Writing Networking Clients in Go

The Design and Implementation of the client from

Waldemar Quevedo / @wallyqs

GopherCon 2017

About me

- Waldemar Quevedo / @wallyqs
- Software Developer at Apcera
  - Development of the Apcera Platform
- NATS clients maintainer
- Using NATS based systems since 2012
- Go favorite feature: Performance!

About this talk

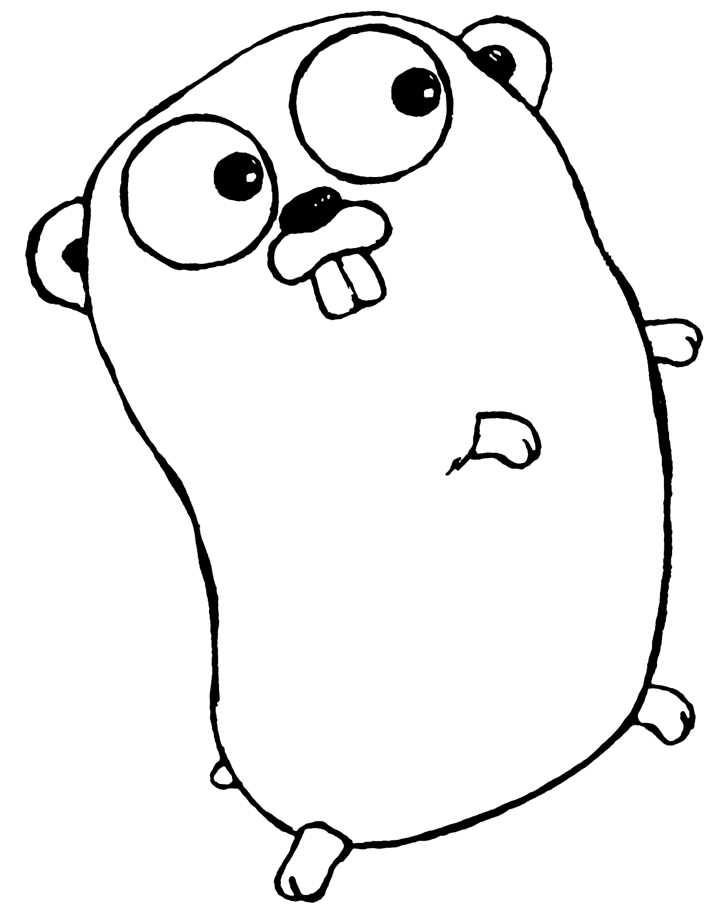
- Brief intro to the NATS project and its protocol
- Deep dive into how the NATS Go client is implemented

We'll cover...

- Techniques used and trade-offs in the Go client:
  - Backpressure of I/O and writes coalescing
  - Fast protocol parsing
  - Graceful reconnection on server failover
  - Usage & Non-usage of channels for communicating
  - Closures, callbacks & channels for events/custom logic

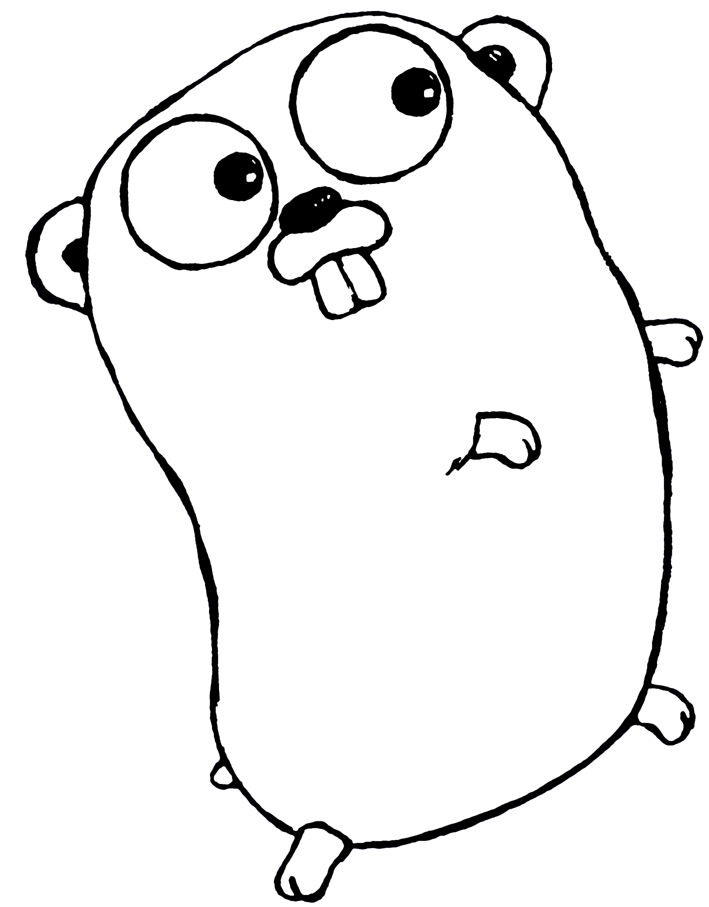
Agenda

- Intro to the NATS Project
- The NATS Protocol
- NATS Client Deep Dive
  - IO Engine
  - Connecting & Graceful Reconnection
  - Sync & Async Subscriptions
  - Request/Response
  - Performance



Agenda

- **Intro to the NATS Project**
- The NATS Protocol
- NATS Client Deep Dive
  - IO Engine
  - Connecting & Graceful Reconnection
  - Sync & Async Subscriptions
  - Request/Response
  - Performance



Brief Intro to the NATS Project

- High Performance Messaging System
  - **Open Source, MIT License**
- Created by Derek Collison
- Github: https://github.com/nats-io
- Website: http://nats.io/
- Development sponsored by APCERA

Used in production by thousands of users for…

- Building Microservices Control Planes
  - Internal communication among components
- Service Discovery
- Low Latency Request Response RPC
- Fire and Forget PubSub

Simple & Lightweight Design

- TCP/IP based
- Plain Text protocol with few commands
- Easy to use API
- Small binary of ~7MB
- Little config
- Just fire and forget, no built-in persistence
- *At-most-once* delivery guarantees

On Delivery Guarantees...

End-To-End Arguments In System Design (J.H. Saltzer, D.P. Reed and D.D. Clark)

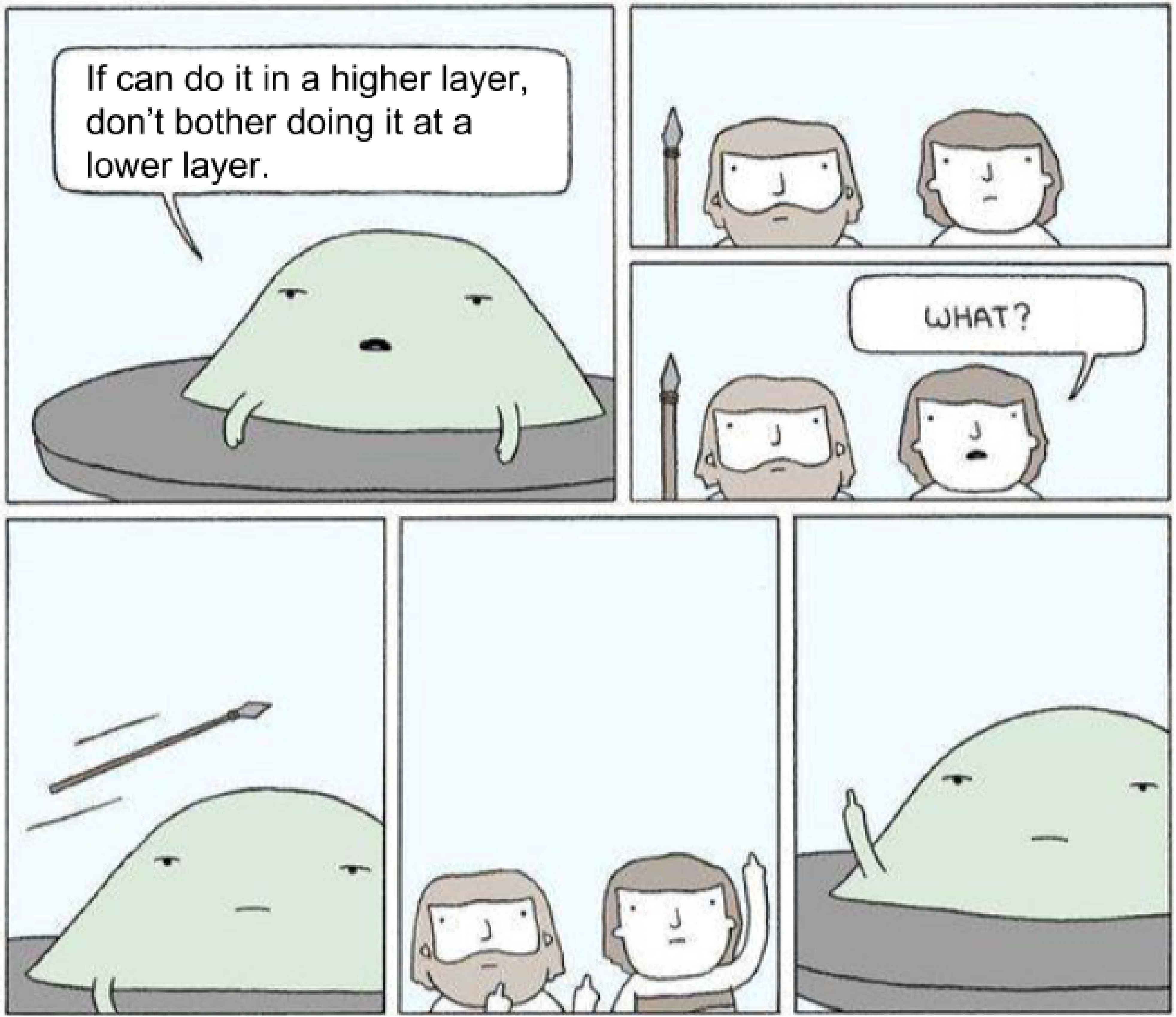
> *"...**a lower-level subsystem** that supports a distributed application **may be wasting its effort providing a function that must by nature be implemented at the application level anyway** can be applied to a variety of functions in addition to reliable data transmission."*

> *"Perhaps the oldest and most widely known form of the argument concerns acknowledgement of delivery. **A data communication network can easily return an acknowledgement to the sender for every message delivered to a recipient."***

End-To-End Arguments In System Design (J.H. Saltzer, D.P. Reed and D.D. Clark)

> *Although this acknowledgement may be useful within the network as a form of congestion control (...) it was never found to be very helpful to applications(...). The reason is that* ***knowing for sure that the message was delivered to the target host is not very important.***
>
> ***What the application wants to know is whether or not the target host acted on the message...***

BUT THEY COULD NOT UNDERSTAND ITS ALIEN LANGUAGE



End-To-End Arguments In System Design (J.H. Saltzer, D.P. Reed and D.D. Clark)

More info

https://en.wikipedia.org/wiki/End-to-end_principle

http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.txt

Recommended Talk!

"*Design Philosophy in Networked Systems*" by Justine Sherry (PWLConf'16)

https://www.youtube.com/watch?v=aR_UOSGEizE

Recommended Reading

"*Smart Endpoints, Dumb Pipes*" by Tyler Treat

http://bravenewgeek.com/smart-endpoints-dumb-pipes/

NATS Streaming

For *at-least-once* delivery check NATS Streaming (also OSS, MIT License).

It is a layer on top of NATS core which enhances it with message redelivery features and persistence.

https://github.com/nats-io/nats-streaming-server

NATS Streaming Server

NATS Streaming is an extremely performant, lightweight reliable streaming platform built on NATS.

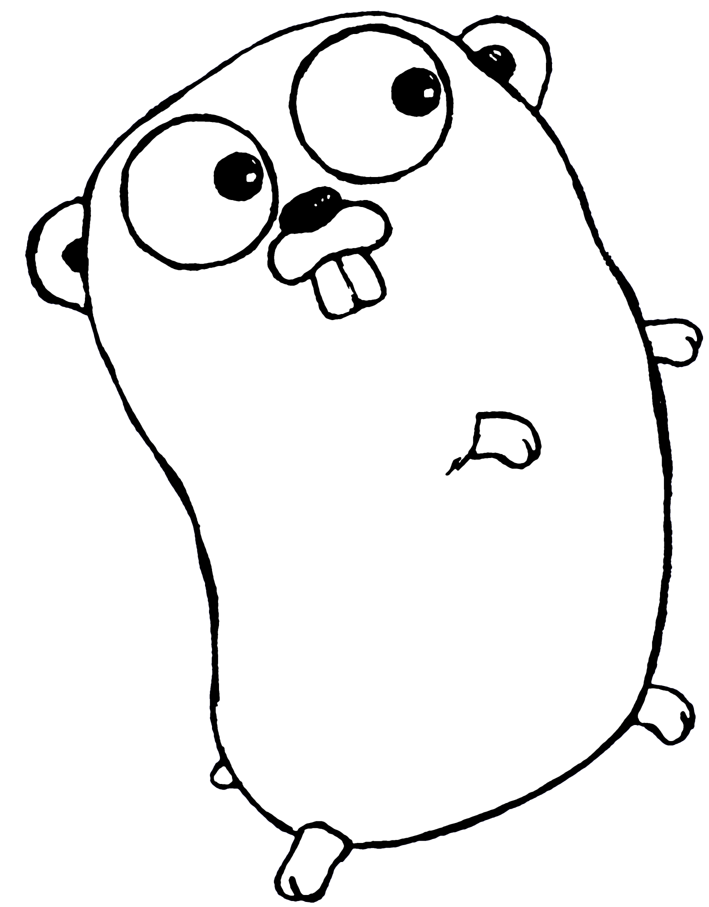
License MIT | go report A+ | build passing | coverage 94%

NATS Streaming provides the following high-level feature set.

- Log based.
- At-Least-Once Delivery model, giving reliable message delivery.
- Rate matched on a per subscription basis.
- Replay/Restart
- Last Value Semantics

Agenda

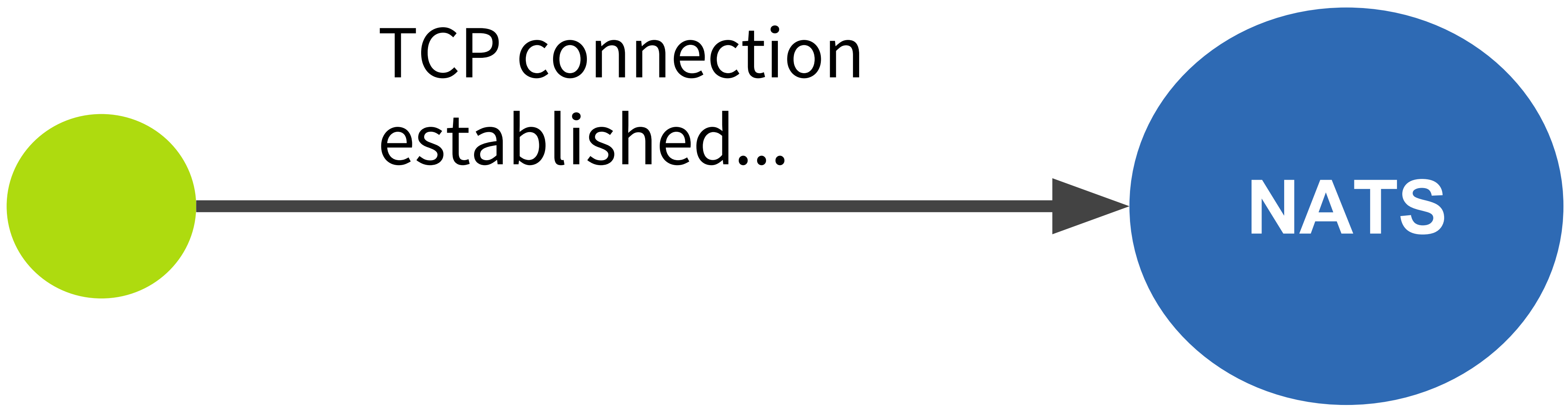
- Intro to the NATS Project
- **The NATS Protocol**
- NATS Client Deep Dive
  - IO Engine
  - Connecting & Graceful Reconnection
  - Sync & Async Subscriptions
  - Request/Response
  - Performance



The NATS Protocol

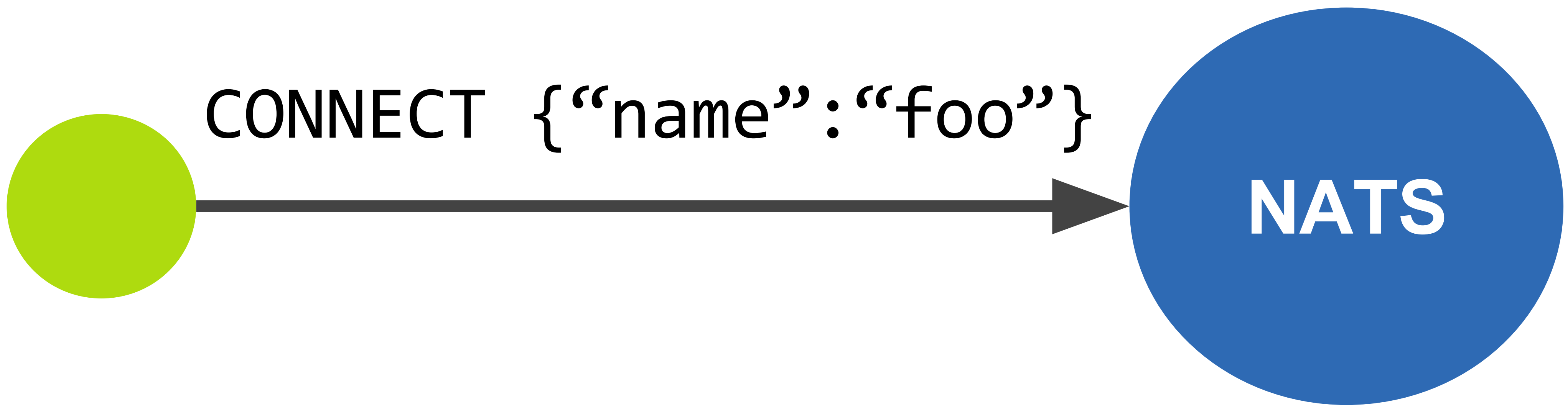
```
Client -> Server

| PUB | SUB | UNSUB | CONNECT |

Client <- Server

| INFO | MSG | -ERR | +OK |

Client <-> Server

| PING | PONG |
```

As soon as client connects…



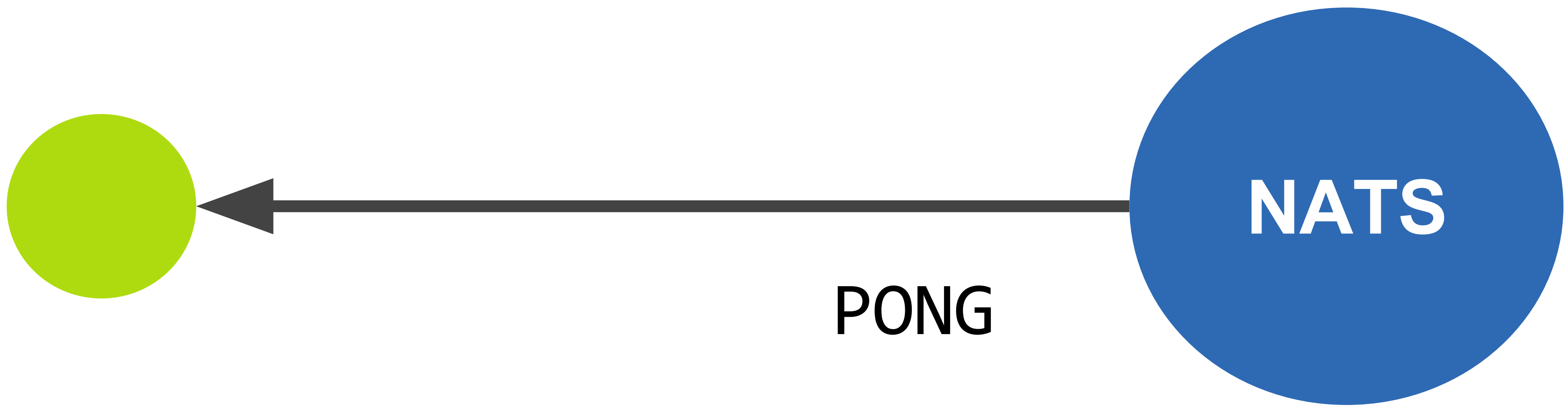
…it receives an `INFO` string from server

Clients can customize connection via `CONNECT`

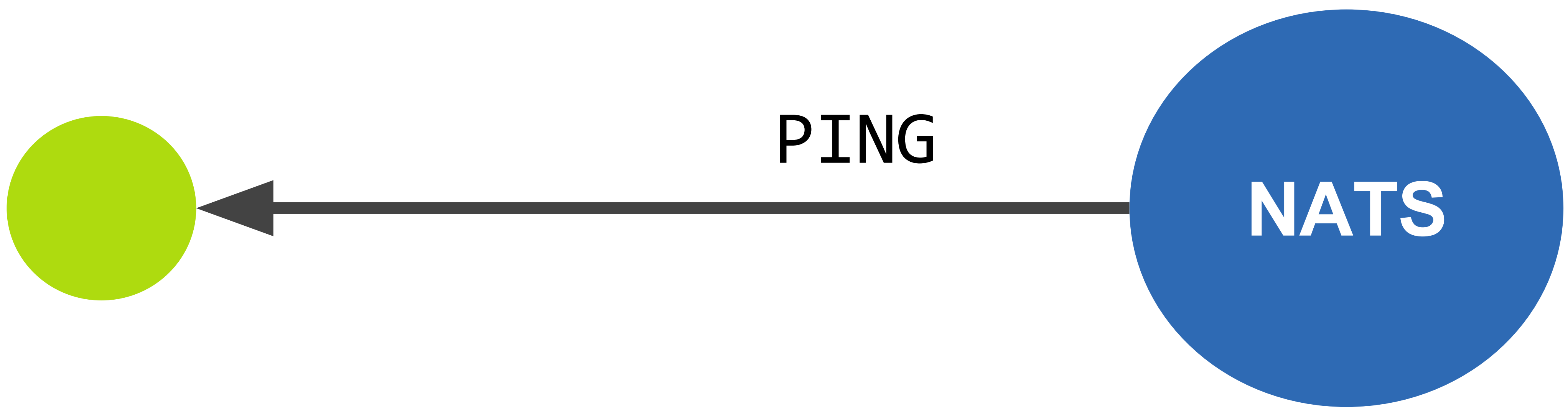


We can send `PING` to the server…

…to which we will get a PONG back.



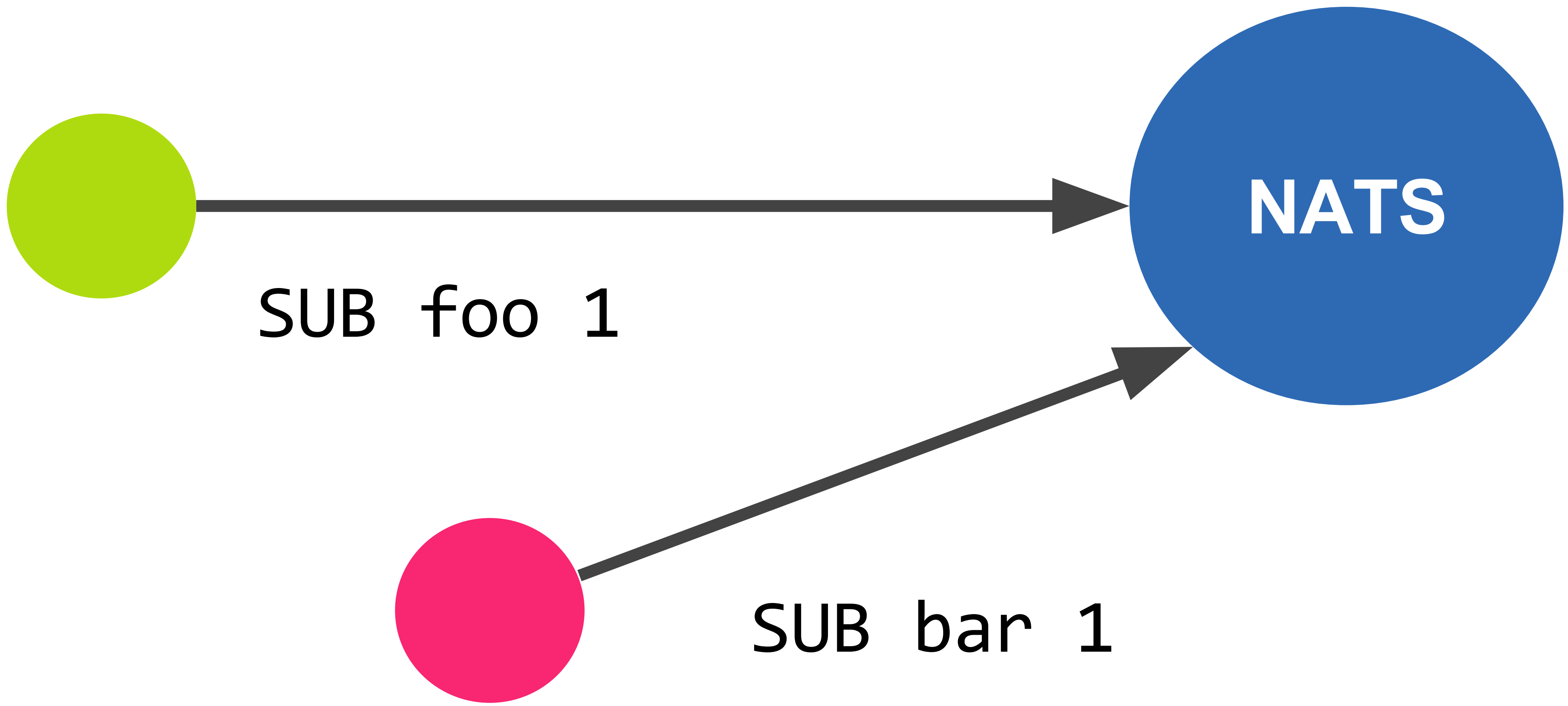
Server periodically sends `PING` to clients too…



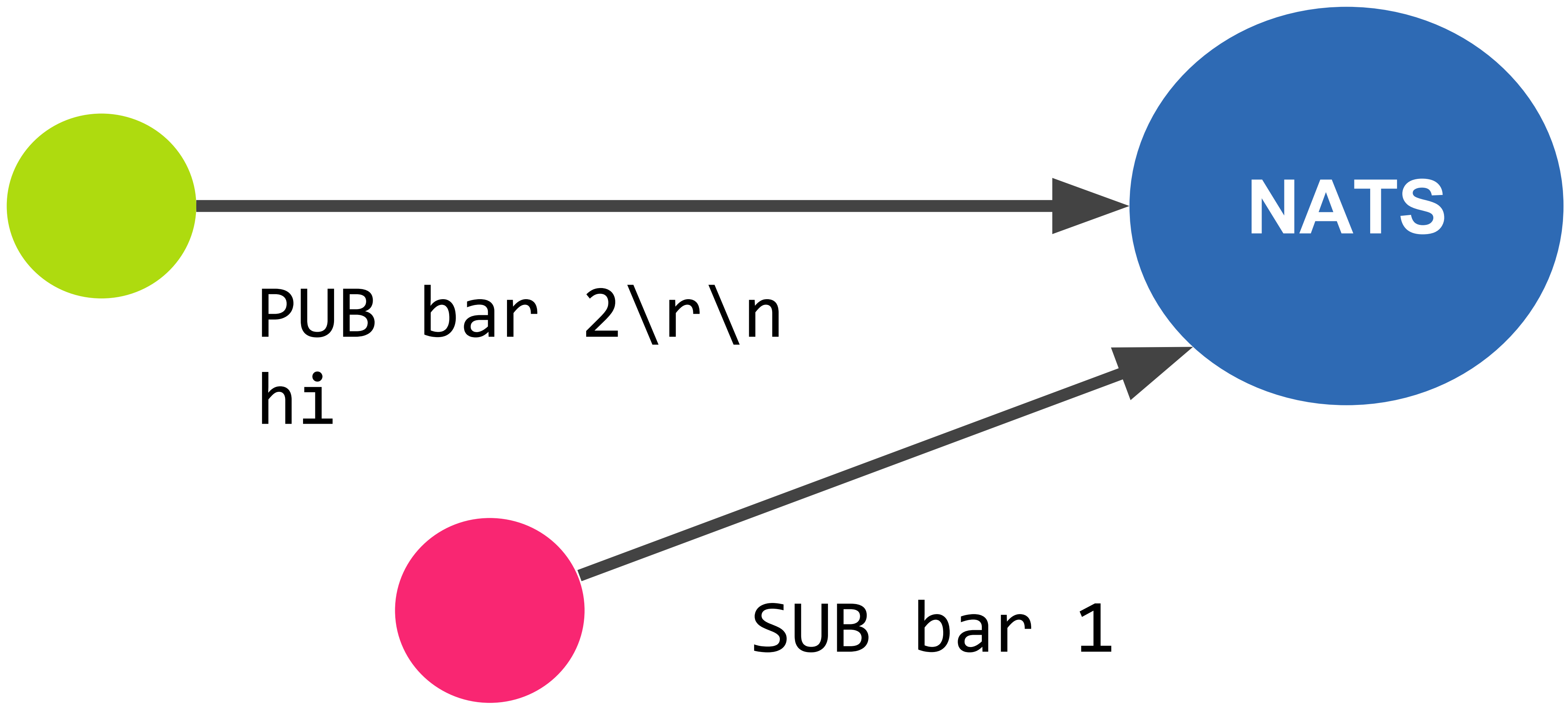
…to which clients have to `PONG` back.

Otherwise, server terminates connection eventually.

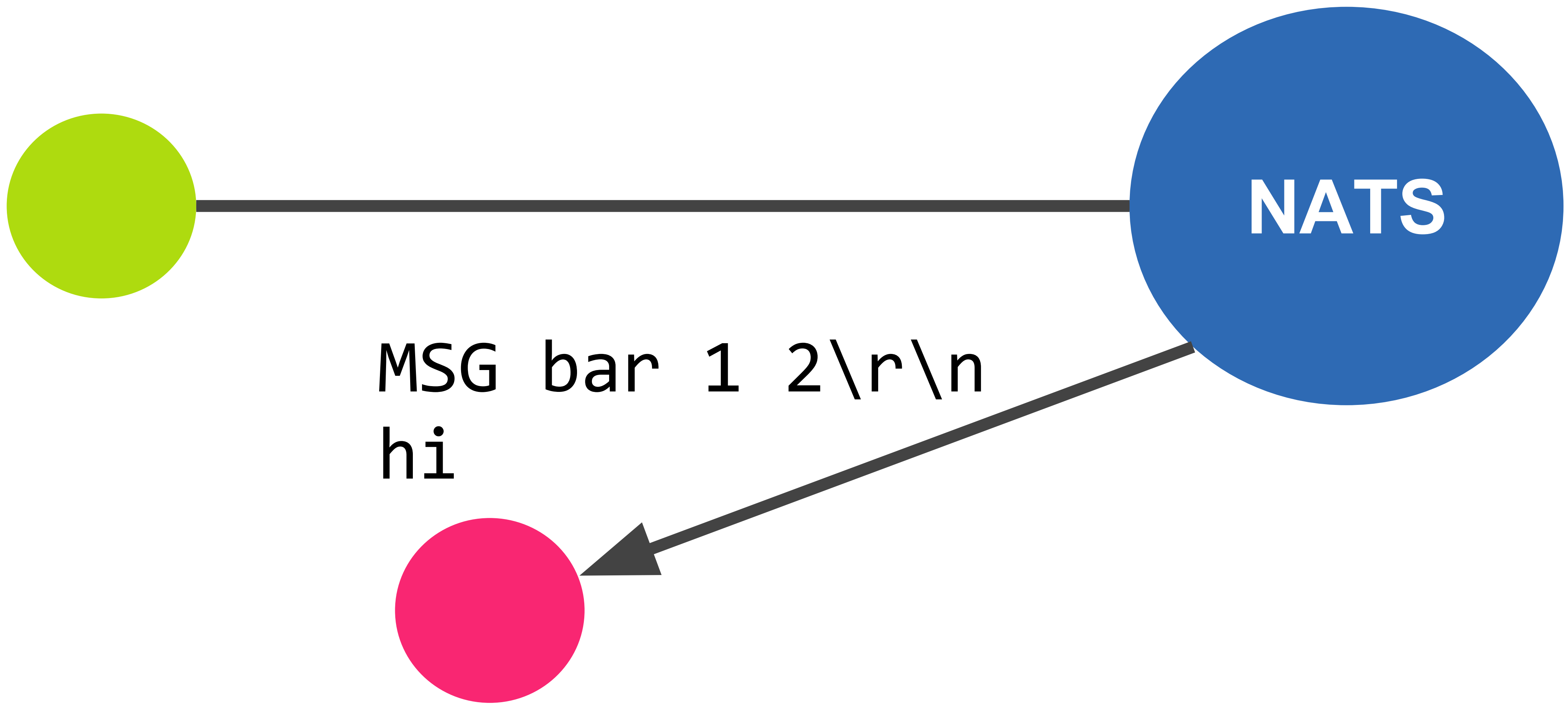
To announce interest in a subject, we use SUB…



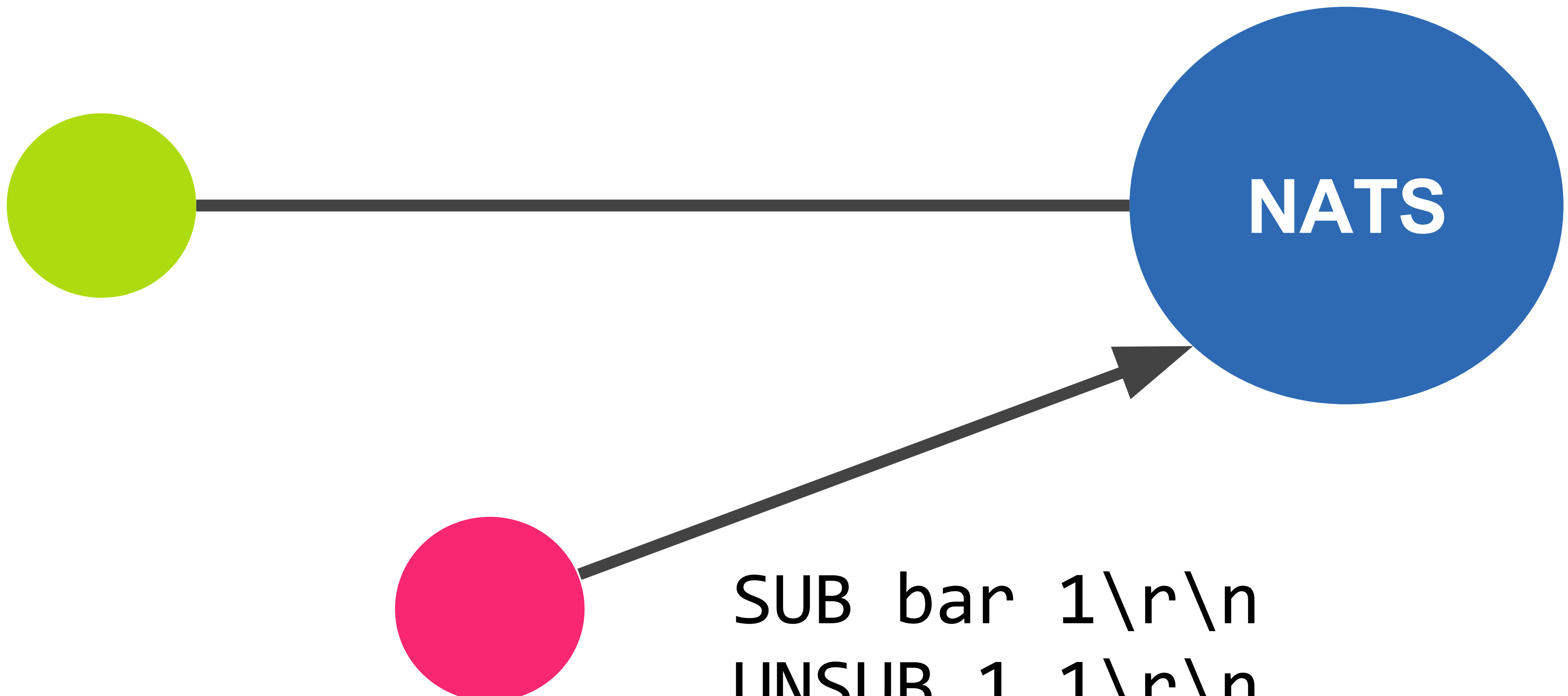
…and PUB to send a command to connected clients.



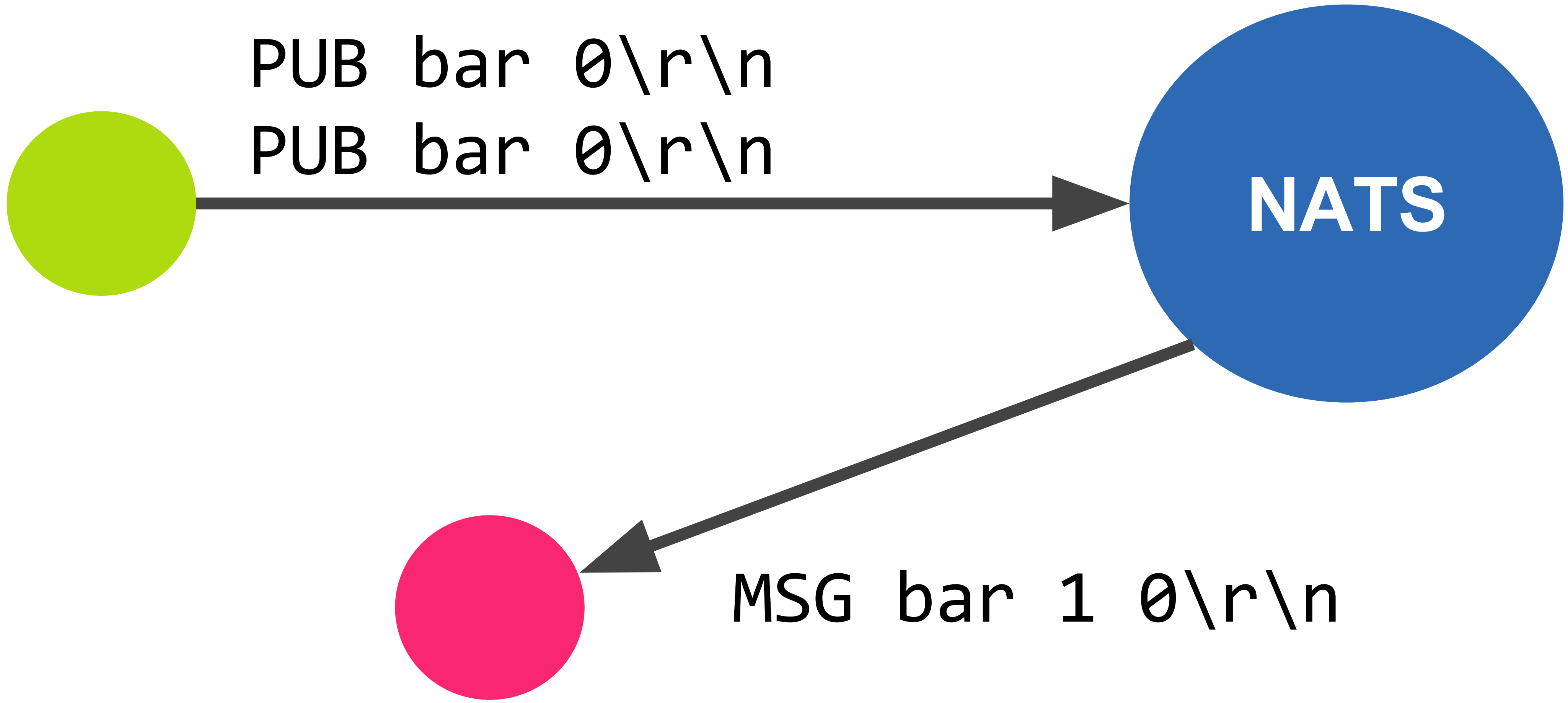
Then, a `MSG` will be delivered to interested clients…



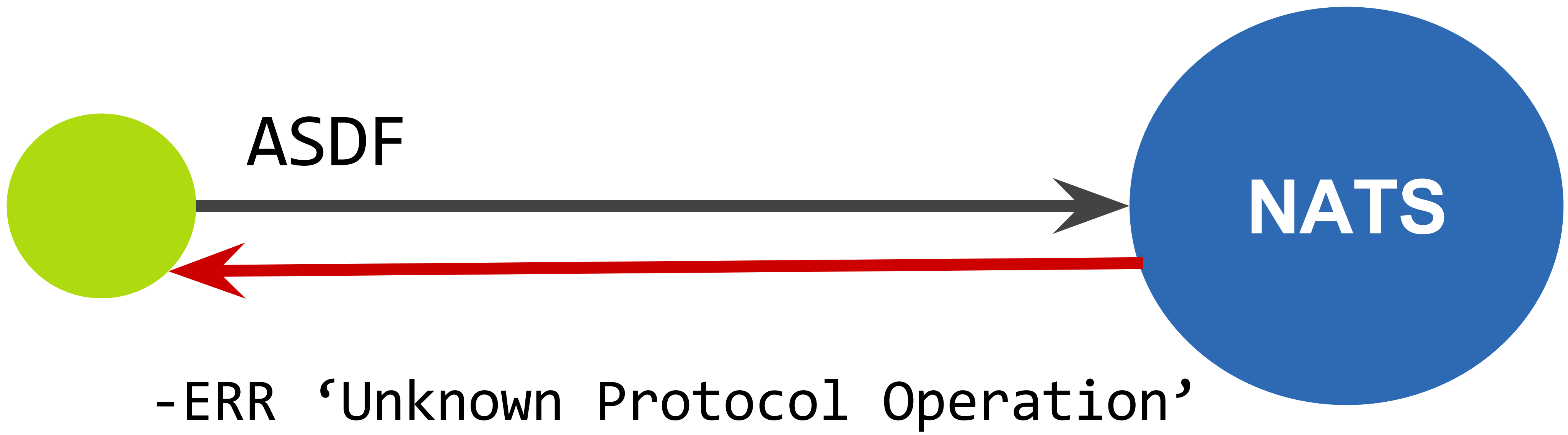
Interest in subject can be limited via `UNSUB`...



…then if many messages are sent, we only get one.

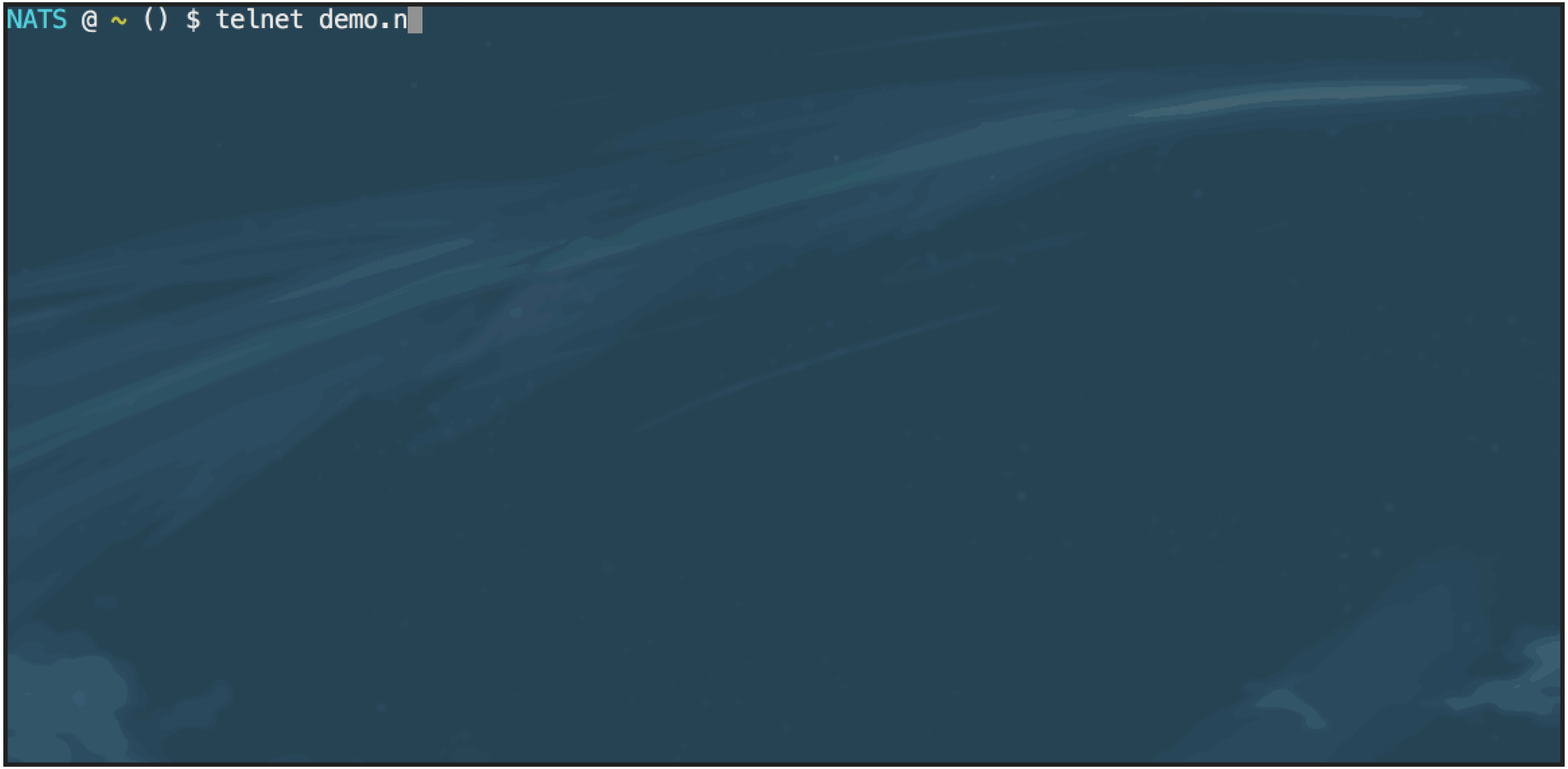


At any time, the server can send us an error…



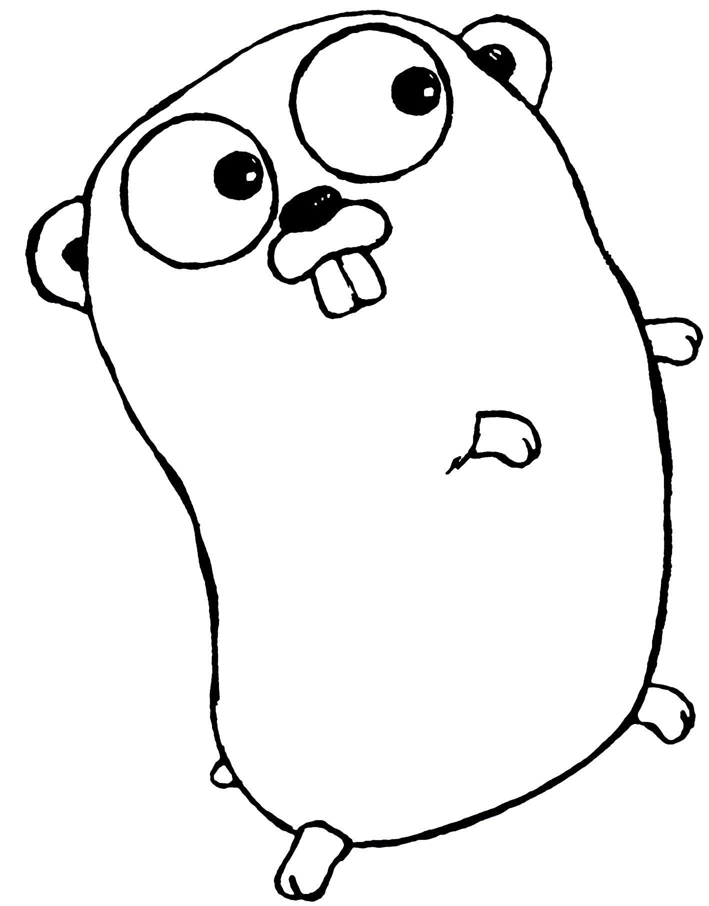
…and disconnect us after sending the error.

Example Telnet session



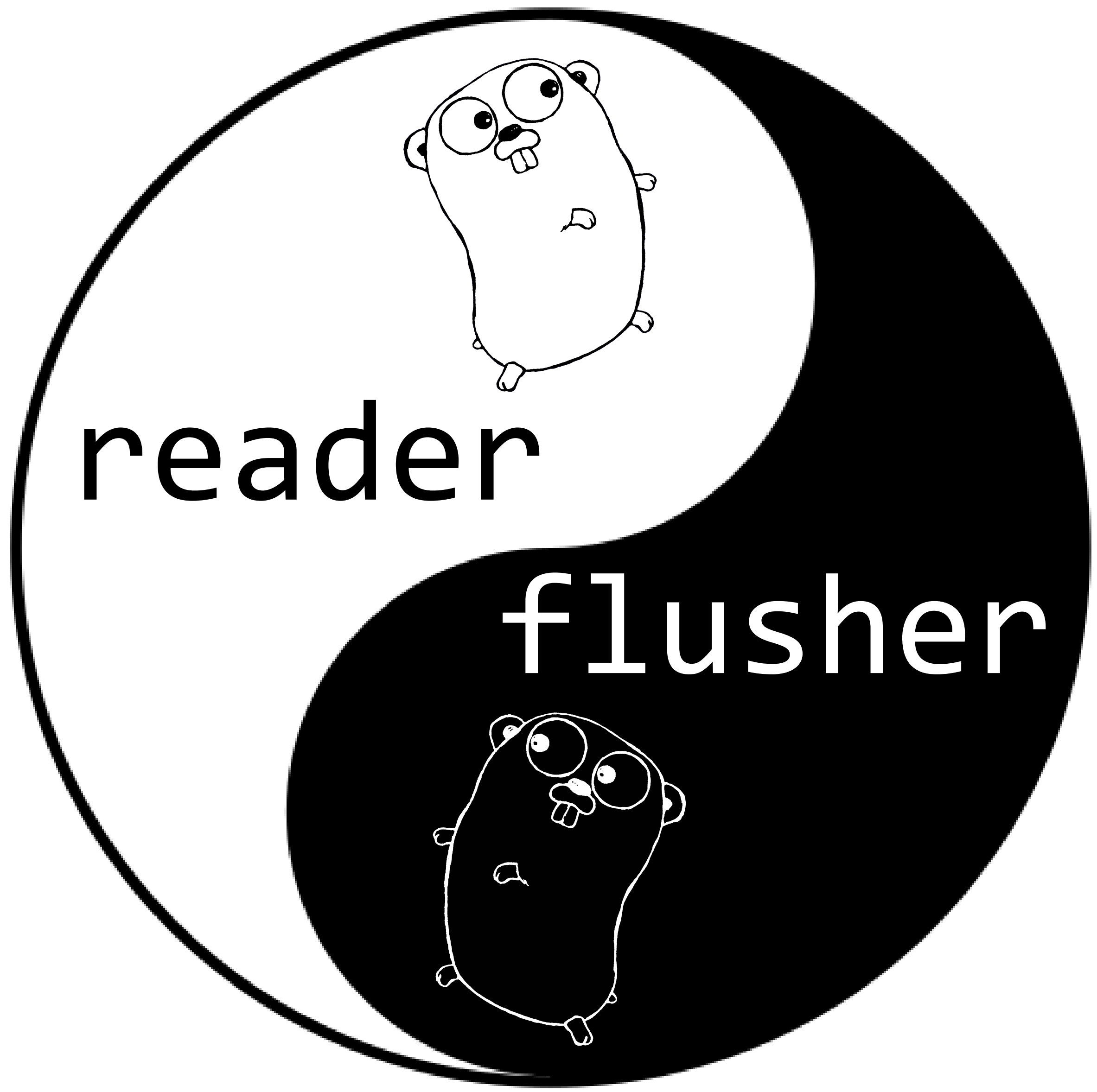
Agenda

- Intro to the NATS Project
- The NATS Protocol
- NATS Client Deep Dive
  - **IO Engine**
    - Optimized Flushing
    - Fast Protocol Parsing Engine
  - Connecting & Graceful Reconnection
  - Sync & Async Subscriptions
  - Request / Response APIs
  - Performance



NATS Client Engine

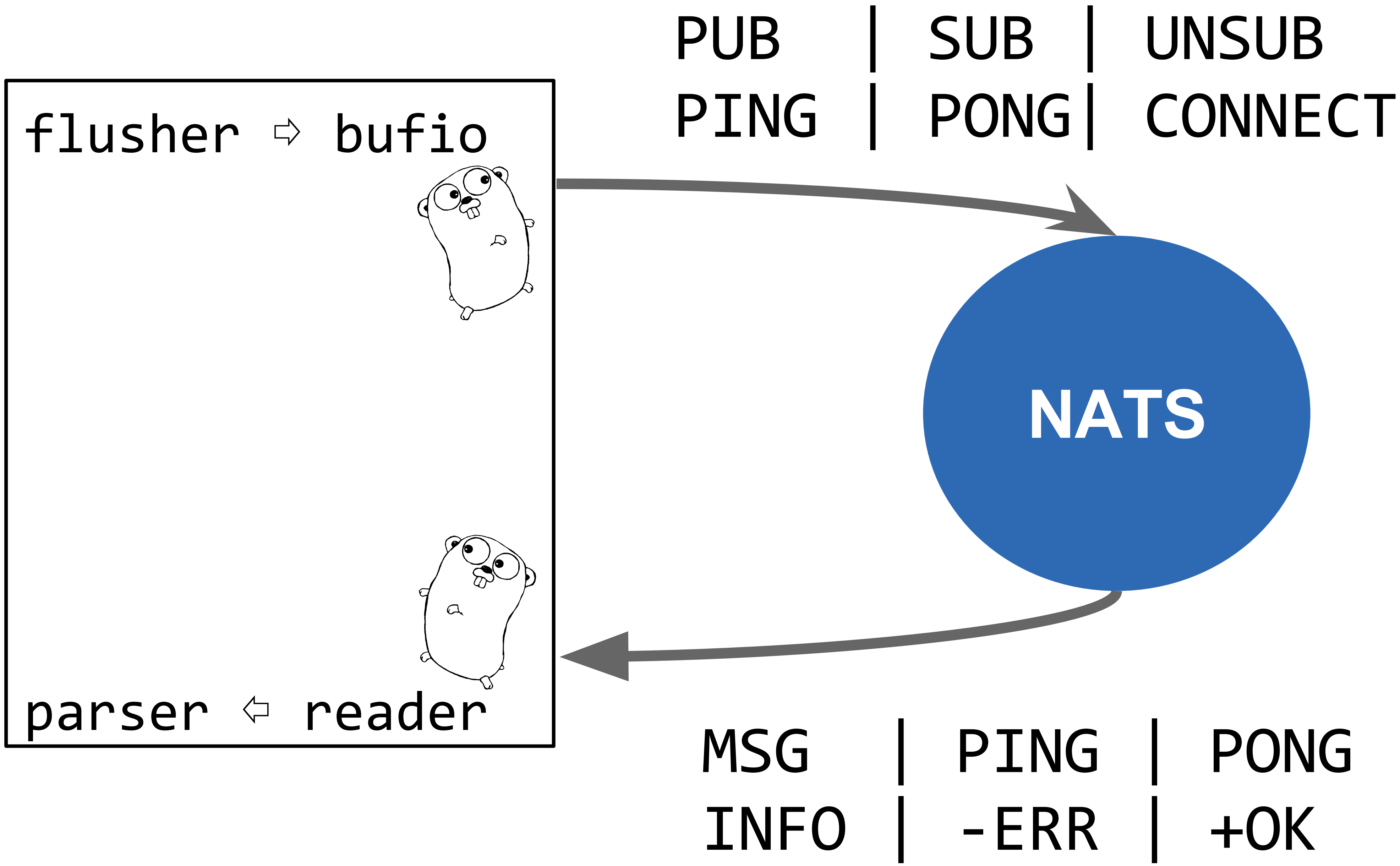
The client is built around a couple of goroutines which cooperate to read / write messages as fast as possible asynchronously.



NATS Client Engine

```
// spinUpGoRoutines will launch the goroutines responsible for
// reading and writing to the socket. This will be launched via a
// go routine itself to release any locks that may be held.
// We also use a WaitGroup to make sure we only start them on a
// reconnect when the previous ones have exited.
func (nc *Conn) spinUpGoRoutines() {
	// Make sure everything has exited.
	nc.waitForExits(nc.wg)

	// Create a new waitGroup instance for this run.
	nc.wg = &sync.WaitGroup{}
	// We will wait on both.
	nc.wg.Add(2)

	// Spin up the readLoop and the socket flusher.
	go nc.readLoop(nc.wg)
	go nc.flusher(nc.wg)
```

NATS Client Engine



Flushing mechanism

Implemented via a buffered channel which is used to signal a flush of any pending data in the *bufio.Writer*.

```
nc.fch = make(chan struct{}, 1024)
```

```
// flusher is a separate goroutines that will process flush requests for the write
// bufio. This allows coalescing of writes to the underlying socket.
func (nc *Conn) flusher(wg *sync.WaitGroup) {
	defer wg.Done()
	// ...
	for {
		if _, ok := <-fch; !ok {
			return
		}
		// ...
		if bw.Buffered() > 0 {
			bw.Flush()
		}
```

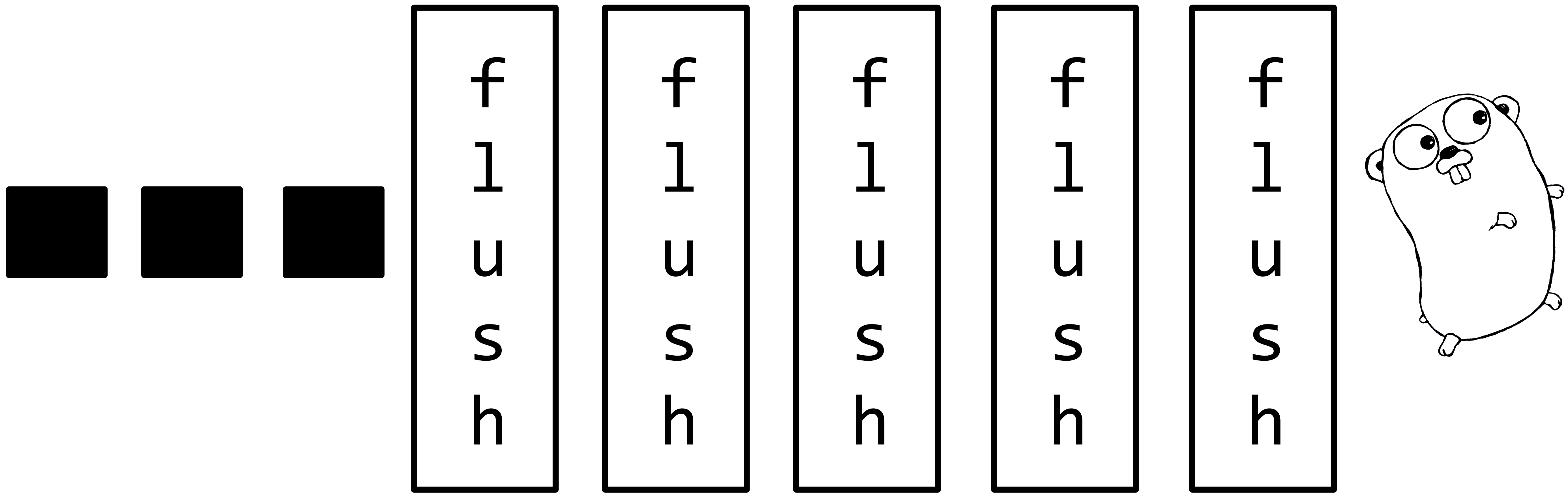
Flushing mechanism

On each command being sent to the server, we prepare a flush of any pending data in case there are no pending flushes already.

```go
func (nc *Conn) publish(subj, reply string, data []byte) error {
	// ...
	_, err := nc.bw.Write(msgh)
	if err != nil {
		return err
	}
	// Schedule pending flush in case there are none.
	if len(nc.fch) == 0 {
		select {
		case nc.fch <- struct{}{}:
		default:
		}
	}
}
```

Flushing mechanism

This helps adding backpressure into the client and do writes coalescing to spend less time writing into the socket.



Reader loop

The client uses a *zero allocation parser* under a read loop based on the stack

```go
// readLoop() will sit on the socket reading and processing the
// protocol from the server. It will dispatch appropriately based
// on the op type.
func (nc *Conn) readLoop(wg *sync.WaitGroup) {
	// ...

	// Stack based buffer
	b := make([]byte, 32768)

	for {
		// ...
		n, err := conn.Read(b)
		if err != nil {
			nc.processOpErr(err)
			break
		}

		if err := nc.parse(b[:n]); err != nil {
			nc.processOpErr(err)
			break
		}
		// ...
```

Fast Protocol Parsing Engine

```
func (nc *Conn) parse(buf []byte) error {
	var i int
	var b byte

	for i = 0; i < len(buf); i++ {
		b = buf[i]

		switch nc.ps.state {
		case OP_START:
			switch b {
			case 'M', 'm':
				nc.ps.state = OP_M
			case 'P', 'p':
				nc.ps.state = OP_P
			case 'I', 'i':
				nc.ps.state = OP_I
		// ...
		case MSG_PAYLOAD:
			if nc.ps.msgBuf != nil {
				if len(nc.ps.msgBuf) >= nc.ps.ma.size {
					nc.processMsg(nc.ps.msgBuf)
					nc.ps.argBuf, nc.ps.msgBuf, nc.ps.state = nil, nil,
		// ...
```

Fast Protocol Parsing Engine

```
const (
        OP_START = iota
        OP_PLUS                 // +
        OP_PLUS_O               // +O
        OP_PLUS_OK              // +OK
        OP_MINUS                // -
        OP_MINUS_E              // -E
        OP_MINUS_ER             // -ER
        OP_MINUS_ERR            // -ERR
        OP_MINUS_ERR_SPC        // -ERR
        MINUS_ERR_ARG           // -ERR   '
        OP_M                    // M
        OP_MS                   // MS
        OP_MSG                  // MSG
        OP_MSG_SPC              // MSG
        MSG_ARG                 // MSG
        MSG_PAYLOAD             // MSG foo bar 1 \r\n
        MSG_END                 // MSG foo bar 1 \r\n
        OP_P                    // P
        OP_PI                   // PI
        OP_PIN                  // PIN
        OP_PING                 // PING
        OP_PO                   // PO
        OP_PON                  // PON
        OP_PONG                 // PONG
        OP_I                    // I
        OP_IN                   // IN
        OP_INF                  // INF
        OP_INFO                 // INFO
        OP_INFO_SPC             // INFO
        INFO_ARG                // INFO
)
```

Commands are sent asynchronously!

Publishing/Subscribing actions do not block, instead we rely on the client internal engine to eventually flush and send to the server.

```
nc1, _ := nats.Connect("nats://demo.nats.io:4222")
nc2, _ := nats.Connect("nats://demo.nats.io:4222")

nc1.Subscribe("foo", func(m *nats.Msg){
        log.Printf("[Received] %+v", m)
})

nc2.Publish("foo",  []byte("Hello World"))
```

In example above, there is no guarantee that `foo` subscription will be receiving the message (may not have flushed yet).

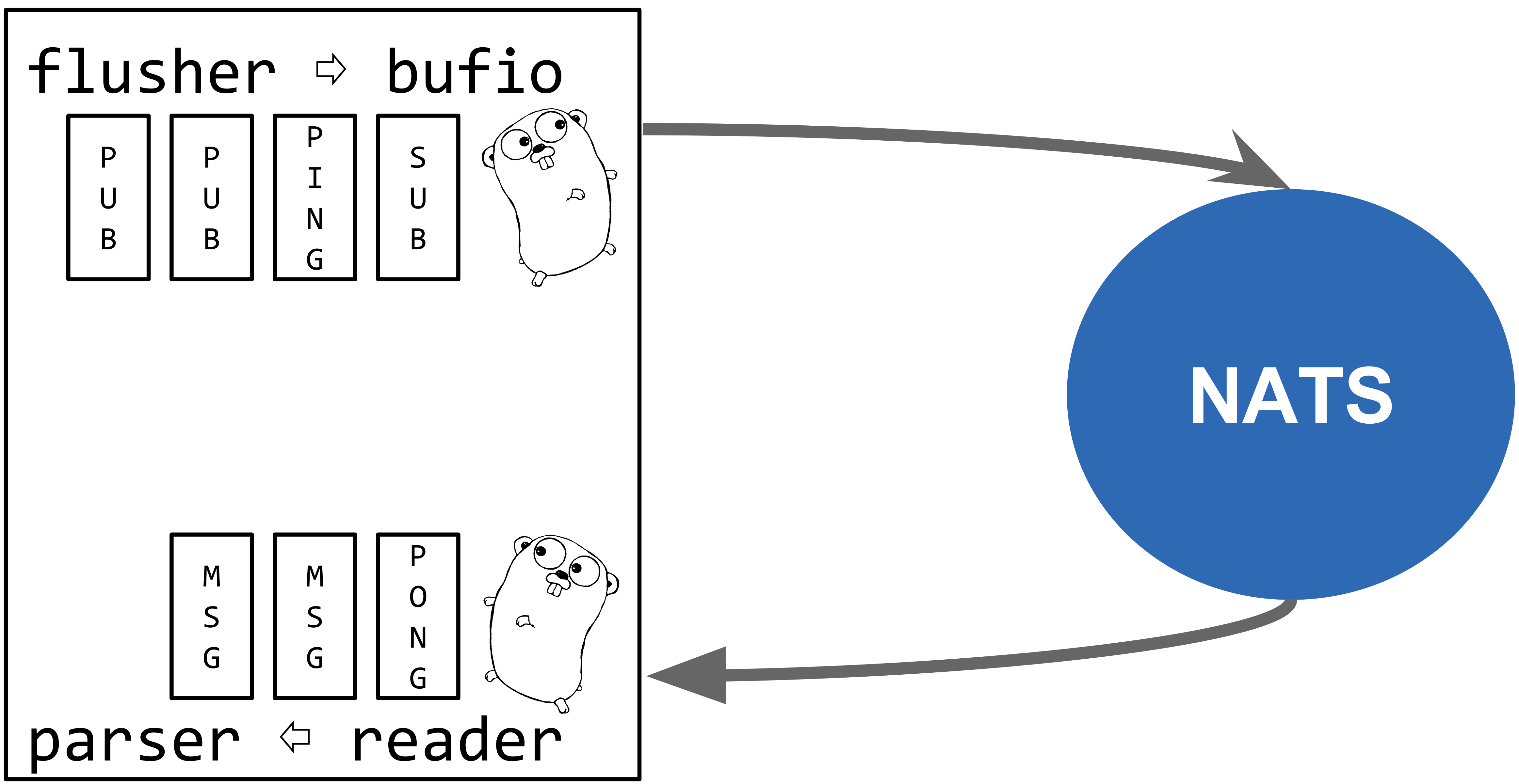
FAQ

If commands are sent asynchronously, how to ensure that a command sent has been processed by the server?

🤔

Ensure ordering via `PING` / `PONG`

Ordering is guaranteed per connection, thus sending a `PING` first immediately after `SUB` would guarantee us receiving `PONG` before any other matching `MSG` on the subscription.



NATS Flush

The client provides a `Flush` API which both flushes the *bufio.Writer* and does the server roundtrip for us.

```
nc1, _ := nats.Connect("nats://demo.nats.io:4222")
nc2, _ := nats.Connect("nats://demo.nats.io:4222")

nc1.Subscribe("foo", func(m *nats.Msg){
        log.Printf("[Received] %+v", m)
})

// Sends PING, flushes buffer, and blocks until receiving PONG reply.
nc1.Flush()

// Would be received by subscription
nc2.Publish("foo",  []byte("Hello World"))
```

NATS Flush

We create an unbuffered channel per each pending pong reply, and append into an array of channels which represent the pending replies.

```
        nc.pongs = make([]chan struct{}, 0, 8)
```

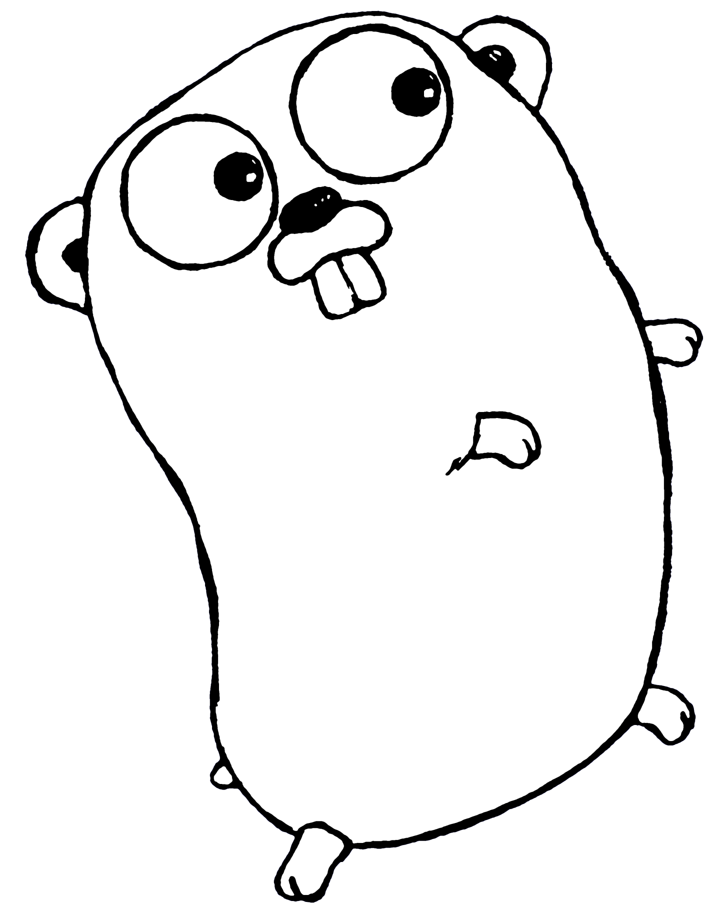
```
func (nc *Conn) FlushTimeout(timeout time.Duration) (err error) {
        // ...
        ch := make(chan struct{})
        nc.pongs = append(nc.pongs, ch)
        nc.bw.WriteString("PING\r\n")
        nc.bw.Flush()
        // ...
        select {
        case <-ch:
                close(ch)
        case <-time.After(timeout):
                return ErrTimeout
        }
```

NATS Flush

Then when we receive the `PONG` back, we signal the client that we have received it so that it stops blocking.

```go
// processPong is used to process responses to the client's ping
// messages. We use pings for the flush mechanism as well.
func (nc *Conn) processPong() {
	var ch chan struct{}

	nc.mu.Lock()
	if len(nc.pongs) > 0 {
		ch = nc.pongs[0]
		nc.pongs = nc.pongs[1:]
	}
	nc.pout = 0
	nc.mu.Unlock()
	if ch != nil {
		// Signal that PONG was received.
		ch <- struct{}{}
	}
}
```

- Intro to the NATS Project
- The NATS Protocol
- NATS Client Deep Dive
  - IO Engine
  - **Connecting & Graceful Reconnection**
  - Synchronous & Asynchronous Subscriptions
  - Request/Response
  - Performance



Connecting Options

For connecting to NATS, we use the functional options pattern to be able to extend the customizations.

```go
// Connect will attempt to connect to the NATS system.
// The url can contain username/password semantics. e.g. nats://derek:pass@localhost:4222
// Comma separated arrays are also supported, e.g. urlA, urlB.
// Options start with the defaults but can be overridden.
func Connect(url string, options ...Option) (*Conn, error) {
	opts := DefaultOptions
	opts.Servers = processUrlString(url)
	for _, opt := range options {
		if err := opt(&opts); err != nil {
			return nil, err
		}
	}
	return opts.Connect()
}
```

https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis

Connecting Options

For example, to customize set of credentials sent on `CONNECT`…

`CONNECT {"user":"secret", "pass":"password"}\r\n`

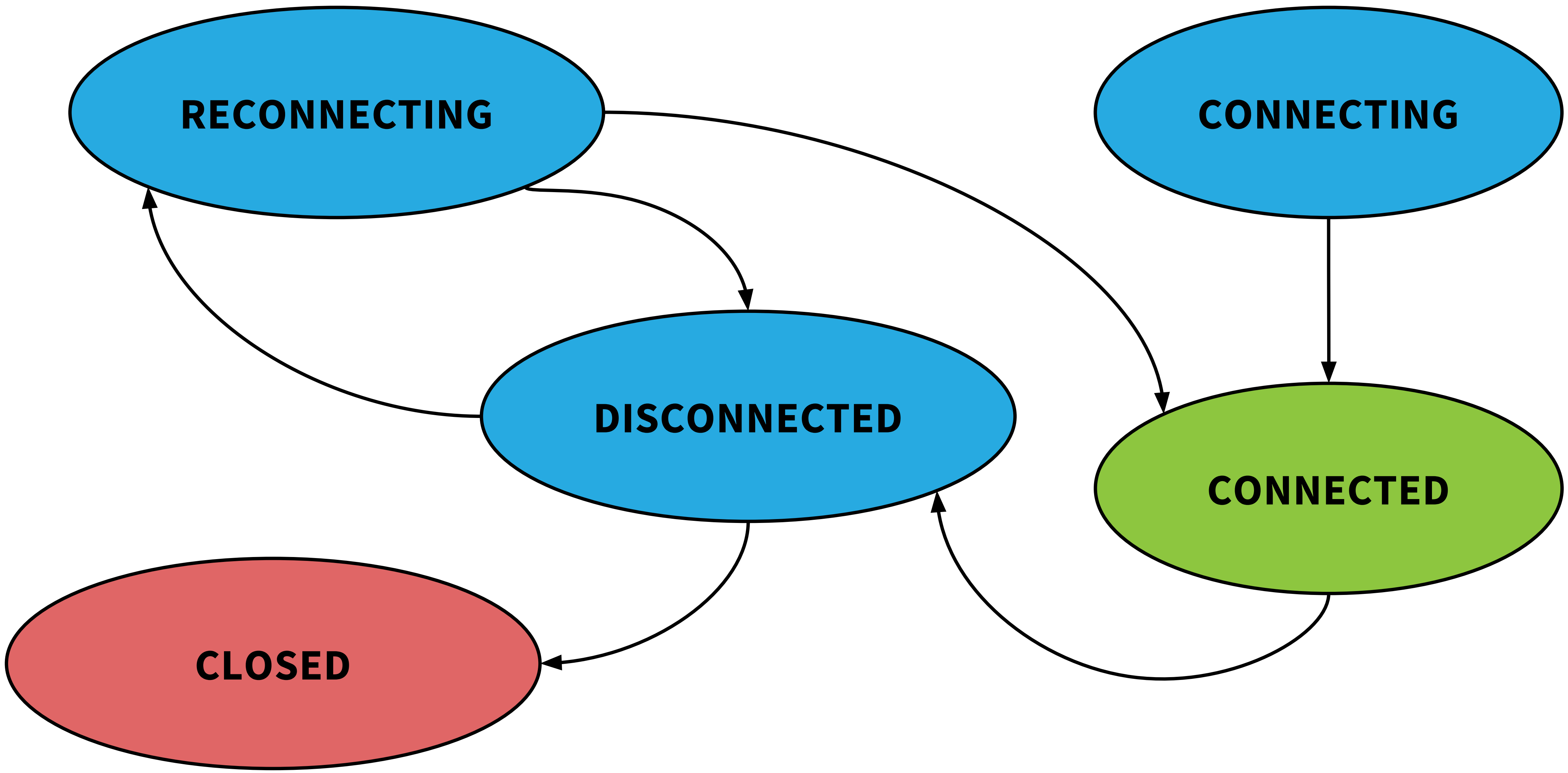
```go
// UserInfo is an Option to set the username and password to
// use when not included directly in the URLs.
func UserInfo(user, password string) Option {
	return func(o *Options) error {
		o.User = user
		o.Password = password
		return nil
	}
}

// Usage:
//
nats.Connect("nats://127.0.0.1:4222", nats.UserInfo("secret", "password"))
```

Connecting Options

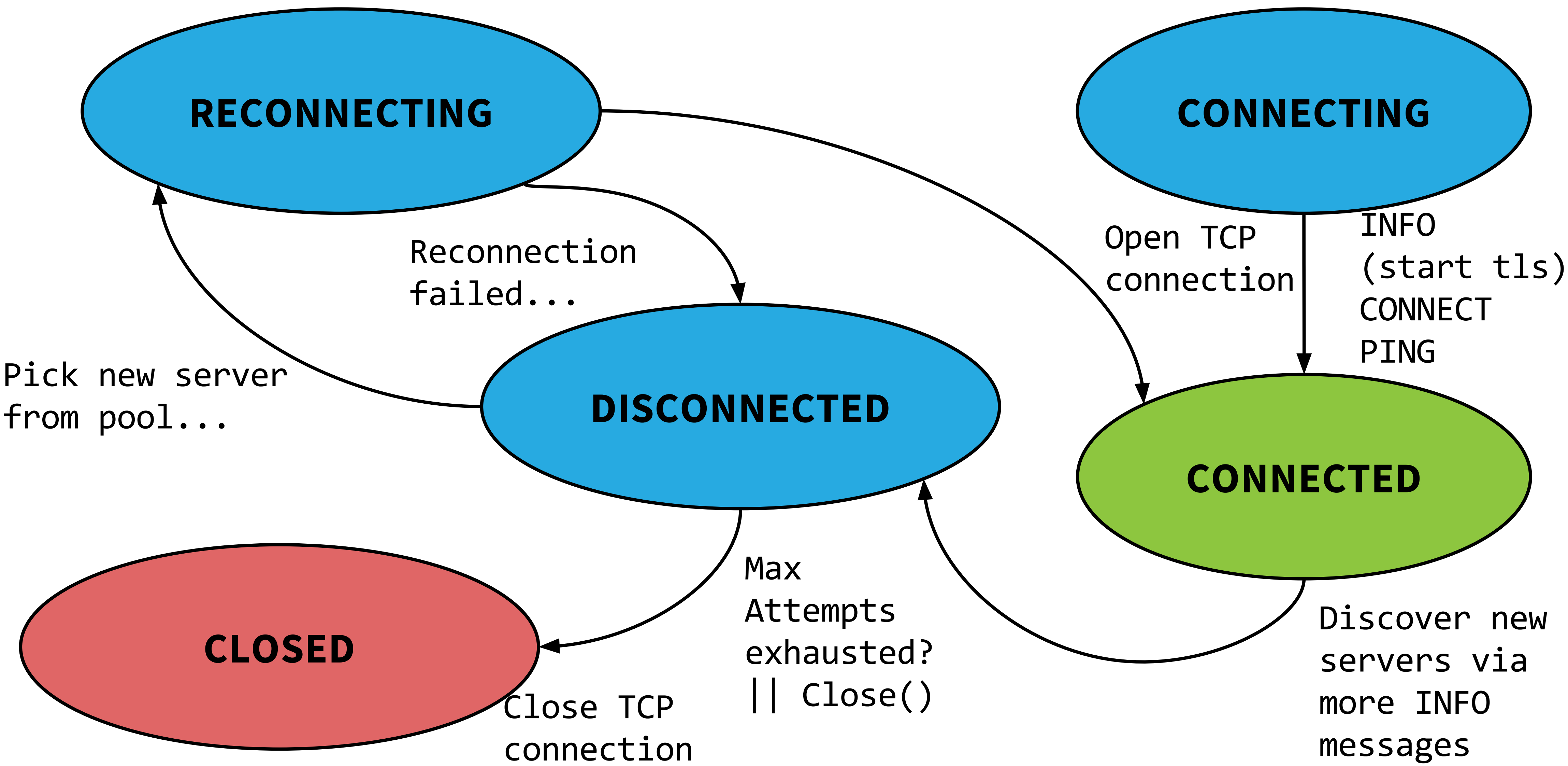
Or to customize TLS:

```
nc, err = nats.Connect(secureURL,
        nats.RootCAs("./ca.pem"),
        nats.ClientCert("./client-cert.pem", "./client-key.pem"))
if err != nil {
        t.Fatalf("Failed to create (TLS) connection: %v", err)
}
defer nc.Close()
```

Connecting & Reconnecting flow



Connecting & Reconnecting flow



Reconnecting Options

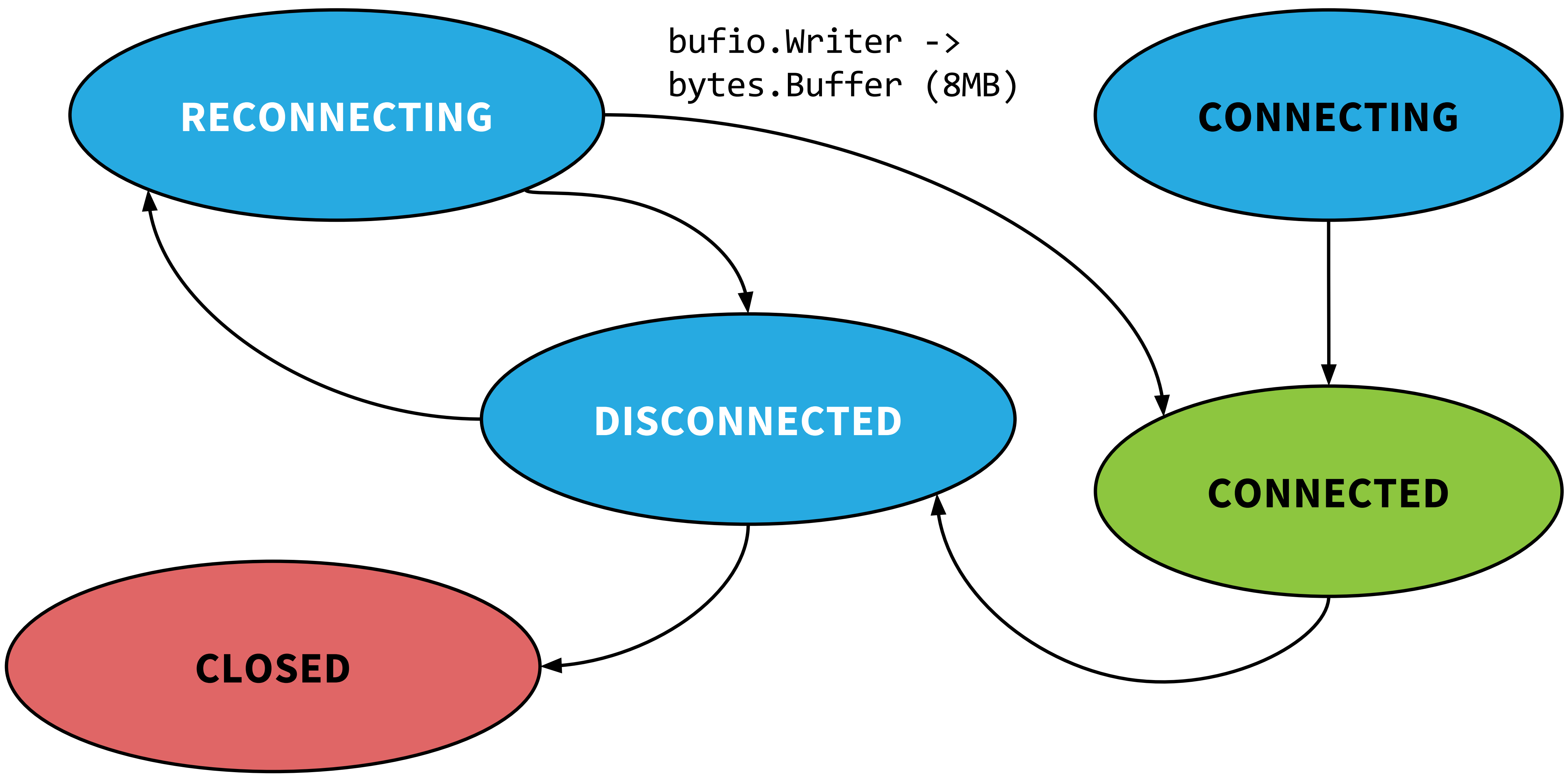
By default, the client has reconnecting enabled but can be customized too.

This means that if there is ever an issue with the server that we were connected to, then it will try to connect to another one in the pool.

```go
// ReconnectWait is an Option to set the wait time between reconnect attempts.
func ReconnectWait(t time.Duration) Option {
	return func(o *Options) error {
		o.ReconnectWait = t
		return nil
	}
}

// MaxReconnects is an Option to set the maximum number of reconnect attempts.
func MaxReconnects(max int) Option {
	return func(o *Options) error {
		o.MaxReconnect = max
		return nil
	}
}
```

Reconnection logic

While the client is in the `RECONNECTING` state, the underlying `bufio.Writer` is replaced with a `bytes.Buffer` instead of the connection.



Reconnection logic

```
	if nc.Opts.AllowReconnect && nc.status == CONNECTED {
		// Set our new status
		nc.status = RECONNECTING
		if nc.ptmr != nil {
			nc.ptmr.Stop()
		}
		if nc.conn != nil {
			nc.bw.Flush()
			nc.conn.Close()
			nc.conn = nil
		}

		// Create a new pending buffer to underpin the bufio Writer while
		// we are reconnecting.
		nc.pending = &bytes.Buffer{}
		nc.bw = bufio.NewWriterSize(nc.pending, nc.Opts.ReconnectBufSize)

		go nc.doReconnect()
		nc.mu.Unlock()
		return
	}
```

Reconnection logic

Then, on reconnect it is flushed once preparing to send all the subscriptions again.

`CONNECT...\r\nPING\r\nSUB foo 1\r\nSUB bar 2\r\nPUB hello 5\r\nworld\r\n`

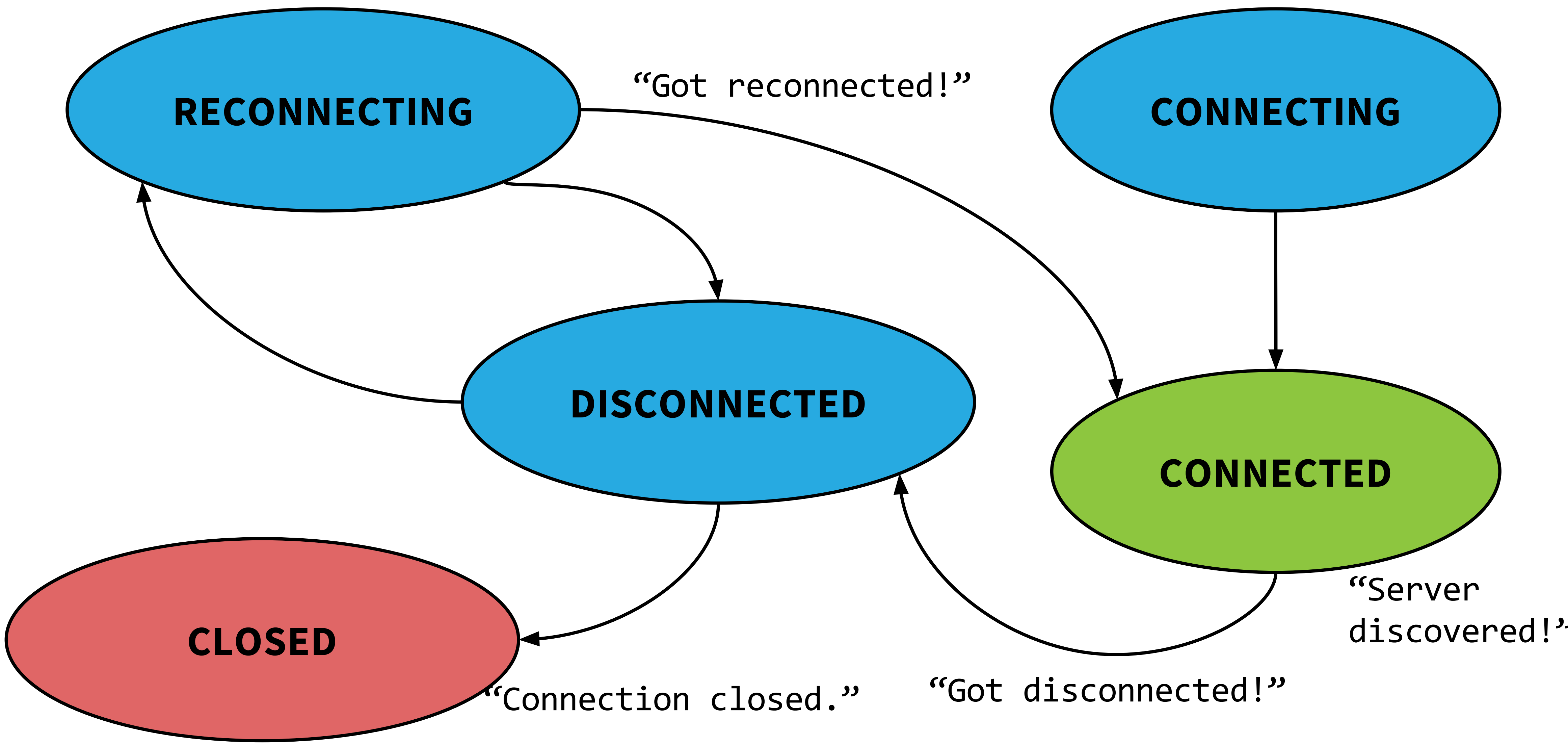
```
func (nc *Conn) doReconnect() { // note: a lot was elided here
        for len(nc.srvPool) > 0 {
                server, _ := nc.selectNextServer()
                nc.createConn()
                nc.resendSubscriptions()
                if nc.pending == nil {
                        if nc.pending.Len() > 0 {
                                nc.bw.Write(nc.pending.Bytes())
                        }
                }
                // Flush the buffer and retry next server on failure
                nc.err = nc.bw.Flush()
                if nc.err != nil {
                        nc.status = RECONNECTING
                        continue
                }
```

Disconnection and Reconnection Callbacks

We can set event callbacks via connect options which could be helpful for debugging or adding custom logic.

```
nc, err = nats.Connect("nats://demo.nats.io:4222",
        nats.MaxReconnects(5),
        nats.ReconnectWait(2 * time.Second),
        nats.DisconnectHandler(func(nc *nats.Conn) {
                log.Printf("Got disconnected!\n")
        }),
        nats.ReconnectHandler(func(nc *nats.Conn) {
                log.Printf("Got reconnected to %v!\n", nc.ConnectedUrl())
        }),
        nats.ClosedHandler(func(nc *nats.Conn) {
                log.Printf("Connection closed. Reason: %q\n", nc.LastError())
        }),
        nats.ErrorHandler(func(nc *nats.Conn, sub *nats.Subscription, err error) {
                log.Printf("Error: %s\n", err)
        })
)
```

Disconnection and Reconnection Callbacks



Async Error & Event Callbacks

In order to ensure ordering of event callbacks, a channel is used…

```
// Create the async callback channel.
nc.ach = make(chan asyncCB, 32)
```

```
// Slow consumer error in case our buffered channel blocked in a Subscription.
if nc.Opts.AsyncErrorCB != nil {
	nc.ach <- func() { nc.Opts.AsyncErrorCB(nc, sub, ErrSlowConsumer) }
}

// On disconnection events
if nc.Opts.DisconnectedCB != nil && nc.conn != nil {
	nc.ach <- func() { nc.Opts.DisconnectedCB(nc) }
}
if nc.Opts.ClosedCB != nil {
	nc.ach <- func() { nc.Opts.ClosedCB(nc) }
}

// On reconnect event
if nc.Opts.ReconnectedCB != nil {
	nc.ach <- func() { nc.Opts.ReconnectedCB(nc) }
}
```

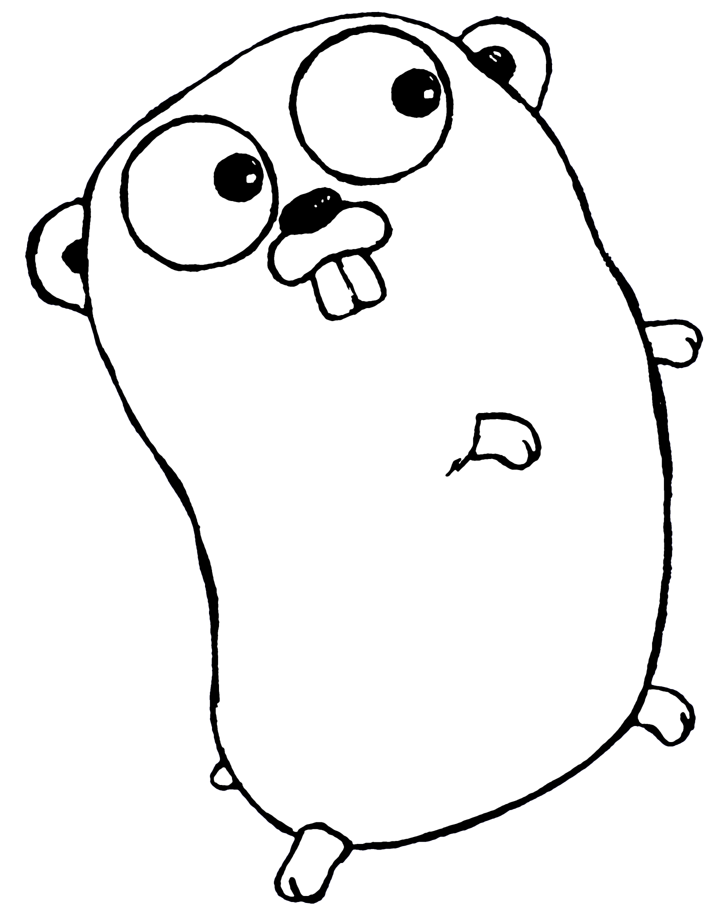
Async Error & Event Callbacks

…then an *asyncDispatch* goroutine will be responsible of calling them in sequence.

```go
// Connect will attempt to connect to a NATS server with multiple options.
func (o Options) Connect() (*Conn, error) {
	nc := &Conn{Opts: o}
	// ...

	// Spin up the async cb dispatcher on success
	go nc.asyncDispatch()
```

```go
// asyncDispatch is responsible for calling any async callbacks
func (nc *Conn) asyncDispatch() {
	// ...
	// Loop on the channel and process async callbacks.
	for {
		if f, ok := <-ach; !ok {
			return
		} else {
			f()
		}
```

- Intro to the NATS Project
- The NATS Protocol
- NATS Client Deep Dive
  - IO Engine
  - Connecting & Graceful Reconnection
  - **Sync & Async Subscriptions**
    - Channels and Callback based APIs
  - Request / Response APIs
  - Performance



Subscription Types

There are multiple APIs for processing the messages received from server.

```go
// Async Subscriber
nc.Subscribe("foo", func(m *nats.Msg) {
    log.Printf("Received a message: %s\n", string(m.Data))
})

// Sync Subscriber which blocks
sub, err := nc.SubscribeSync("foo")
m, err := sub.NextMsg(timeout)

// Channel Subscriber
ch := make(chan *nats.Msg, 64)
sub, err := nc.ChanSubscribe("foo", ch)
msg := <- ch
```

Subscription Types

Each type of subscription has a *QueueSubscribe* variation too.

```go
// Async Queue Subscriber
nc.QueueSubscribe("foo", "group", func(m *nats.Msg) {
    log.Printf("Received a message: %s\n", string(m.Data))
})

// Sync Queue Subscriber which blocks
sub, err := nc.QueueSubscribeSync("foo", "group")
m, err := sub.NextMsg(timeout)

// Channel Subscriber
ch := make(chan *nats.Msg, 64)
sub, err := nc.ChanQueueSubscribe("foo", "group", ch)
msg := <- ch
```

Async Subscribers

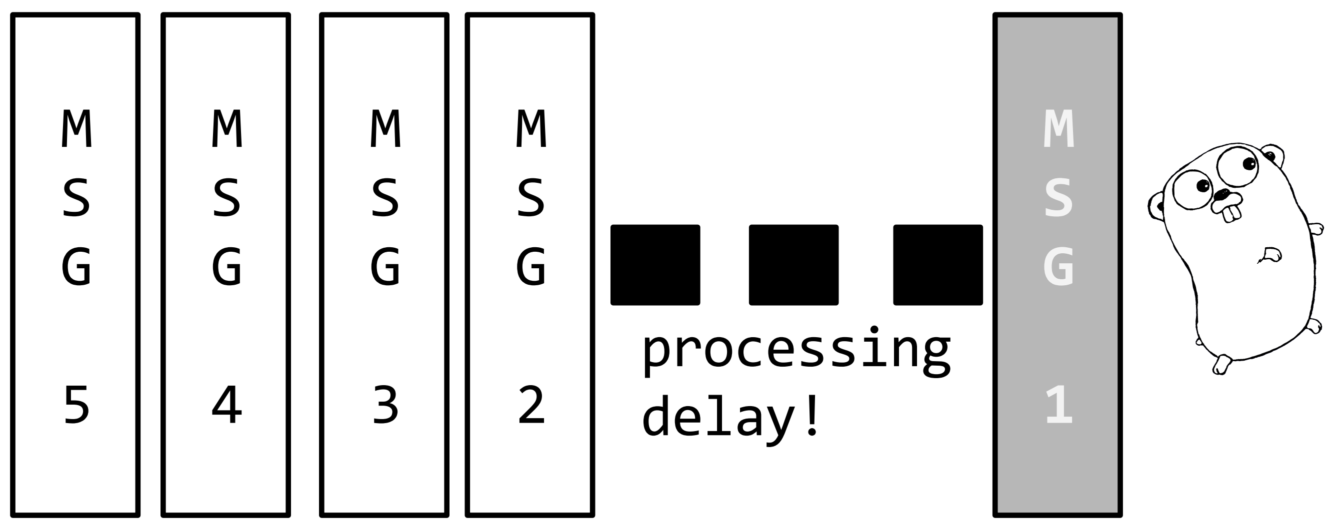
`Subscribe` takes a subject and callback and then dispatches each of the messages to the callback, processing them sequentially.

```
nc.Subscribe("foo", func(m *nats.Msg) {
    log.Printf("Received a message: %s\n", string(m.Data))
})
```

Async Subscribers

Then, we need to keep in mind that when multiple messages are received on the same subscription, there is potential of *head of line* blocking issues.

```
nc.Subscribe("foo", func(m *nats.Msg) {
        log.Printf("Received a message: %s\n", string(m.Data))
        // Next message would not be processed until
        // this sleep is done.
        time.Sleep(5 * time.Second)
})
```



Async Subscribers

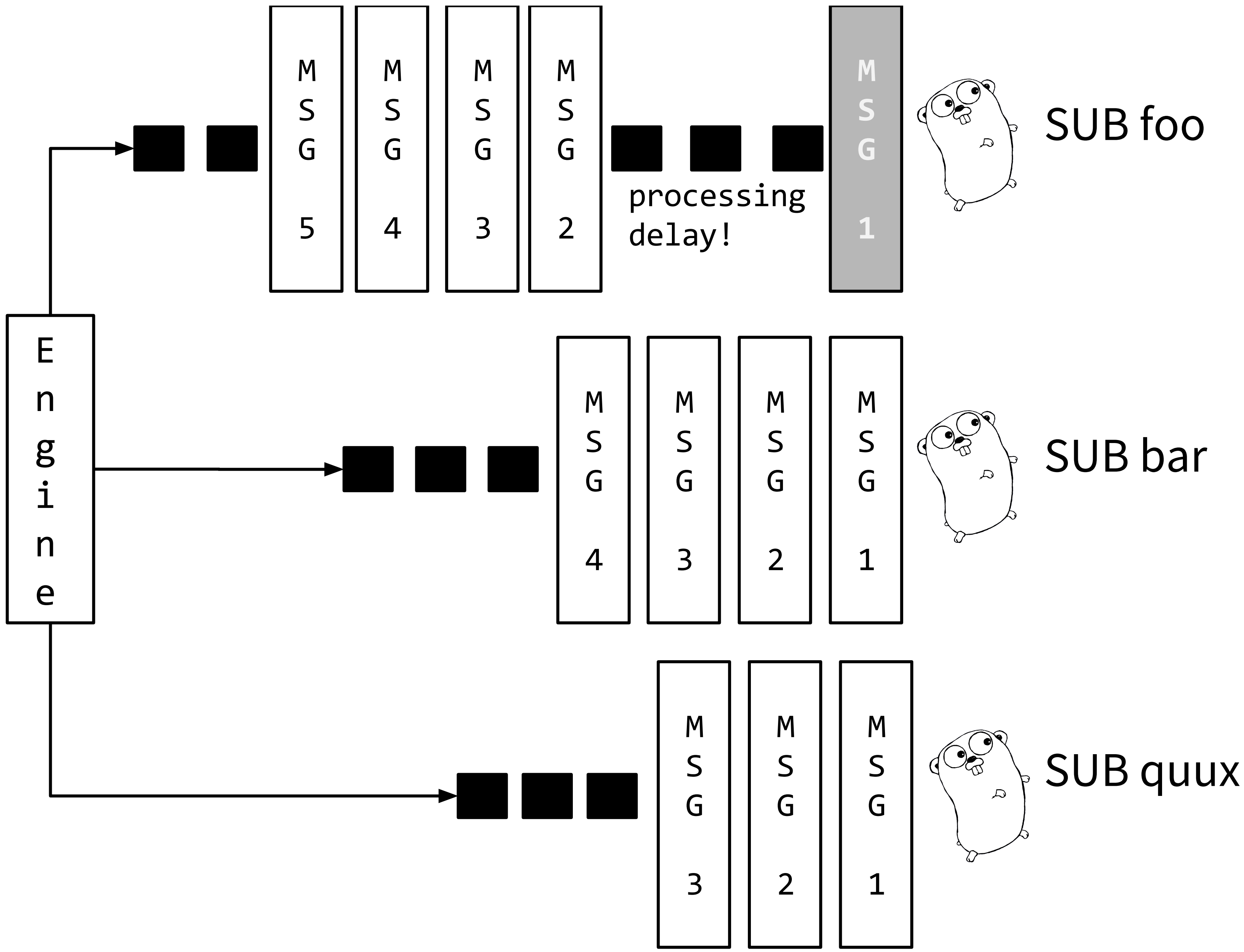
If that is an issue, user can opt into parallelize the processing by spinning up goroutines when receiving a message.

```
nc.Subscribe("foo", func(m *nats.Msg) {
	log.Printf("Received a message: %s\n", string(m.Data))
	go func(){
		// Next message would not block because
		// of this sleep.
		time.Sleep(5 * time.Second)
	}()
})
```

Determining whether to use a goroutine and implementing backpressure is up to the user.

Async Subscribers

Each Async Subscription has its own goroutine for processing so other subscriptions can still be processing messages (they do not block each other).



Async Subscribers

Async subscribers are implemented via a *linked list* of messages as they were received, in combination with use of conditional variables (no channels)

```
type Subscription struct {
        // ...
        pHead *Msg
        pTail *Msg
        pCond *sync.Cond
        // ...
```

One main benefit of this approach, is that the list can grow only as needed when not sure of how many messages will be received.

This way when having many subscriptions we do not have to allocate a channel of a large size in order to not be worried of dropping messages.

Async Subscribers

When subscribing, a goroutine is spun for the subscription, waiting for the parser to feed us new messages.

```
func (nc *Conn) subscribe(...) {
        // ...
        // If we have an async callback, start up a sub specific
        // goroutine to deliver the messages.
        if cb != nil {
                sub.typ = AsyncSubscription
                sub.pCond = sync.NewCond(&sub.mu)
                go nc.waitForMsgs(sub)
```

Async Subscribers

Then we wait for the conditional…

```
func (nc *Conn) waitForMsgs(s *Subscription) {
        var closed bool
        var delivered, max uint64

        for {
                s.mu.Lock()
                if s.pHead == nil && !s.closed {
                        // Parser will signal us when a message for
                        // this subscription is received.
                        s.pCond.Wait()
                }

                // ...
                mcb := s.mcb
                // ...

                // Deliver the message.
                if m != nil && (max == 0 || delivered <= max) {
                        mcb(m)
                }
```

Async Subscribers

…until signaled by the parser when getting a message.

```
func (nc *Conn) processMsg(data []byte) {
        // ...
        sub := nc.subs[nc.ps.ma.sid]
        // ...
        if sub.pHead == nil {
                sub.pHead = m
                sub.pTail = m
                sub.pCond.Signal() //
        } else {
                sub.pTail.next = m
                sub.pTail = m
        }
```

Sync & Channel Subscribers

Internally, they are fairly similar since both are channel based.

```
// Sync Subscriber which blocks until receiving next message.
sub, err := nc.SubscribeSync("foo")
m, err := sub.NextMsg(timeout)

// Channel Subscriber
ch := make(chan *nats.Msg, 64)
sub, err := nc.ChanSubscribe("foo", ch)
msg := <- ch
```

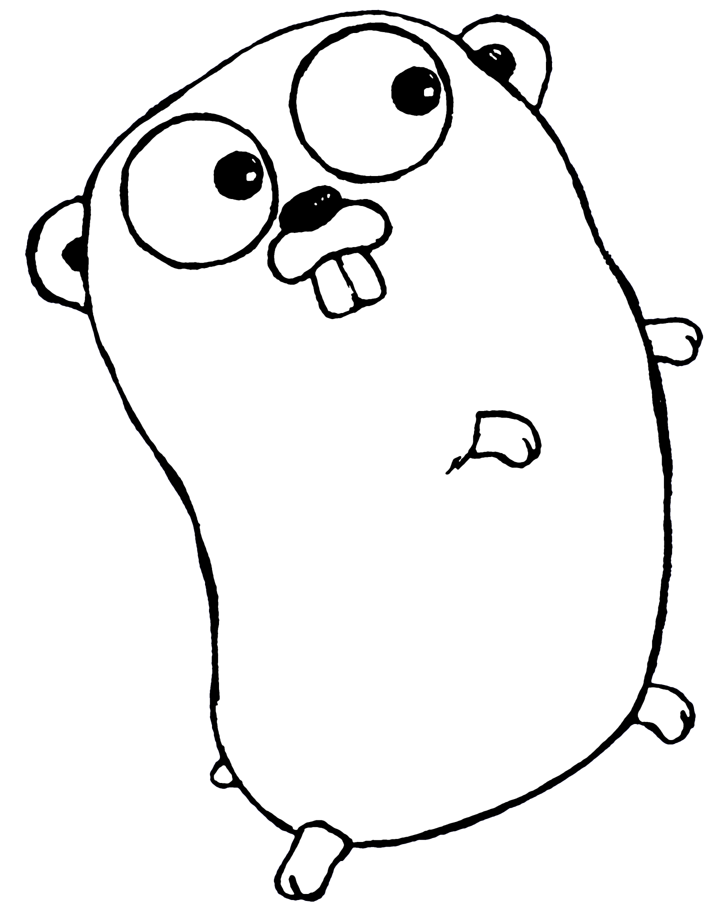
Sync & Channel Subscribers

Once parser yields a message, it is sent to the channel.

```
func (nc *Conn) processMsg(data []byte) {
        // ...
        sub := nc.subs[nc.ps.ma.sid]
        // ...
        // We have two modes of delivery. One is the channel, used by channel
        // subscribers and syncSubscribers, the other is a linked list for async.
        if sub.mch != nil {
                select {
                case sub.mch <- m:
                default:
                        goto slowConsumer
                }
```

Client needs to be ready to receive, otherwise if it blocks then it would drop the message and report a *slow consumer* error in the async error callback.

- Intro to the NATS Project
- The NATS Protocol
- NATS Client Deep Dive
  - IO Engine
  - Connecting & Graceful Reconnection
  - Sync & Async Subscriptions
  - **Request/Response**
    - Unique inboxes generation
    - Context support for cancellation
  - Performance



Request / Response

Building on top of PubSub mechanisms available, the client has support for 1:1 style of communication.

```go
func main() {
	nc, err := nats.Connect("nats://127.0.0.1:4222")
	if err != nil {
		log.Fatalf("Error: %s", err)
	}
	nc.Subscribe("help", func(m *nats.Msg) {
		log.Printf("[Received] %+v", m)
		nc.Publish(m.Reply, []byte("I can help"))
	})
	response, err := nc.Request("help", []byte("please"), 2*time.Second)
	if err != nil {
		log.Fatalf("Error: %s", err)
	}
	log.Printf("[Response] %+v", response)
}
```

Request / Response

Protocol-wise, this is achieved by using ephemeral subscriptions.

Requestor

```
SUB _INBOX.95jM7MgZoxZt94fhLpewjg  2
UNSUB 2 1
PUB help _INBOX.95jM7MgZoxZt94fhLpewjg 6
please
MSG _INBOX.95jM7MgZoxZt94fhLpewjg 2 10
I can help
```

Responder

```
SUB help  1
MSG help 1 _INBOX.95jM7MgZoxZt94fhLpewjg 6
please
PUB _INBOX.95jM7MgZoxZt94fhLpewjg 10
I can help
```

Unique Inboxes Generation

The subscriptions identifiers are generated via the `NUID` which can generate identifiers as fast as ~16 M per second.

https://github.com/nats-io/nuid

NUID

license MIT | go report A+ | build passing | release v1.0.0 | godoc reference | coverage 96%

A highly performant unique identifier generator.

Basic Usage

```
// Utilize the global locked instance
nuid := nuid.Next()

// Create an instance, these are not locked.
n := nuid.New()
nuid = n.Next()

// Generate a new crypto/rand seeded prefix.
// Generally not needed, happens automatically.
n.RandomizePrefix()
```

Request / Response

```
func (nc *Conn) Request(subj string, data []byte, timeout time.Duration) (
        *Msg,
        error,
) {
        inbox := NewInbox()
        ch := make(chan *Msg, RequestChanLen)

        s, err := nc.subscribe(inbox, _EMPTY_, nil, ch)
        if err != nil {
                return nil, err
        }
        s.AutoUnsubscribe(1) // UNSUB after receiving 1
        defer s.Unsubscribe()

        // PUB help _INBOX.nwOOaWSe...
        err = nc.PublishRequest(subj, inbox, data)
        if err != nil {
                return nil, err
        }

        // Block until MSG is received or timeout.
        return s.NextMsg(timeout)
}
```

Request / Response (Context variation)

```go
func (nc *Conn) RequestWithContext(ctx context.Context, subj string, data []byte) (
	*Msg,
	error,
) {
	inbox := NewInbox()
	ch := make(chan *Msg, RequestChanLen)

	s, err := nc.subscribe(inbox, _EMPTY_, nil, ch)
	if err != nil {
		return nil, err
	}
	s.AutoUnsubscribe(1) // UNSUB after receiving 1
	defer s.Unsubscribe()

	// PUB help _INBOX.nwOOaWSe...
	err = nc.PublishRequest(subj, inbox, data)
	if err != nil {
		return nil, err
	}

	// Block until MSG is received or context canceled.
	return s.NextMsgWithContext(ctx)
}
```

Request / Response (+v1.3.0)

In next release of the client, a less chatty version of the request/response mechanism is available.

Requestor

```
SUB _INBOX.nwOOaWSeWrt0ok20pKFfNz.*  2
PUB help _INBOX.nwOOaWSeWrt0ok20pKFfNz.nwOOaWSeWrt0ok20pKFfPo 6
please
MSG _INBOX.nwOOaWSeWrt0ok20pKFfNz.nwOOaWSeWrt0ok20pKFfPo 2 10
I can help
```

Responder

```
SUB help  1
MSG help 1 _INBOX.nwOOaWSeWrt0ok20pKFfNz.nwOOaWSeWrt0ok20pKFfPo 6
please
PUB _INBOX.nwOOaWSeWrt0ok20pKFfNz.nwOOaWSeWrt0ok20pKFfPo 10
I can help
```

Request / Response (new version)

```go
func (nc *Conn) Request(subj string, data []byte, timeout time.Duration) (*Msg, error) {
        // ... (note: not actual implementation)
        nc.mu.Lock()
        if nc.respMap == nil {
                // Wildcard subject for all requests, e.g. INBOX.abcd.*
                nc.respSub = fmt.Sprintf("%s.*", NewInbox())
                nc.respMap = make(map[string]chan *Msg)
        }
        mch := make(chan *Msg, RequestChanLen)
        respInbox := nc.newRespInbox() // Await for reply on subject: _INBOX.abcd.xyz
        token := respToken(respInbox)  // Last token identifying this request: xyz
        nc.respMap[token] = mch
        // ...
        nc.mu.Unlock()

        // Broadcast request, then wait for reply or timeout.
        nc.PublishRequest(subj, respInbox, data)

        // (Wait for message or timer timeout)
        select {
        case msg, ok := <-mch:
                // Message received or Connection Closed
        case <-time.After(timeout):
                // Discard request and yield error
                return nil, ErrTimeout
        }
```

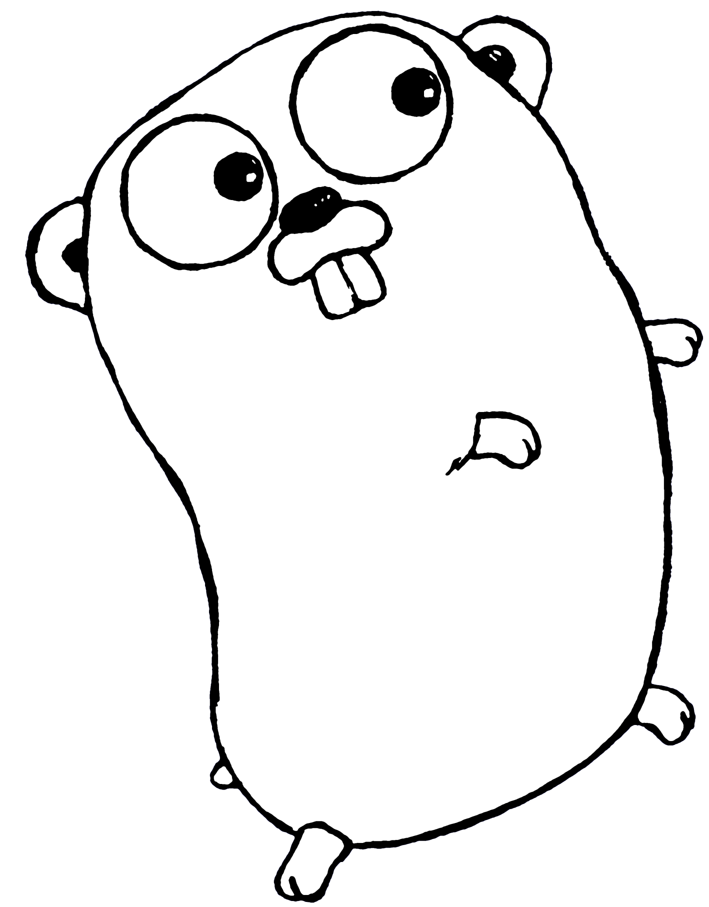
Request / Response (new version+context)

```
func (nc *Conn) RequestWithContext(ctx context.Context, subj string, data []byte) (*Msg, er
	// ... (note: not actual implementation)
	nc.mu.Lock()
	if nc.respMap == nil {
		// Wildcard subject for all requests, e.g. INBOX.abcd.*
		nc.respSub = fmt.Sprintf("%s.*", NewInbox())
		nc.respMap = make(map[string]chan *Msg)
	}
	mch := make(chan *Msg, RequestChanLen)
	respInbox := nc.newRespInbox() // Await for reply on subject: _INBOX.abcd.xyz
	token := respToken(respInbox)  // Last token identifying this request: xyz
	nc.respMap[token] = mch
	// ...
	nc.mu.Unlock()

	// Broadcast request, then wait for reply or timeout.
	nc.PublishRequest(subj, respInbox, data)

	// (Wait for message or timer timeout)
	select {
	case msg, ok := <-mch:
		// Message received or Connection Closed
	case <-ctx.Done():
		// Discard request and yield error
		return nil, ErrTimeout
	}
```

Request / Response (new version)

```
// In Request(...)
//
// Make sure scoped subscription is setup only once via sync.Once
var err error
nc.respSetup.Do(func() {
        var s *Subscription
        // Create global subscription: INBOX.abcd.*
        s, err = nc.Subscribe(ginbox, func(m *Msg){
                // Get token: xyz ← INBOX.abcd.xyz
                rt := respToken(m.Subject)
                nc.mu.Lock()

                // Signal channel that we received the message.
                mch := nc.respMap[rt]
                delete(nc.respMap, rt)
                nc.mu.Unlock()
                select {
                case mch <- m:
                default:
                        return
                }
        })

        nc.mu.Lock()
        nc.respMux = s
        nc.mu.Unlock()
})
```

- Intro to the NATS Project
- The NATS Protocol
- NATS Client Deep Dive
  - IO Engine
  - Connecting & Graceful Reconnection
  - Synchronous & Asynchronous Subscriptions
  - Request / Response APIs
  - **Performance**



nats-bench

Current implementation of the client/server show a performance of ~10M messages per second (1B payload)

```
NATS @ ~/repos/nats-dev/src/github.com/nats-io/go-nats (master) $ ./nats-bench -np 20 -n 20000000 -ms 1 a
Starting benchmark [msgs=20000000, msgsize=1, pubs=20, subs=0]
Pub stats: 10,631,133 msgs/sec ~ 10.14 MB/sec
 [1] 666,289 msgs/sec ~ 650.67 KB/sec (1000000 msgs)
 [2] 646,907 msgs/sec ~ 631.75 KB/sec (1000000 msgs)
 [3] 627,218 msgs/sec ~ 612.52 KB/sec (1000000 msgs)
 [4] 607,676 msgs/sec ~ 593.43 KB/sec (1000000 msgs)
 [5] 587,845 msgs/sec ~ 574.07 KB/sec (1000000 msgs)
 [6] 592,241 msgs/sec ~ 578.36 KB/sec (1000000 msgs)
 [7] 600,041 msgs/sec ~ 585.98 KB/sec (1000000 msgs)
 [8] 582,020 msgs/sec ~ 568.38 KB/sec (1000000 msgs)
 [9] 580,952 msgs/sec ~ 567.34 KB/sec (1000000 msgs)
 [10] 559,497 msgs/sec ~ 546.38 KB/sec (1000000 msgs)
 [11] 561,585 msgs/sec ~ 548.42 KB/sec (1000000 msgs)
 [12] 552,145 msgs/sec ~ 539.20 KB/sec (1000000 msgs)
 [13] 543,149 msgs/sec ~ 530.42 KB/sec (1000000 msgs)
 [14] 558,306 msgs/sec ~ 545.22 KB/sec (1000000 msgs)
 [15] 553,398 msgs/sec ~ 540.43 KB/sec (1000000 msgs)
 [16] 538,763 msgs/sec ~ 526.14 KB/sec (1000000 msgs)
 [17] 544,214 msgs/sec ~ 531.46 KB/sec (1000000 msgs)
 [18] 542,912 msgs/sec ~ 530.19 KB/sec (1000000 msgs)
 [19] 541,483 msgs/sec ~ 528.79 KB/sec (1000000 msgs)
 [20] 538,483 msgs/sec ~ 525.86 KB/sec (1000000 msgs)
 min 538,483 | avg 576,256 | max 666,289 | stddev 36,668 msgs
NATS @ ~/repos/nats-dev/src/github.com/nats-io/go-nats (master) $
```

Let's try escape analysis on a simple snippet

```go
func main() {
	nc, err := nats.Connect("nats://127.0.0.1:4222")
	if err != nil {
		os.Exit(1)
	}

	nc.Subscribe("help", func(m *nats.Msg) {
		println("[Received]", m)
	})
	nc.Flush()

	nc.Request("help", []byte("hello world"), 1*time.Second)

	nc.Publish("help", []byte("please"))
	nc.Flush()

	time.Sleep(1 * time.Second)
}
```

```
go run -gcflags "-m" example.go

# ...
./example.go:16: can inline main.func1
./example.go:16: func literal escapes to heap
./example.go:16: func literal escapes to heap
./example.go:21: ([]byte)("hello world") escapes to heap #
./example.go:23: ([]byte)("please") escapes to heap      #
./example.go:16: main.func1 m does not escape
[Received] 0xc420100050
[Received] 0xc4200de050
```


ESCAPES
ESCAPES EVERYWHERE
imgflip.com

Turns out that all the messages being published escape and end up in the heap, why??

```
    nc.Request("help", []byte("hello world"), 1*time.Second)

    nc.Publish("help", []byte("please"))
```

When publishing, we are just doing `bufio.Writer` after all…

```
func (nc *Conn) publish(subj, reply string, data []byte) error {
	// ...
	_, err := nc.bw.Write(msgh)
	if err == nil {
		_, err = nc.bw.Write(data)
	}
	if err == nil {
		_, err = nc.bw.WriteString(_CRLF_)
	}
```

https://github.com/nats-io/go-nats/blob/master/nats.go#L1956-L1965

bufio: Writer.Writer's []byte argument escapes #5492

Closed **bradfitz** opened this issue on May 16, 2013 · 3 comments

bradfitz commented on May 16, 2013 **Owner**

```
It's not possible to Write to a *bufio.Writer without the []byte argument escaping, due
to the fast path passing p through an interface value:

package bufio
...
func (b *Writer) Write(p []byte) (nn int, err error) {
    for len(p) > b.Available() && b.err == nil {
            var n int
        if b.Buffered() == 0 {
            // Large write, empty buffer.
            // Write directly from p to avoid copy.
            n, b.err = b.wr.Write(p)    // <------------- escapes
        } else {
                    n = copy(b.buf[b.n:], p)
            b.n += n
            b.Flush()
        }
            nn += n
        p = p[n:]
    }
```

https://github.com/golang/go/issues/5492

```go
// Write writes the contents of p into the buffer.
// It returns the number of bytes written.
// If nn < len(p), it also returns an error explaining
// why the write is short.
func (b *Writer) Write(p []byte) (nn int, err error) {
	for len(p) > b.Available() && b.err == nil {
		var n int
		if b.Buffered() == 0 {
			// Large write, empty buffer.
			// Write directly from p to avoid copy.
			n, b.err = b.wr.Write(p) // <---- Interface causes alloc!
		} else {
			n = copy(b.buf[b.n:], p)
			b.n += n
			b.Flush()
		}
		nn += n
		p = p[n:]
	}
```

https://github.com/golang/go/blob/ae238688d2813e83f16050408487ea34ba1c2fff/src/bufio/bufio.go#L600-L602

We could swap out the `bufio.Writer` implementation with a custom one, use concrete types, etc. then prevent the escaping…

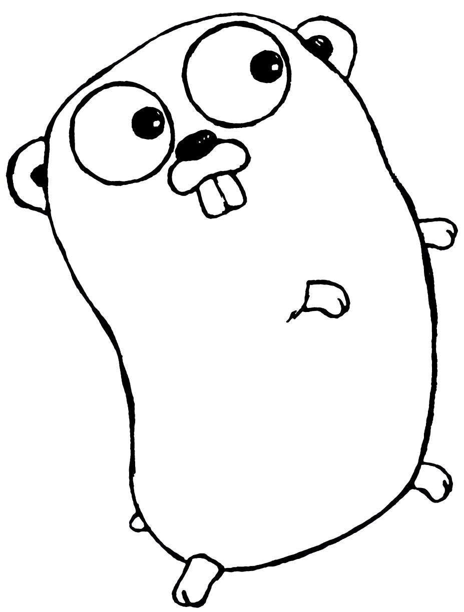
```
+	var bwr *bufioWriter
+	if conn, ok := w.(*net.TCPConn); ok {
+		bwr = &bufioWriter{
+			buf:  make([]byte, size),
+			wr:   w,
+			conn: conn,
+		}
+	} else if buffer, ok := w.(*bytes.Buffer); ok {
+		bwr = &bufioWriter{
+			buf:     make([]byte, size),
+			wr:      w,
+			pending: buffer,
+		}
+	} else if sconn, ok := w.(*tls.Conn); ok {
+		bwr = &bufioWriter{
+			buf:   make([]byte, size),
+			wr:    w,
+			sconn: sconn,
+		}
+	}
```

Performance matters, but is it worth it the complexity?

Although for converting number to byte, approach not using `strconv.AppendInt` was taken since much more performant.

```
// msgh = strconv.AppendInt(msgh, int64(len(data)), 10)

var b [12]byte
var i = len(b)
if len(data) > 0 {
	for l := len(data); l > 0; l /= 10 {
		i -= 1
		b[i] = digits[l%10]
	}
} else {
	i -= 1
	b[i] = digits[0]
}
```

Wrapping up



- Techniques used and trade-offs in the Go client:
  - Backpressure of I/O and writes coalescing
  - Fast protocol parsing
  - Graceful reconnection on server failover
  - Usage & Non-usage of channels for communicating
  - Closures, callbacks & channels for events/custom logic

Conclusions

- NATS approach to simplicity fits really well with what Go provides and has helped the project greatly.
- Go is a very flexible systems programming language, allowing us to choose our own trade offs.
- There is great tooling in Go which makes it possible to analyze when to take them.

Thanks!

github.com/nats-io / @nats_io

Twitter: @wallyqs