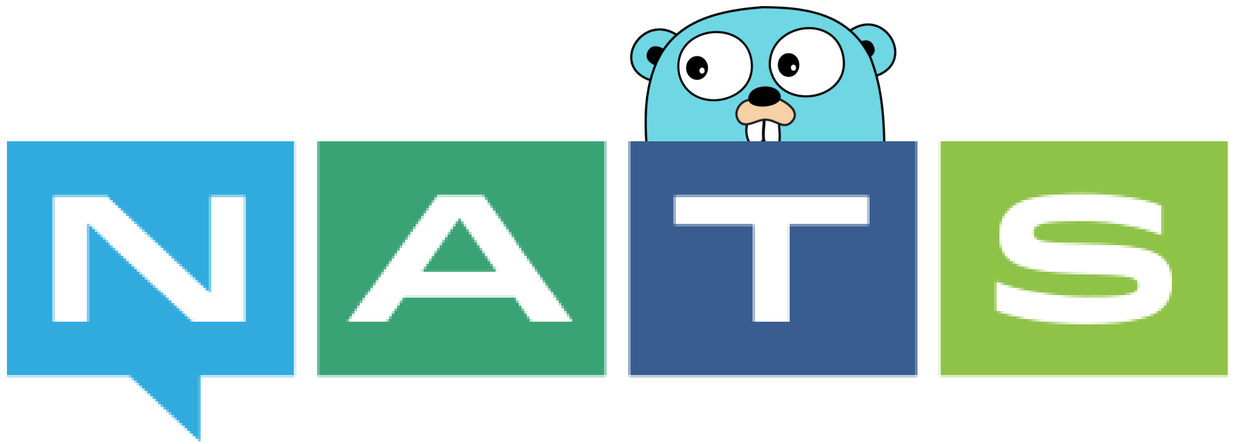
NATS Community Day

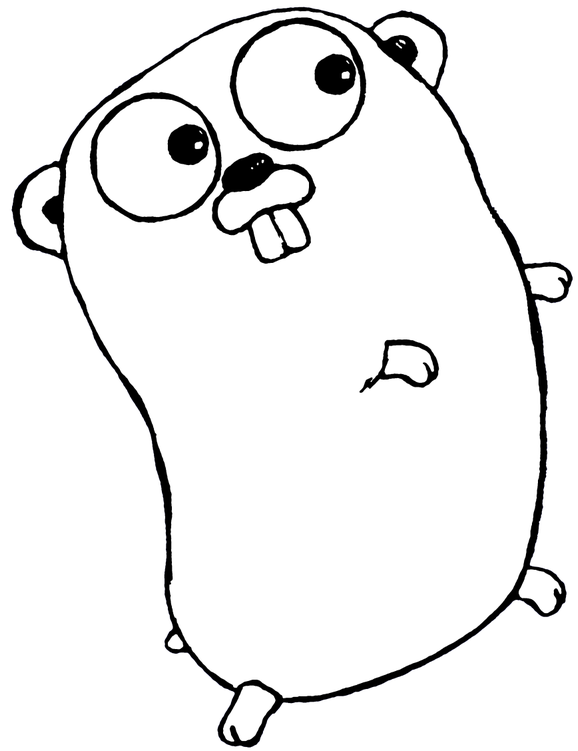


Waldemar Quevedo / @wallyqs

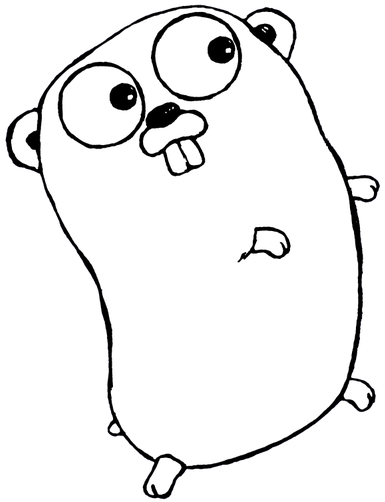
GopherCon 2017

Agenda

- NATS Protocol
  - Walk through on a very basic client
- Operational aspects from NATS
  - Configuration
  - Clustering
- Monitoring Tools
  - nats-top
  - Prometheus NATS Exporter



Writing a Minimal Client in Go



Implementation (~ 260 lines)

https://goo.gl/UAJ2Xt

A Simple Example

Client which connects and publishes a message.

```
package main

import (
        "log"
        "net"
)

func main() {
        // Establish TCP connection.
        conn, err := net.Dial("tcp", "demo.nats.io:4222")
        if err != nil {
                log.Fatalf("Error: %s", err)
        }

        // Publish 'world' message on 'hello' subject.
        pub := []byte("PUB hello 5\r\nworld\r\n")
        _, err = conn.Write(pub)
        if err != nil {
                log.Fatalf("Error: %s", err)
        }

        // Done!
        conn.Close()
}
```

The **PUB** command

Note: Payload is opaque to the protocol.

```
PUB subject number_of_bytes
payload
```

When interacting with the server, we have to announce the number of expected number of bytes in the control line:

```
PUB hello 5
world
```

Though there is a limit in the payload size (by default 1MB). We can get this value from the initial `INFO` encoded in `JSON`:

```
{
    "auth_required": false,
    "go": "go1.7.4",
    "host": "0.0.0.0",
    "max_payload": 1048576,  //
    "port": 4222,
    "server_id": "zrPhBhrjbbUdp2vndDIvE7",
    "tls_required": false,
    "tls_verify": false,
    "version": "0.9.6"
}
```

Sending more bytes than what announced by the server on connect, will cause a client disconnect.

```
NATS @ ~ () $ telnet demo.nats.io 4222
Trying 107.170.221.32...
Connected to demo.nats.io.
Escape character is '^]'.
INFO {"server_id":"zrPhBhrjbbUdp2vndDIvE7","version":"0.9.6","go":"go1.7.4","host":"0.0.0.0","port":4222,
"auth_required":false,"ssl_required":false,"tls_required":false,"tls_verify":false,"max_payload":1048576}

pub hell
```

Processing INFO

```go
package main

import (
        "bufio"
        "encoding/json"
        "log"
        "net"
        "strings"
)

func main() {
        // Establishing a connection to a NATS server (blocks!)
        conn, _ := net.Dial("tcp", "demo.nats.io:4222")

        // Read the first line (max control line is 1024 by default btw)
        line, _ := bufio.NewReader(conn).ReadString('\n')

        // Split on first space from control line
        // INFO {...}
        toks := strings.SplitN(line, " ", 2)
        info := struct {
                MaxPayload int `json:"max_payload"`
        }{}
        json.Unmarshal([]byte(toks[1]), &info)

        // => 1048576 bytes by default
        log.Println(info.MaxPayload)
}
```

Registering interest on subject with **SUB**

A **SUB** control line contains the subject and an identifier of the subscription *for the client*.

```
SUB subject identifier
```

e.g. *Register interest on 'foo' and 'hello' subjects using identifiers 1 and 100*

```
SUB foo 1
SUB hello 100
```

Receiving messages from server

A **MSG** delivered by the server looks like this:

```
MSG subject identifier number_of_bytes
payload
```

For example, given that we made a subscription on **SUB foo 1** and receiving a message with a '*bar*' payload we would get:

```
MSG foo 1 3
bar
```

Example consuming messages from a subscription

```
r := bufio.NewReader(conn)
w := bufio.NewWriter(conn)

// Subscribing to a couple of subjects
w.WriteString("SUB foo 1\r\n")
w.WriteString("SUB hello 100\r\n")

// Publishing a couple of commands.
w.WriteString("PUB foo 3\r\nbar\r\n")
w.WriteString("PUB hello 5\r\nworld\r\n")
w.Flush()

for {
        line, err := r.ReadString('\n')
        if err != nil {
                log.Fatalf("Error: %s", err)
        }
        log.Print(line)
}
```

Output would look something like this:

```
INFO {"server_id":"Cxc2FA7Nvx97w0nyoltdRf","version":"0.9.6",...
+OK
+OK
+OK
MSG foo 1 3       # matched: SUB foo 1 and need to read 3 bytes
bar
+OK
MSG hello 100 5   # matched: SUB hello 100 and need to read 5 bytes
world
```

Output would look something like this:

```
INFO {"server_id":"Cxc2FA7Nvx97w0nyoltdRf","version":"0.9.6",...
+OK        # ?
+OK        # ?
MSG foo 1 3
bar
+OK        # ?
MSG hello 100 5
world
```

+OK?

These replies from the server are received because by default clients will follow the *verbose mode* from the protocol, where the server will be acking back with `+OK` after processing each command.

We can disable this behavior via `CONNECT`.

Disabling verbose mode via `CONNECT`

```
r := bufio.NewReader(conn)
w := bufio.NewWriter(conn)

// Disable verbose mode
connect := struct {
        Verbose bool   `json:"verbose"`
}{
        Verbose: false,
}
connectOp, _ := json.Marshal(connect)
connectCmd := fmt.Sprintf("CONNECT %s\r\n", connectOp)
w.WriteString(connectCmd)

w.WriteString("SUB foo 1\r\n")
w.WriteString("SUB hello 100\r\n")
w.WriteString("PUB foo 3\r\nbar\r\n")
w.WriteString("PUB hello 5\r\nworld\r\n")
w.Flush()

for {
        line, err := r.ReadString('\n')
        if err != nil {
                log.Fatalf("Error: %s", err)
        }
        log.Print(line)
}
```

Customized protocol interaction after `CONNECT`

```
INFO {"server_id":"Cxc2FA7Nvx97w0nyoltdRf","version":"0.9.6",...
MSG foo 1 3
bar
MSG hello 100 5
world
```

Further customizations via `CONNECT`

We could add a name to the client too…

```
w := bufio.NewWriter(conn)

// Disable verbose mode
connect := struct {
        Name    string `json:"name"`
        Verbose bool   `json:"verbose"`
}{
        Name:    "gopher",
        Verbose: false,
}
connectOp, _ := json.Marshal(connect)
connectCmd := fmt.Sprintf("CONNECT %s\r\n", connectOp)
w.WriteString(connectCmd)
w.Flush()
```

Useful to identify more easily a client via `nats-top`

```
NATS server version 0.9.6 (uptime: 39m23s)
Server:
  Load: CPU:  0.0%  Memory: 9.3M  Slow Consumers: 0
  In:   Msgs: 20  Bytes: 80  Msgs/Sec: 0.0  Bytes/Sec: 0
  Out:  Msgs: 20  Bytes: 80  Msgs/Sec: 0.0  Bytes/Sec: 0

Connections Polled: 1
  HOST              CID    NAME     SUBS    PENDING     MSGS_TO     MSGS_FROM   BYTES_TO    BYTES_FROM
  127.0.0.1:51940   10     gopher   2       0           2           2           8           8
```

At any point in time we may receive from the server any of the following commands:

- `MSG`
- `+OK`
- `-ERR`
- `PING`
- `PONG`
- `INFO`

By default we would get every 2 minutes a `PING` from server, so we must reply back otherwise will be disconnected…

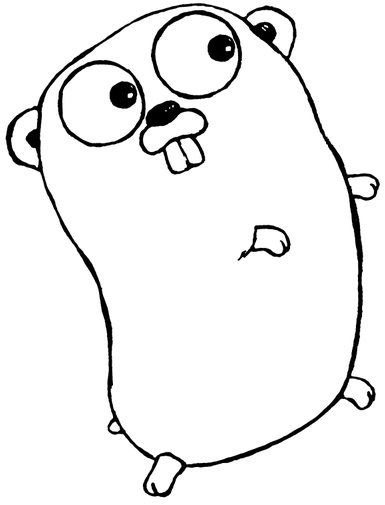
```
INFO {"server_id":"Cxc2FA7Nvx97w0nyoltdRf","version":"0.9.6",...
MSG foo 1 3
bar
MSG hello 100 5
world
PING
PING
-ERR 'Stale Connection'
```

A client connected to NATS is meant to try to always have an established connection to a server in the cluster.

One of the goals from NATS is to provide a *dial tone* for clients to communicate.

Writing a Minimal Client in Go

Protocol Parsing



Parsing the NATS Protocol

Go makes it simple for us to write a naive implementation using `bufio` and `strings`.

```
for {
        line, err := r.ReadString('\n')
        if err != nil {
                log.Fatalf("Error: %s", err)
        }
        args := strings.SplitN(line, " ", 2)
        if len(args) < 1 {
                log.Fatalf("Error: malformed control line")
        }
        op := strings.TrimSpace(args[0])
        switch op {
        case "MSG":
        case "INFO":
        case "PING":
        case "PONG":
        case "+OK":
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

`PING` → `PONG` reply to prevent disconnection

```
for {
        // ...
        switch op {
        case "MSG":
        case "INFO":
        case "PING":
                // Reply back to prevent stale connection error.
                err := w.WriteString("PONG\r\n")
                w.Flush()
                if err != nil {
                        log.Fatalf("Error: %s", err)
                }
        case "PONG":
        case "+OK":
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Processing `INFO`

We get one on connect and also when new servers join the cluster.

```
for {
        // ...
        switch op {
        case "MSG":
        case "INFO":
                info := struct {
                        MaxPayload int `json:"max_payload"`
                        ConnectUrls []string `json:"connect_urls"`
                }{}
                json.Unmarshal([]byte(args[1]), &info)
        case "PING": //
        case "PONG":
        case "+OK":
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Handling a received MSG

A MSG may be tagged with an reply inbox, so we have to handle 2 cases:

```
for {
        // ...
        switch op {
        case "MSG":
                // without reply: MSG foo 1 3\r\n
                // with reply:    MSG foo 1 bar 4\r\n
        case "INFO": //
        case "PING": //
        case "PONG":
        case "+OK":
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Handling a received MSG

Case 1) Without reply inbox:

```
switch op {
case "MSG":
        var subject, reply string
        var sid, size int

        n := strings.Count(args[1], " ") + 1
        switch n {
        case 3:
                // MSG foo 1 3\r\n
                // bar\r\n
                _, err := fmt.Sscanf(args[1], "%s %d %d",
                        &subject, &sid, &size)
                if err != nil {
                        log.Fatalf("Error: malformed control line: %s", err)
                }

                // Prepare buffer for the payload with the given size.
                payload := make([]byte, size)
                _, err = io.ReadFull(r, payload)
                if err != nil {
                        log.Fatalf("Error: problem gathering bytes: %s", err)
                }
```

Handling a received MSG

Case 2) With a reply inbox

```
switch op {
case "MSG":
        var subject, reply string
        var sid, size int

        n := strings.Count(args[1], " ") + 1
        switch n {
        case 4:
                // MSG foo 1 bar 4\r\n
                // quux\r\n
                _, err := fmt.Sscanf(args[1], "%s %d %s %d",
                        &subject, &sid, &reply, &size)
                if err != nil {
                        log.Fatalf("Error: malformed control line: %s", err)
                }

                // Prepare buffer for the payload with the given size.
                payload := make([]byte, size)
                _, err = io.ReadFull(r, payload)
                if err != nil {
                        log.Fatalf("Error: problem gathering bytes: %s", err)
                }
```

```
for {
        // ...
        switch op {
        case "MSG":  //
        case "INFO": //
        case "PING": //
        case "PONG":
        case "+OK":
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Handling PONG

This one is triggered by us when doing a `PING`, we may disconnect if we have many missing for example.

```
for {
        // ...
        switch op {
        case "MSG":  //
        case "INFO": //
        case "PING": //
        case "PONG": // skip for now
        case "+OK":
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Handling +OK

Nothing to handle

```
for {
        // ...
        switch op {
        case "MSG":  //
        case "INFO": //
        case "PING": //
        case "PONG": //
        case "+OK":  // do nothing
        case "-ERR":
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Handling –ERR

First argument is the reported error by server

```
for {
        // ...
        switch op {
        case "MSG":  //
        case "INFO": //
        case "PING": //
        case "PONG": //
        case "+OK":  //
        case "-ERR":
                log.Fatalf("Protocol Error: %s", args[1])
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

e.g. when not replying to many pings we would get

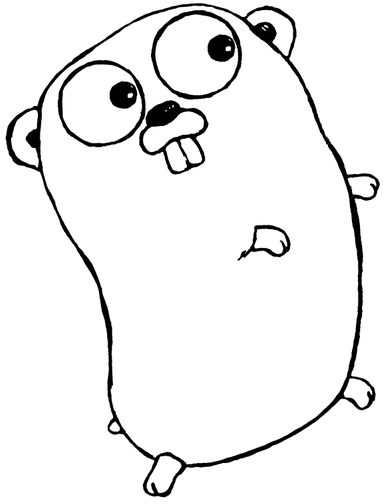
```
-ERR 'Stale Connection'
```

Mostly working parser

```
for {
        // ...
        switch op {
        case "MSG":  //
        case "INFO": //
        case "PING": //
        case "PONG": //
        case "+OK":  //
        case "-ERR": //
        default:
                log.Fatalf("Error: malformed control line")
        }
}
```

Writing a Minimal Client in Go

Client API



Public API for the minimal client

```
type Client interface {
    // Connect establishes connection to NATS.
    Connect(location string) error

    // Close wraps up connection to NATS.
    Close()

    // Publish sends a message to a subject.
    Publish(subj, reply string, payload []byte) error

    // Subscribe registers interest in subject and
    // delivers messages onto the provided callback.
    Subscribe(subj, queue string, cb func(subj, reply string, data []byte)) error
}
```

Connect API

```
func (c *client) Connect(netloc string) error {
        conn, err := net.Dial("tcp", netloc)
        if err != nil {
                return err
        }
        c.conn = conn
        c.r = bufio.NewReader(conn)
        c.w = bufio.NewWriter(conn)
        c.subs = make(map[int]func(string, string, []byte))

        connect := struct {
                Name    string `json:"name"`
                Verbose bool   `json:"verbose"`
        }{
                Name:    "gopher",
                Verbose: false,
        }
        connectOp, _ := json.Marshal(connect)
        connectCmd := fmt.Sprintf("CONNECT %s\r\n", connectOp)
        c.w.WriteString(connectCmd)
        c.w.Flush()

        // Spawn goroutine for the parser reading loop.
        go c.runParserLoop()

        return nil
}
```

Basic client which uses the parser

Client which takes a connection and would be called during the parser loop.

```
type client struct {
        conn net.Conn
        r    *bufio.Reader
        w    *bufio.Writer
}

func (c *client) processInfo(info string) { /* TODO */ }

func (c *client) processMsg(subj string, reply string, sid int, payload []byte) {
        /* TODO */
}

func (c *client) processPing() { /* TODO */ }

func (c *client) processPong() { /* TODO */ }

func (c *client) processErr(msg string) { /* TODO */}
```

Parsing loop → client internal API

```
func (c *client) runParserLoop() {
        for {
                line, _ := c.r.ReadString('\n')
                args := strings.SplitN(line, " ", 2)
                if len(args) < 1 {
                        log.Fatalf("Error: malformed control line")
                }
                switch op {
                case "MSG":
                        // ...
                        c.processMsg(subject, reply, sid, payload)
                case "INFO":
                        c.processInfo(args[1])
                case "PING":
                        c.processPing()
                case "PONG":
                        c.processPong()
                case "+OK":
                        // Do nothing
                case "-ERR":
                        c.processErr(args[1])
                default:
                        log.Fatalf("Error: malformed control line")
                }
        }
}
```

Publish API

```
func (c *client) Publish(subject, reply string, payload []byte) error {
        c.Lock()
        defer c.Unlock()

        pub := fmt.Sprintf("PUB %s %s %d\r\n", subject, reply, len(payload))
        _, err := c.w.WriteString(pub)
        if err != nil {
                return err
        }
        _, err = c.w.Write(payload)
        if err != nil {
                return err
        }
        _, err = c.w.WriteString("\r\n")
        if err != nil {
                return err
        }
        err = c.w.Flush()
        if err != nil {
                return err
        }

        return nil
}
```

Subscribe API / Message Delivery

To support dispatching messages, we need to add a bit more state to the client:

```
type client struct {
        conn net.Conn
        r    *bufio.Reader
        w    *bufio.Writer

        // sid increments monotonically per subscription and it is
        // used to identify a subscription from the client when
        // receiving a message.
        sid int

        // subs maps a subscription identifier to a callback.
        subs map[int]func(subject, reply string, b []byte)

        sync.Mutex
}
```

Subscribe API / Message Delivery

```
func (c *client) Subscribe(subject, queue string,
        cb func(subject, reply string, b []byte),
) error {
        c.Lock()
        defer c.Unlock()
        c.sid += 1
        sid := c.sid

        sub := fmt.Sprintf("SUB %s %s %d\r\n", subject, queue, sid)
        _, err := c.w.WriteString(sub)
        if err != nil {
                return err
        }

        err = c.w.Flush()
        if err != nil {
                return err
        }

        c.subs[sid] = cb

        return nil
}
```

Close API

```
func (c *client) Close() {
        c.Lock()
        defer c.Unlock()

        // Send any pending commands to server previous
        // to closing.
        c.w.Flush()
        c.conn.Close()
}
```

Sample usage of the toy client

Client is roughly 300 lines of code but basic API works!

```
package main

import (
        "log"
        "time"
)

func main() {
        c := &client{}
        err := c.Connect("127.0.0.1:4222")
        if err != nil {
                log.Fatalf("Error: %s", err)
        }
        defer c.Close()
        c.Subscribe("foo", "", func(subject, reply string, data []byte) {
                log.Println("[SUB ] -", subject, reply, "->", string(data))
        })
        for {
                c.Publish("foo", "", []byte("bar"))
                time.Sleep(1 * time.Second)
        }
}
```

Sample usage of the toy client

```
[SUB ] - foo  -> bar
[MSG ] - subject="hello", reply="", payload=world
[SUB ] - hello  -> world
[PONG]
2017-07-04 02:46:58.839492601 -0700 PDT
[MSG ] - subject="foo", reply="", payload=bar
[SUB ] - foo  -> bar
[MSG ] - subject="hello", reply="", payload=world
[SUB ] - hello  -> world
[PONG]
[PING]
2017-07-04 02:46:59.84080511 -0700 PDT
[MSG ] - subject="foo", reply="", payload=bar
[SUB ] - foo  -> bar
[MSG ] - subject="hello", reply="", payload=world
[SUB ] - hello  -> world
[PONG]
2017-07-04 02:47:00.841473462 -0700 PDT
[MSG ] - subject="foo", reply="", payload=bar
[SUB ] - foo  -> bar
[MSG ] - subject="hello", reply="", payload=world
[SUB ] - hello  -> world
[PONG]
[PING]
2017-07-04 02:47:01.842108937 -0700 PDT
[MSG ] - subject="foo", reply="", payload=bar
[SUB ] - foo  -> bar
[MSG ] - subject="hello", reply="", payload=world
[SUB ] - hello  -> world
[PONG]
```

Operational aspects from NATS

Clustering

Full mesh one hop

```
./go/bin/gnatsd -p 4222 -m 8222 --cluster nats://127.0.0.1:6222 --routes
nats://127.0.0.1:6222,nats://127.0.0.1:6223,nats://127.0.0.1:6224
```

```
./go/bin/gnatsd -p 4223 -m 8223 --cluster nats://127.0.0.1:6223 --routes
nats://127.0.0.1:6222,nats://127.0.0.1:6223,nats://127.0.0.1:6224
```

```
./go/bin/gnatsd -p 4224 -m 8224 --cluster nats://127.0.0.1:6224 --routes
nats://127.0.0.1:6222,nats://127.0.0.1:6223,nats://127.0.0.1:6224
```

Monitoring Tools

nats-top

https://github.com/nats-io/nats-top/releases/tag/v0.3.2

Prometheus NATS Exporter

https://github.com/nats-io/prometheus-nats-exporter/tree/master/walkthrough