# Scuttlego

## An implementation of the Secure Scuttlebutt protocol.

29 January 2023

boreq

## Secure Scuttlebutt ecosystem

Clients:

- Patchwork
- Patchfox
- Manyverse
- Planetary

Implementations:

- JavaScript stack
- Go stack

## Running go-ssb on iOS

- stability leaving much to be desired
- problems with memory usage
- problems with performance
- a lot of code outside of go-ssb in the cgo bindings

## Scuttlego

A new Secure Scuttlebutt implementation written in Go.

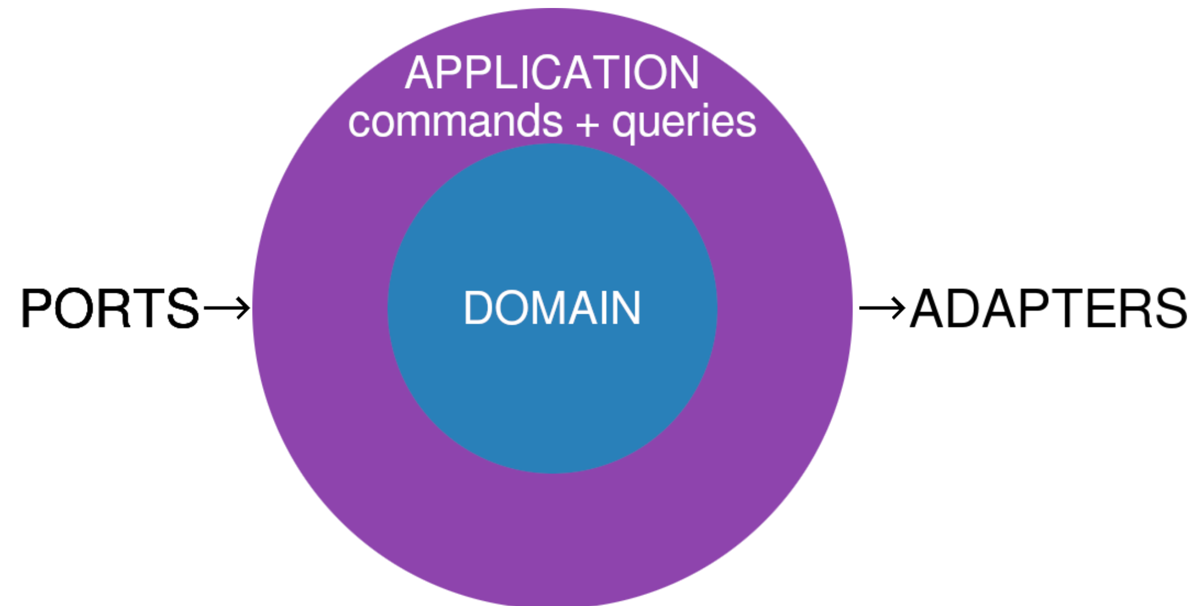
Reuses some elements of go-ssb:

- the handshake mechanism
- the box stream protocol
- the verification and signing of messages
- broadcasting and receiving local UDP advertisements

# Key concepts

- Hexagonal architecture
- Domain driven design
- Command and query separation

# Hexagonal architecture




- Domain
- Application
- Ports
- Adapters

## Domain Driven Design

> "The structure and language of software code should match the business domain."

- strong types
- state in memory always correct by using constructors
- immutable structs if possible

# Domain

```
type Message struct {
    Id       string
    Sequence int
}

func DoSomething(msg Message) error {
    if msg.Id == "" {
        return Message{}, errors.New("empty id")
    }

    if msg.Sequence <= 0 {
        return Message{}, errors.New("sequence must be positive")
    }

    // do things

    return nil
}
```

# Domain

```
type Message struct {
    id       string
    sequence int
}

func NewMessage(id string, sequence int) Message {
    return Message{
        id: id,
        sequence: sequence,
    }
}

func (msg Message) Id() string {
    return msg.id
}

func (msg Message) Sequence() int {
    return msg.sequence
}
```

# Domain

```
type Message struct {
    id       string
    sequence int
}

func NewMessage(id string, sequence int) (Message, error) {
    if id == "" {
        return Message{}, errors.New("empty id")
    }

    if sequence <= 0 {
        return Message{}, errors.New("sequence must be positive")
    }

    return Message{
        id: id,
        sequence: sequence,
    }, nil
}

func (msg Message) Id() string { return msg.id }

func (msg Message) Sequence() int { return msg.sequence }
```

# Domain

```
type Message struct {
    id       Id
    sequence Sequence
}

func NewMessage(id Id, sequence Sequence) Message {
    return Message{
        id: id,
        sequence: sequence,
    }
}

func (msg Message) Id() Id { return msg.id }

func (msg Message) Sequence() Sequence { return msg.sequence }
```

# Domain

```
type Id struct {
    id string
}

func NewId(id string) (Id, error) {
    if id == "" {
        return Id{}, errors.New("empty id")
    }

    return Id{
        id: id,
    }, nil
}
```

# Domain

```
type Sequence struct {
    sequence int
}

func NewSequence(sequence int) (Sequence, error) {
    if sequence <= 0 {
        return Sequence{}, errors.New("sequence must be positive")
    }

    return Sequence{
        sequence: sequence,
    }, nil
}
```

# Domain

```
type Message struct {
    id       Id
    sequence Sequence
}

func NewMessage(id Id, sequence Sequence) (Message, error) {
    if id.IsZero() {
        return Message{}, errors.New("zero value of id")
    }

    if sequence.IsZero() {
        return Message{}, errors.New("zero value of sequence")
    }

    return Message{
        id: id,
        sequence: sequence,
    }, nil
}

func (id Id) IsZero() bool { return id == Id{} }

func (seq Sequence) IsZero() bool { return seq == Sequence{} }
```

# Domain

```
type Message struct {
  // ...
}

type Feed struct {
  Messages []Message
}

func AddToFeed(feed Feed, message Message) error {
    // validate feed
    // validate message
}
```

# Domain

```
type Message struct {
  // ...
}

type Feed struct {
  messages []Message
}

func (f *Feed) AddToFeed(message Message) error {
    if len(f.messages) > 0 {
        // ...

        if !f.lastMsg().ComesDirectlyBefore(message) {
            return errors.New("this is not the next message in this feed")
        }
    } else {
        if !message.IsRootMessage() {
            return errors.New("first message in the feed must be a root message")
        }
    }

    f.messages = append(f.messages, message)
    return nil
}
```

# Commands and queries

```go
type AppendMessage struct {
    msg Message
}

type UpdateFeedFn func(f *domain.Feed) error

type FeedRepository interface {
    UpdateFeed(id domain.FeedRef, fn UpdateFeedFn) error
}

type AppendMessageHandler struct {
    repository FeedRepository
}

func NewAppendMessageHandler(repository FeedRepository) AppendMessageHandler {
    return AppendMessageHandler{repository: repository}
}

func (h AppendMessageHandler) Handle(cmd AppendMessage) error {
    return h.repository.UpdateFeed(cmd.msg.Feed(), func(f *domain.Feed) error {
        return f.AppendMessage(cmd.msg)
    })
}
```

# Commands and queries

## In application layer:

```
type Commands struct {
  AppendMessage *commands.AppendMessageHandler
}

type Queries struct {
  GetMessage *queries.GetMessageHandler
}

type Application struct {
    Commands Commands
    Queries  Queries
}
```

## Outside of the application layer e.g. in ports:

```
func (h HTTPHandler) DoSomething(...) error {
    cmd := commands.NewAppendMessage(...)
    return h.app.Commands.AppendMessage.Handle(cmd)
}
```

# Replacing the database layer completely

Initially bbolt seemed like a good option.

Problem:

```
mmap allocate error: cannot allocate memory

// ...
b, err := unix.Mmap(int(db.file.Fd()), 0, sz, syscall.PROT_READ, syscall.MAP_SHARED|db.MmapFlags)
// ...
```

Solution:

## Tests

Well-tested domain layer prevents a lot of bugs and allows you to avoid writing component tests for anything other than complex behaviours.

Using `github.com/stretchr/testify` is a good idea.

Table tests are a good idea.

Test fixtures e.g. `SomeProcedureName`, `SomeBool`, `SomeDirectory` are useful.

## Performance

Performance tailored for mobile devices:

- lower memory usage with smaller spikes
- lower CPU usage when idling and not doing anything useful (hot phone syndrome)

Noticable performance improvements when using the app mostly due to:

- avoiding blocking (retrieving stats, retrieving blobs)
- not "getting stuck" when replicating

## Source code

https://github.com/planetary-social/scuttlego (https://github.com/planetary-social/scuttlego)

MIT licensed.

# Thank you

boreq

https://planetary.social (https://planetary.social)
https://github.com/planetary-social/ (https://github.com/planetary-social/)

https://0x46.net (https://0x46.net)