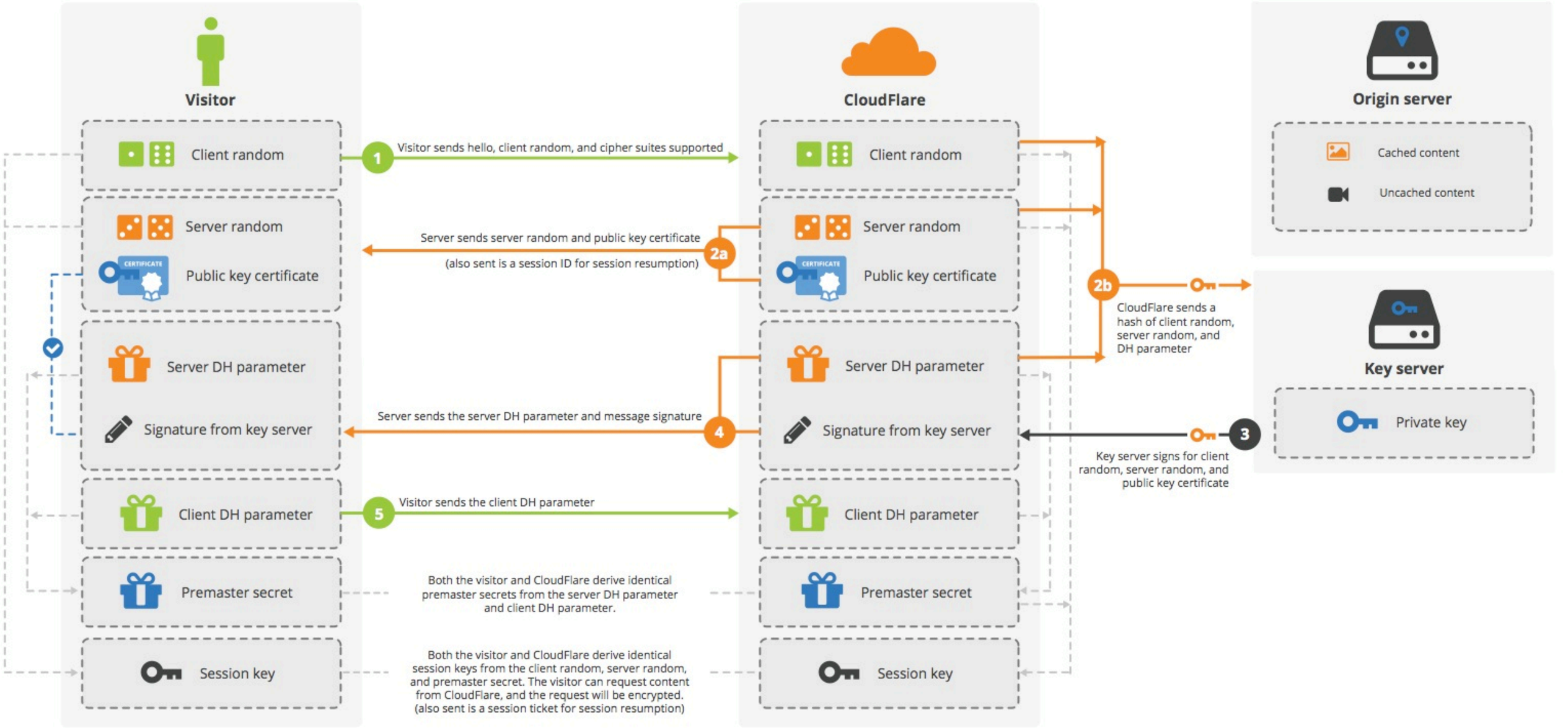
crypto/tls in production

GetCertificate

```
type Config struct {
    GetCertificate func(*ClientHelloInfo) (*Certificate, error)
```

CloudFlare Keyless SSL (Diffie-Hellman)

Handshake



Keyless and GetCertificate

```
type Certificate struct {
        Certificate [][]byte

        PrivateKey crypto.PrivateKey // crypto.Signer / crypto.Decrypter

        OCSPStaple []byte
        SignedCertificateTimestamps [][]byte

        Leaf *x509.Certificate
}
```

See github.com/cloudflare/gokeyless/client

Where GetCertificate can’t go

```
type Config struct {
        NextProtos []string
        ClientAuth ClientAuthType
        AcceptORTTData bool
```

Enter GetConfigForClient

```
type Config struct {
        GetConfigForClient func(*ClientHelloInfo) (*Config, error)
```

https://golang.org/issues/15707

With some extra ClientHelloInfo

```
 type ClientHelloInfo struct {
     CipherSuites      []uint16
     ServerName        string
     SupportedCurves   []CurveID
     SupportedPoints   []uint8
+    SignatureSchemes  []uint16
+    SupportedProtos   []string
+    SupportedVersions []uint16
+    Conn              net.Conn
 }
```

https://golang.org/issues/17430

Last remaining: session tickets

```
// A SessionTicketWrapper provides a way to securely incapsulate
// session state for storage on the client.
type SessionTicketWrapper interface {
    Wrap(cs *ConnectionState, content []byte) ([]byte, error)
    Unwrap(chi *ClientHelloInfo, ticket []byte) ([]byte, bool)
    Clone() SessionTicketWrapper
}
```

https://golang.org/issues/19199

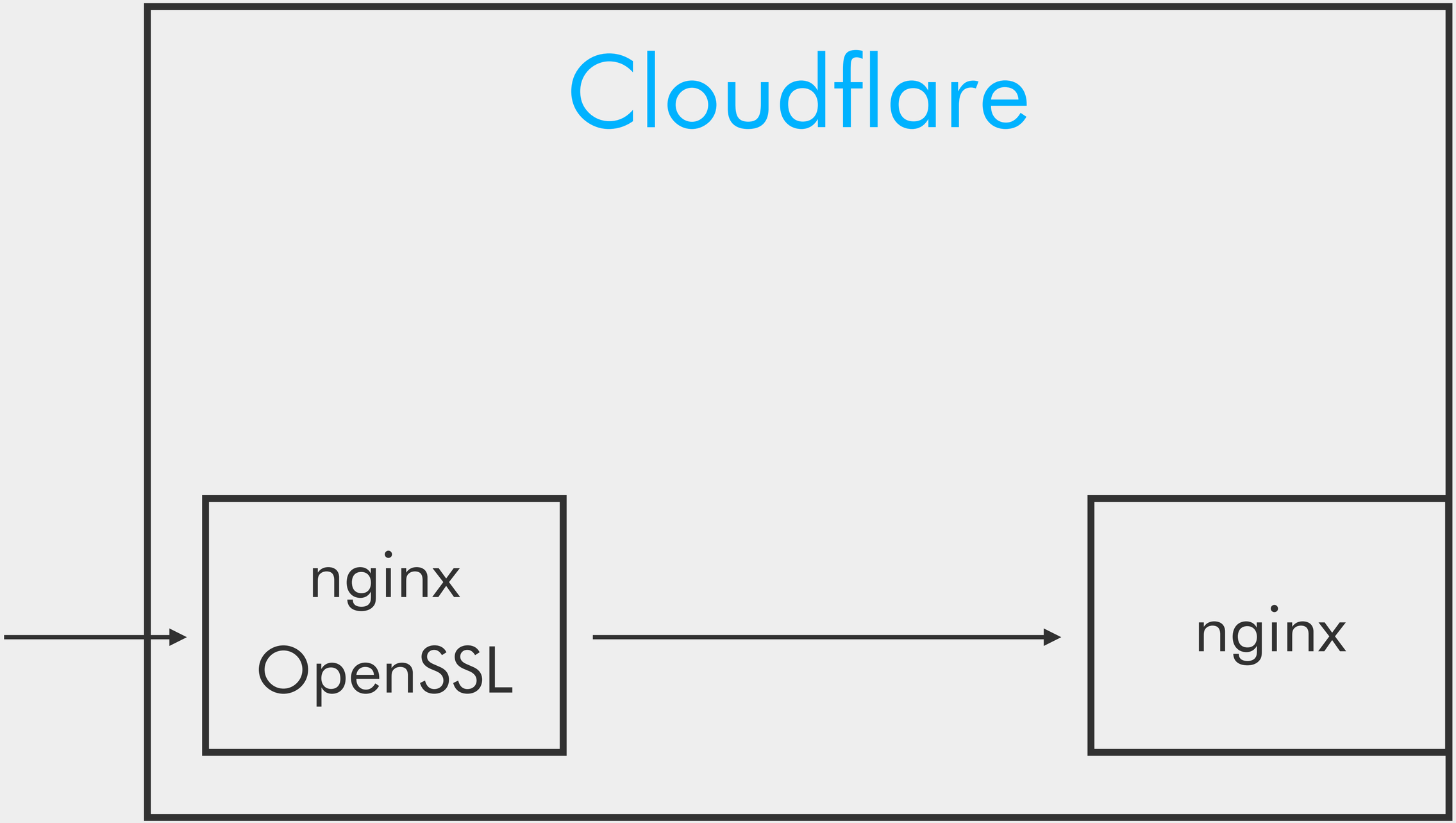
Only use assembly crypto

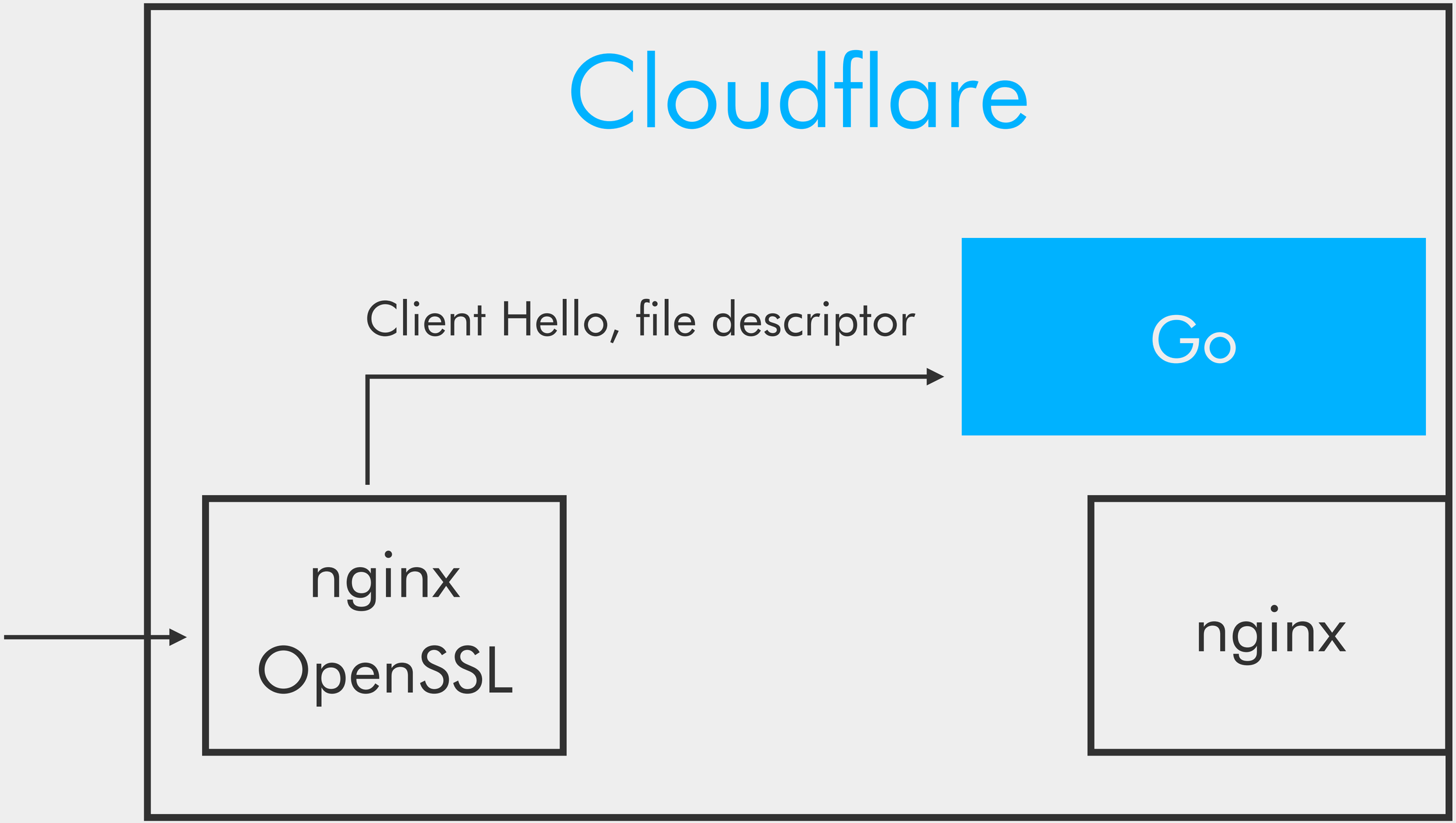
To get fast and constant time crypto, you currently need assembly implementations. On amd64:

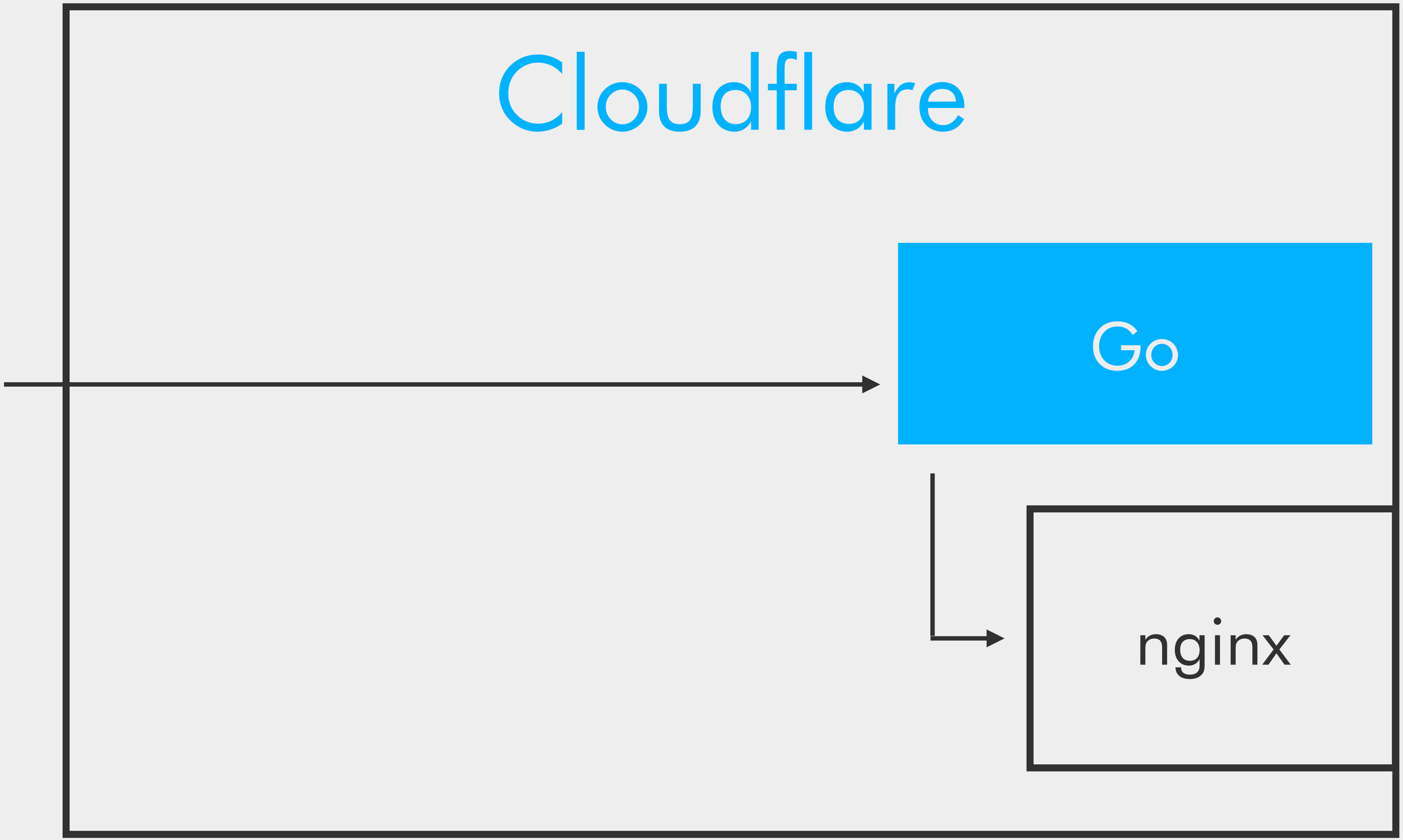
- crypto/elliptic.P256
  https://blog.cloudflare.com/go-crypto-bridging-the-performance-gap/
- golang.org/x/crypto/curve25519
- crypto/aes (especially fast with GCM)
- golang.org/x/crypto/chacha20poly1305

Only use assembly crypto

```
&tls.Config{
    // Causes servers to use Go's default ciphersuite preferences,
    // which are tuned to avoid attacks. Does nothing on clients.
    PreferServerCipherSuites: true,

    // Only use curves which have assembly implementations
    CurvePreferences: []tls.CurveID{
        tls.CurveP256,
        tls.X25519, // Go 1.8 only
    },
}
```


Cloudflare
nginx
OpenSSL
nginx


Cloudflare
Client Hello, file descriptor
Go
nginx
OpenSSL
nginx


Cloudflare
Go
nginx

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])

file := os.NewFile(uintptr(fds[0]), “")
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])

file := os.NewFile(uintptr(fds[0]), “")

fileconn, err := net.FileConn(file)
file.Close()
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])

file := os.NewFile(uintptr(fds[0]), “")

fileconn, err := net.FileConn(file)
file.Close()

tcp, ok := fileconn.(*net.TCPConn)
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])

file := os.NewFile(uintptr(fds[0]), “")

fileconn, err := net.FileConn(file)
file.Close()

tcp, ok := fileconn.(*net.TCPConn)

if tcp.RemoteAddr() == nil || tcp.RemoteAddr().(*net.TCPAddr) == nil { continue }
```

File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])

file := os.NewFile(uintptr(fds[0]), “")

fileconn, err := net.FileConn(file)
file.Close()

tcp, ok := fileconn.(*net.TCPConn)

if tcp.RemoteAddr() == nil || tcp.RemoteAddr().(*net.TCPAddr) == nil { continue }

return io.MultiReader(bytes.NewBuffer(hello), tcp)
```

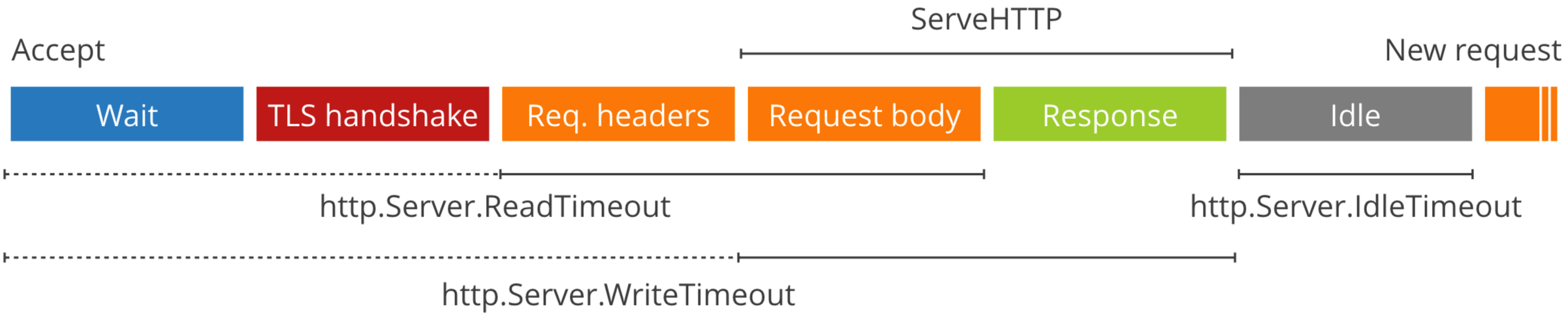
File descriptor passing

```
n, oobn, _, _, err := c.ReadMsgUnix(hello, oob)

cmsgs, err := syscall.ParseSocketControlMessage(oob[0:oobn])

fds, err := syscall.ParseUnixRights(&cmsgs[0])

file := os.NewFile(uintptr(fds[0]), "")

fileconn, err := net.FileConn(file)
file.Close()

tcp, ok := fileconn.(*net.TCPConn)

if tcp.RemoteAddr() == nil || tcp.RemoteAddr().(*net.TCPAddr) == nil { continue }

return io.MultiReader(bytes.NewBuffer(hello), tcp)
```


three days of my life

And now… net/http

Timeouts




```
http: Accept error: accept tcp [::]:80: accept: too many open files; retrying in 1s
```

Timeouts

```
srv := &http.Server{
    ReadTimeout:  5 * time.Second,
    WriteTimeout: 10 * time.Second,
    IdleTimeout:  120 * time.Second,
    TLSConfig:    tlsConfig,
    Handler:      serveMux,
}
log.Println(srv.ListenAndServeTLS("", ""))
```

Timeouts

```
func (c *conn) readRequest(ctx context.Context) (*response, error) {

    if d := c.server.ReadTimeout; d != 0 {
        c.rwc.SetReadDeadline(time.Now().Add(d))
    }
    if d := c.server.WriteTimeout; d != 0 {
        defer func() {
            c.rwc.SetWriteDeadline(time.Now().Add(d))
        }()
    }
```

Timeouts in Go 1.8

You really want Go 1.8, possibly 1.9.

- fixed ReadTimeout never resetting in H/2 #16450
- neutered WriteTimeout never resetting in H/2 #18437 (reintroduced and fixed in Go 1.9)
- introduced a proper IdleTimeout #14204

Still, timeouts are server-wide

So you can't change them based on path (maybe a streaming endpoint?) or authentication.

Should be addressed in 1.10. Come discuss the designs!

https://golang.org/issues/16100

What about TCP keep-alives?

- TCP-level feature, involving a “ping” on the wire
- Not a deadline for completion, just a requirement for the other party to answer the ping
- Useless against malicious DoS (“slowloris”)
- Enabled only by ListenAndServe[TLS], hard-coded at 3m

Keeping an eye on open connections

```
var idle = make(map[net.Conn]struct{}) // NOTE: protect with a sync.RWMutex

func connState(conn net.Conn, state http.ConnState) {
    switch state {
    case http.StateNew: // -> StateNew
    case http.StateActive:
        if _, ok := idle[conn]; ok { // StateIdle -> StateActive
            delete(idle, conn)
        } else { // StateNew -> StateActive }
    case http.StateIdle:
        idle[conn] = struct{}{} // StateActive -> StateIdle
    case http.StateHijacked: // StateActive -> StateHijacked
    case http.StateClosed:
        if _, ok := idle[conn]; ok { // StateIdle -> StateClosed
            delete(idle, conn)
        } else { // StateNew, StateActive -> StateClose }
```

http.Server and tls.Conn

```
func (c *conn) serve(ctx context.Context) {
    // [...]
    if tlsConn, ok := c.rwc.(*tls.Conn); ok {
```

http.Server type-asserts tls.Conn, which is why we can't use an external TLS package.

http.Server and tls.Conn

```
func (c *conn) serve(ctx context.Context) {
    // [...]
    if tlsConn, ok := c.rwc.(*tls.Conn); ok {
        if d := c.server.ReadTimeout; d != 0 {
            c.rwc.SetReadDeadline(time.Now().Add(d))
        }
        if d := c.server.WriteTimeout; d != 0 {
            c.rwc.SetWriteDeadline(time.Now().Add(d))
        }
```

First, it sets the timeouts.

http.Server and tls.Conn

```
func (c *conn) serve(ctx context.Context) {
    // [...]
    if tlsConn, ok := c.rwc.(*tls.Conn); ok {
        // [...]
        if err := tlsConn.Handshake(); err != nil {
            c.server.logf("http: TLS handshake error from %s: %v”,
                c.rwc.RemoteAddr(), err)
            return
        }
```

Then, it runs Handshake and logs the error if any.

… so, how do we drop a connection? Well…

http.Server and tls.Conn

```
func (c *conn) serve(ctx context.Context) {
    // [...]
    defer func() {
        if err := recover(); err != nil {
            c.server.logf("http: panic serving %v: %v", c.remoteAddr, err)
        }
        c.close()
        c.setState(c.rwc, StateClosed)
    }()
```

panic(nil), of course!

(Go 1.8 added ErrAbortHandler, thankfully.)

http.Server and tls.Conn

```
func (c *conn) serve(ctx context.Context) {
    // [...]
    if tlsConn, ok := c.rwc.(*tls.Conn); ok {
        // [...]
        if proto := c.tlsState.NegotiatedProtocol; validNPN(proto) {
            if fn := c.server.TLSNextProto[proto]; fn != nil {
                h := initNPNRequest{tlsConn, serverHandler{c.server}}
                fn(c.server, tlsConn, h)
            }
            return
        }
```

And finally it invokes the HTTP/2 handler!

When is HTTP/2 auto-enabled?

When:

- Server.TLSNextProto is nil (not empty map)
- Server.TLSConfig is set and ListenAndServeTLS is used
  or
  tls.Config.NextProtos includes "h2"

Server.TLSNextProto is automatically set (and used if the connection is HTTPS and the client offered H/2).

net/http in production

httputil.ReverseProxy

To connect the HTTP server with the Cloudflare backend, we used a modified httputil.ReverseProxy:

- Dial overridden to always open a connection to nginx
- A sizeable pool of connections to nginx
- Websockets
- H/2 Push
- 0-RTT with confirmation

Dial and pool to nginx

```
proxy := &httputil.ReverseProxy{
    Transport: &http.Transport{
        MaxIdleConns:        200,
        MaxIdleConnsPerHost: 200,
        IdleConnTimeout:     10 * time.Second,
        Dial: func(string, string) (net.Conn, error) {
            return net.DialTimeout(nginxNet, nginxAddr, timeout)
        },
    },
}
```

HTTP/2 Push

```
pu, ok := rw.(http.Pusher)
linkHeader := res.Header.Get("Link")
for i, ph := range derivePushHints(linkHeader) {
    po := &http.PushOptions{ Header: make(http.Header) }
    for _, k := range pushPromiseHeaders {
        v := res.Request.Header.Get(k)
        if v != "" { po.Header.Set(k, v) }
    }
    pu.Push(ph.uri, po)
}
```

0-RTT!

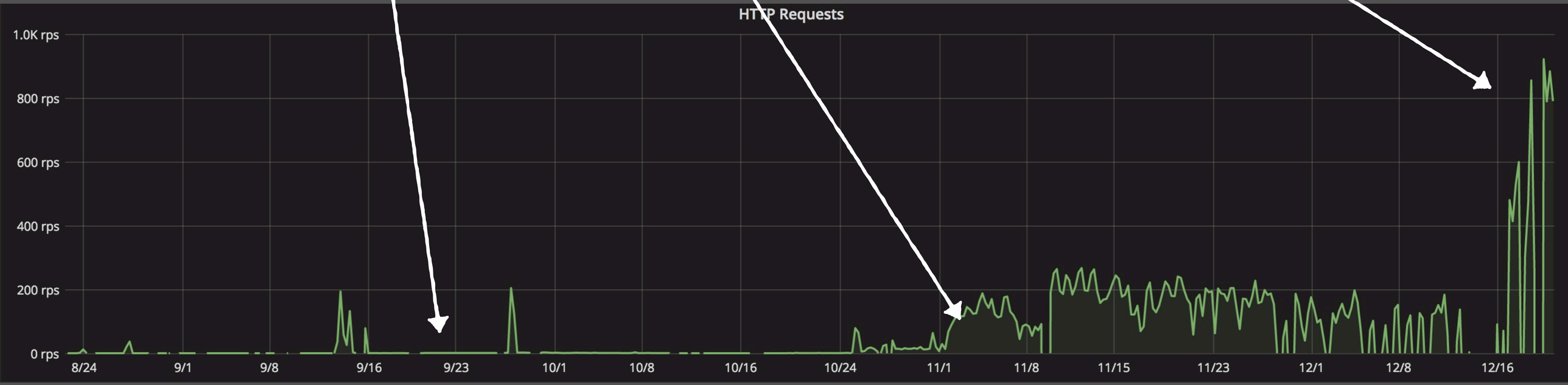
```
func director(req *http.Request) {
    tlsConn := req.Context().Value(http.TLSConnContextKey).(*tls.Conn)
    if !tlsConn.ConnectionState().HandshakeConfirmed {
        // Request came in 0-RTT data
        if req.Method == "GET" && req.URL.RawQuery == "" {
            req.Header.Set("CF-0RTT-Unique", connState.Fingerprint)
        } else {
            if err := tlsConn.ConfirmHandshake(); err != nil {
                panic(err)
            }
        }
    }
```

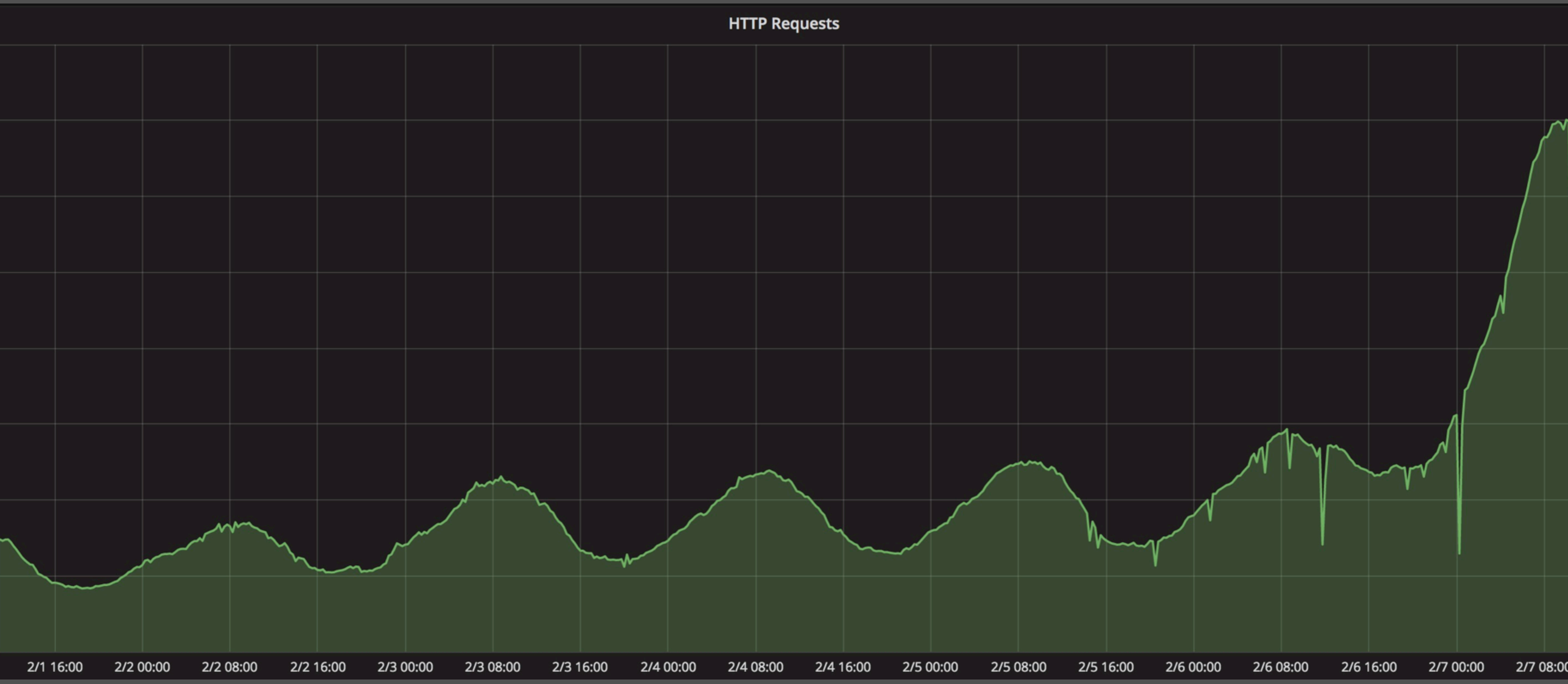
We didn’t break the Internet!

Cloudflare Launch

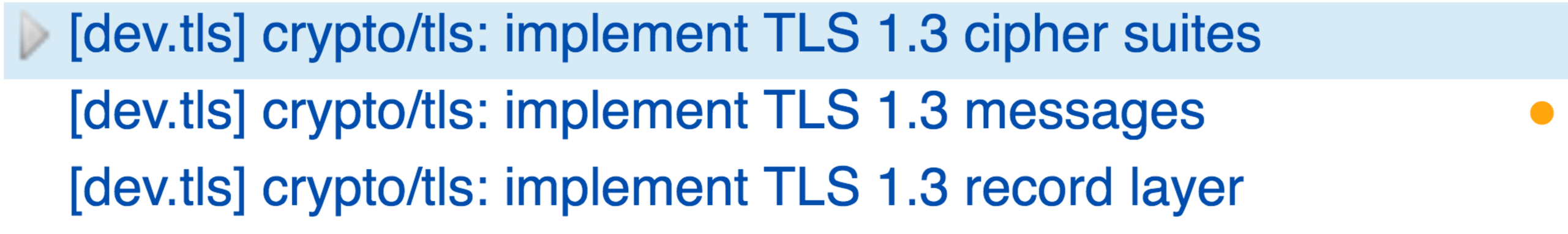
Firefox Nightly

Chrome Field Test




HTTP Requests
2/1 16:00
2/2 00:00
2/2 08:00
2/2 16:00
2/3 00:00
2/3 08:00
2/3 16:00
2/4 00:00
2/4 08:00
2/4 16:00
2/5 00:00
2/5 08:00
2/5 16:00
2/6 00:00
2/6 08:00
2/6 16:00
2/7 00:00
2/7 08:00

TLS 1.3 code: audited, upstreaming



https://go-review.googlesource.com/q/branch:+dev.tls

TLS 1.3: hung :-(

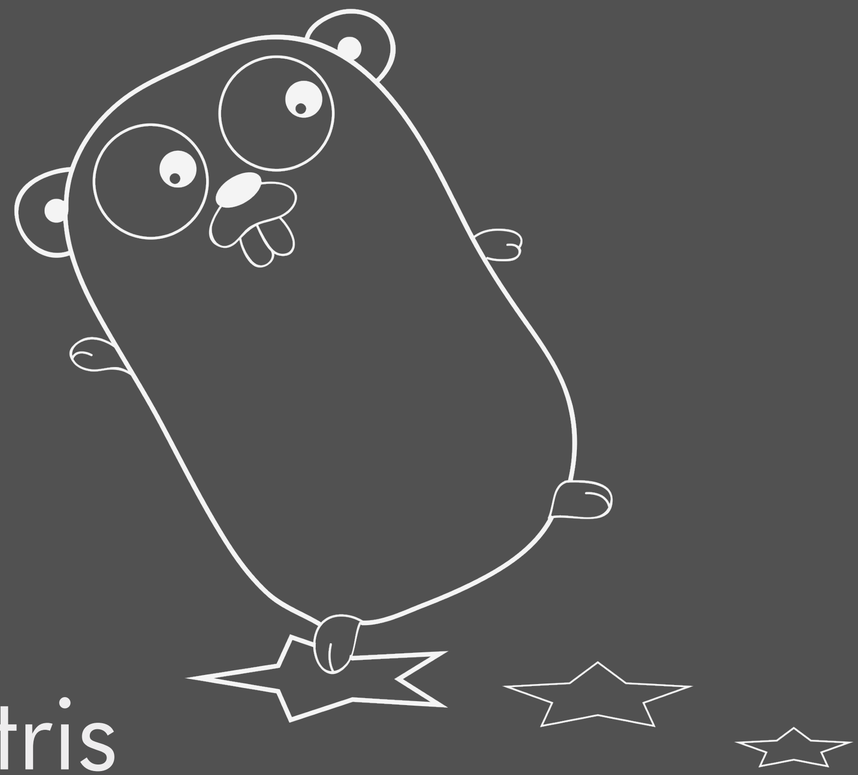
BlueCoat and other proxies hang up during TLS 1.3

Project Member Reported by jayhlee@google.com, Feb 21

Back to list

crypto/tls and net/http: awesome

Thank you!



https://golang.org/issues/9671

https://github.com/cloudflare/tls-tris

https://blog.cloudflare.com/tag/tls-1-3/

https://blog.gopheracademy.com/advent-2016/exposing-go-on-the-internet/

Filippo Valsorda
hi@filippo.io
@FiloSottile