

DIPLOMARBEIT

# Programming Reactive Systems in Go



FernUniversität in Hagen

Erstellt und präsentiert von STEFAN HANS

Matrikelnummer: Q4918851

FAKULTÄT FÜR MATHEMATIK UND INFORMATIK

Lehrgebiet PROGRAMMIERSYSTEME

Betreut von DR. DANIELA KELLER

Abgegeben am 17.09.2019

This thesis wants to explore a way to design, implement, and test decentralized systems, and, furthermore, to create a prototype of a development system to do so. We will measure the accomplishment against the principles of the Reactive Manifesto [Jon14]. We will get an overview of relevant theories and principles along with explanations of parts of the Go language and selected libraries. After some thoughts about the design approach, we work on prototypes following the methodology of Creative Learning [Mit07].

The reached software embodies two aspects - an example of a decentralized and reactive application, and the prototype of a development system to program the same.

*"Make everything as simple as possible, but not simpler."*

Quote by ALBERT EINSTEIN

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Theory</b>	<b>10</b>
2.1	Reactive Paradigms . . . . .	11
2.1.1	Message Driven . . . . .	11
2.1.2	Elastic . . . . .	11
2.1.3	Resilient . . . . .	12
2.1.4	Responsive . . . . .	12
2.2	Architecture Of Distributed Systems . . . . .	12
2.2.1	Aspects Of Distinction . . . . .	12
2.2.2	Infrastructural Aspects . . . . .	13
2.2.3	Sharing Architectures . . . . .	14
2.2.4	Client-Server . . . . .	14
2.2.5	Cloud . . . . .	15
2.2.6	Services And Microservices . . . . .	15
2.2.7	Peer-To-Peer . . . . .	18
2.3	Kind Of Components . . . . .	18
2.3.1	Clients . . . . .	18
2.3.2	Services . . . . .	19
2.3.3	Container . . . . .	19
2.4	Composition Of Functionality . . . . .	19
2.4.1	Layer . . . . .	19
2.4.2	Service . . . . .	19
2.4.3	Libraries . . . . .	19
2.5	Conclusion . . . . .	20
<b>3</b>	<b>Relevant Features and Libraries in Go</b>	<b>21</b>
3.1	Language Specification . . . . .	21
3.1.1	Slices, Maps, and Structs . . . . .	21
3.1.2	Functions, Methods, and Interfaces . . . . .	24
3.1.3	Goroutines, Channels, and Select Statements . . . . .	34
3.2	Standard Library . . . . .	40
3.2.1	Network Packages . . . . .	40
3.2.2	Context Package . . . . .	43
3.2.3	JSON Package . . . . .	45
3.2.4	Testing Package . . . . .	49

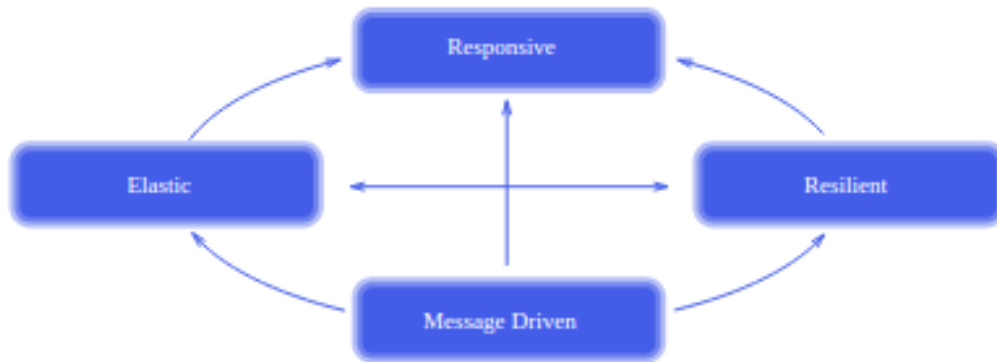
3.3	Libraries for Distributed Systems . . . . .	50
3.3.1	HashiCorp's 'memberlist' . . . . .	50
3.3.2	Protocol Lab's 'go-libp2p-kad-dht' . . . . .	51
3.4	Other Libraries . . . . .	54
3.4.1	Protocol Buffer and gRPC . . . . .	54
3.4.2	TUI . . . . .	61
3.4.3	Liner . . . . .	64
<b>4</b>	<b>Designing Reactive Systems</b>	<b>66</b>
4.1	Approach . . . . .	66
4.1.1	Chat as an example application . . . . .	67
4.1.2	Development system prototype . . . . .	67
4.2	Requirements . . . . .	67
4.2.1	Requirements of the chat application . . . . .	67
4.2.2	Requirements of the development system . . . . .	68
4.3	Methodical Proceedings . . . . .	69
4.3.1	Complete information in an idealized world . . . . .	69
4.3.2	Incomplete information in an idealized world . . . . .	70
4.3.3	Incomplete information in a realistic world . . . . .	70
<b>5</b>	<b>Programmed Prototypes</b>	<b>71</b>
5.1	Chat With Protocol Buffers And gRPC . . . . .	72
5.1.1	Design . . . . .	72
5.1.2	Implementation . . . . .	72
5.2	Chat With Protocol Buffers Via TCP Or UDP . . . . .	74
5.2.1	Design . . . . .	74
5.2.2	Implementation . . . . .	74
5.3	Chat Using Cloud Functions As List Of Members Service . . . . .	77
5.3.1	Design . . . . .	77
5.3.2	Implementations . . . . .	78
5.3.3	Testing . . . . .	80
5.4	Pre-Chat Using 'memberlist' . . . . .	83
5.4.1	Design . . . . .	83
5.4.2	Implementation . . . . .	84
5.5	Pre-Chat Using 'libp2p' . . . . .	89
5.6	Findings . . . . .	89
5.6.1	Simplicity . . . . .	89
5.6.2	Services . . . . .	90
5.6.3	Command Line . . . . .	91
<b>6</b>	<b>The Final System Prototype</b>	<b>92</b>
6.1	Design . . . . .	94
6.1.1	Bootstrap Service . . . . .	94
6.1.2	Distributed Group Membership Protocol Layer . . . . .	95

6.1.3	Integrated Command Line Interpreter . . . . .	96
6.1.4	The Chat As Example Application . . . . .	96
6.1.5	Multi-Client Testing . . . . .	97
6.2	Implementation . . . . .	97
6.2.1	Bootstrap API . . . . .	97
6.2.2	Bootstrap Server . . . . .	98
6.2.3	Bootstrap Cloud Functions . . . . .	98
6.2.4	'memberlist' Integration . . . . .	99
6.2.5	Terminal UI . . . . .	99
6.2.6	Chat . . . . .	99
6.2.7	Command Line . . . . .	99
6.2.8	Test API . . . . .	99
6.2.9	Test Server . . . . .	100
6.3	Testing . . . . .	100
6.4	Status . . . . .	102
<b>7</b>	<b>Summary</b>	<b>103</b>
7.1	Creative Learning . . . . .	103
7.2	Chat As An Example Application . . . . .	104
7.3	Development System Prototype . . . . .	104
7.4	Prove It As Reactive System . . . . .	104
7.4.1	Message Driven . . . . .	104
7.4.2	Elastic . . . . .	105
7.4.3	Resilient . . . . .	105
7.4.4	Responsive . . . . .	105
7.5	Outlook . . . . .	105
	<b>Books &amp; Articles</b>	<b>107</b>
	<b>Websites</b>	<b>109</b>
	<b>Repositories</b>	<b>113</b>
	<b>Code</b>	<b>114</b>
	<b>Acronyms</b>	<b>119</b>

# 1 Introduction

During the last decade, the rise of disruptive, cutting-edge technologies was shaking up the developer communities in the field of computer science. The top dogs among the technology leaders were using their own developments to cope with massive data streams and high customer expectations about responsiveness. To be able always to respond in an acceptable time, reactive systems react to various changes in their surroundings. Different demands like changing numbers of client requests or changing requirements regarding the amount of data to process need a system which can scale itself accordingly. The system has to be resilient because computer and networks are not reliable. To achieve resilience, we need a non-blocking communication over the network driven by asynchronous message transfers. *Responsive*, *Resilient*, *Elastic*, and *Message Driven* are the four aspects we need to accomplish to have a reactive system. These systems are described concisely as *Reactive Systems* in *The Reactive Manifesto* [Jon14].

Figure 1.1: Reactive Manifesto [Jon14]



Systems built as *Reactive Systems* are traditionally acting as a server providing functionality to other clients. Nowadays, these systems are known as cloud and offered as a service (aaS), and the clients are named the edge.

At the same time, the price reduction of edge devices leads to another major topic, IoT. But the expected scenarios in IoT seem to be only partially manageable by the approach of current cloud technology. Mainly due to two reasons, the further increase of the data magnitudes and the demands of time critical applications. As an obvious consequence,

the top architecture has to be transformed once again from a more centralized approach to a decentralized and distributed one. In this regard, the rising of distributed hash table based technologies like blockchain brought the peer-to-peer approach back into focus. The idea is to take container orchestration, a platform for creating platforms approach, out in the wild of the internet in a peer-to-peer fashion.

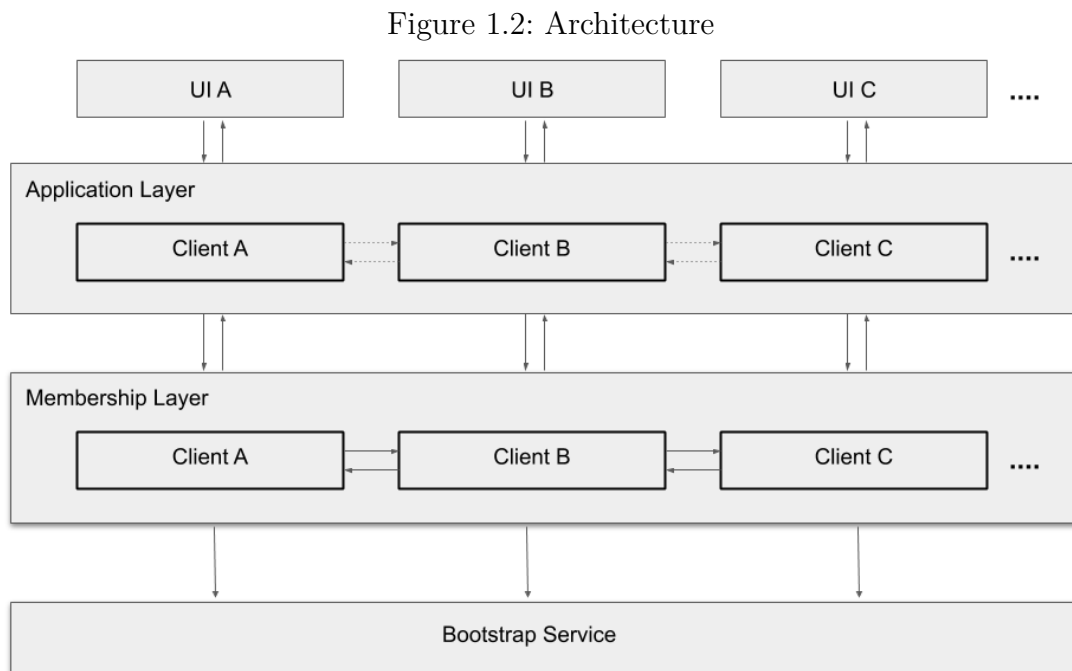
You can see this from two perspectives:

- shift the cloud from datacenters to the internet
- generalize the peer-to-peer approach

We want neither deeply analyse the cloud nor the peer-to-peer technology. Instead, we will follow a from scratch approach and develop prototypes. From prototype to prototype, we develop towards a system of applications to fulfilling the properties as proclaimed in the Reactive Manifesto:

- responsive
- elastic
- resilient
- message driven

Here, we see a figure of the architecture to achieve this reactive system.



In the core of the architecture are two layers. The *Application Layer* is for the communication directly related to the application purpose. The *Memberlist Layer* has the infrastructural task to keep the information about the network of peers up to date. Every client is part of both layers, or, in other words, the same set of clients is building both layers. The Bootstrap Service makes the network accessible for new peers.

The thesis will attempt to prove the following hypothesis:

*A set of applications connected over a network can act as a Reactive System to the user. Every application is playing two roles:*

- *listening for requests to provide or receive data, like a server*
- *sending or requesting data interacting with the user, like a client*

*In contrast to classic client-server models, the server functionality gets distributed over the applications being the Reactive System. This approach does not need a hierarchical structure. Nevertheless, a minimal bootstrap-like feature is required - the **bootstrap service***

*Who is an available member of the system and how to connect to him, is the question to be answered in and from the **membership layer**.*

*Then, the applications can use the provided information to work together with the other clients as needed by its purpose. The communication on the **application layer** can be within the layer or via the membership layer.*

*Above these layers, we have the user interfaces - **UI** [Eri04].*

***It will be proven that this system is responsive, elastic, resilient, and message driven - in one word, reactive.***

After the introduction, we will give an overview of relevant theoretical concepts and paradigms. The next part is about the programming language Go and some libraries we use later. Concise code samples show features and concepts we want to highlight. Then, we continue with general explanations about the design of the system and how to implement the prototypes. Presenting the prototypes is next and leads to the final system prototype to be discussed in detail. In the end, we summarize the results and experiences, and we give an outlook.



To complete the introduction, some words about the choice for Go as the programming language:

*"Go is a general-purpose language designed with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming."* [Gol]

Go's concurrent programming features are goroutines, channels, and the select statement. They are following the famous quote *"Don't communicate by sharing memory; share memory by communicating."* [And10] and the minimalistic approach of Go in general.

If you recognize, what the programming languages are for the reference implementations of the cutting-edge software in the fields of cloud [Cnc] and peer-to-peer, then the decision for Go as a programming language is evident. Here just some of the most important ones:

- Docker [Doc]
- Kubernetes [K8sa]
- Ethereum [Eth]
- IPFS [Ipf]
- HashiCorp's main software [Has]

To give you a short impression, here is a "Hello World" example of a Go program:

Listing 1.1: "main.go"

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello World!")
7 }
```

Listing 1.2: Compile and run "main.go"

```
1 $ go run main.go
2 Hello World
```

## 2 Theory

*"Controlling complexity is the essence of computer programming."*

Quote by BRIAN KERNIGHAN [Bri81]

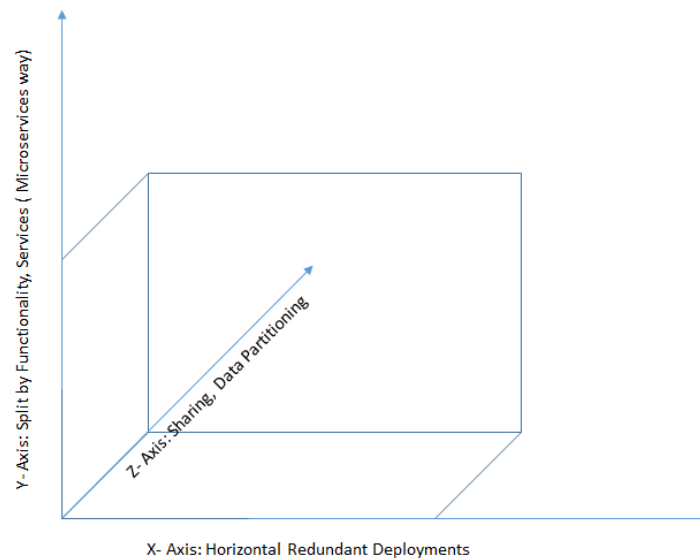
The probably most important paradigm in our context is the idea to share information between processes not by common data but by communicating.

To make this more understandable imagine people working in an open-plan office which has an extensive filing system in its center. This corresponds to monolithic systems having one database for data needed by all processes as well as to the concept of shared memory used by process threads.

Without one central filing system, people do have to communicate to get information directly from their colleagues who provide specific services and information, respectively. This scenario is being realized at various scales: By the microservice architecture [Jam14] and within cloud orchestrations at a large scale and inside single Go executables by implementing Hoare's Communicating Sequential Processes (CSP) [Hoa04] paradigm.

Nevertheless, persistent and consistent data are still needed and shared but in a much more decentralized and flexible way. This decentralized approach gives us the scalability to create reactive systems. Scale cube as a model shows the main aspects of scalability [Mar15b].

Figure 2.1: The Scale Cube [The]



We can scale by functional decomposition, an indirect way which allows adapting implementation and resources as needed. In the open-plan office, this would mean a perfectly tailored remit for every worker.

Due to having communication in the center of the system we can now scale by the deployment of instances with redundant functionality - hiring new employees.

If the amount of persistent data is a performance problem, we can partition data horizontally to spread the load. This data-oriented decomposition makes it possible to add new instances as well.

After this kind of introductory overview, let's dive into more details, now.

## 2.1 Reactive Paradigms

*"Message Driven is the groundwork of a Reactive System."*

*The Reactive Manifesto [Jon14]*

As the main goal, Reactive Systems provide the clients with responsiveness. To achieve this they react on load changes with elasticity and on failure with resilience. This is possible by sending messages asynchronously between components of the system to orchestrate and choreograph, respectively.

### 2.1.1 Message Driven

In a system of components connected over a network, the communication takes place by exchange of data. But, network connections, as well as computers, are not reliable. The system is not reactive if it cannot handle unreliability and errors. The mentioned communication between components tries to ensure this by the following means:

- isolation - a component is not directly affected by problems of other components
- location transparency - the communication relies not on the location of the components
- loose coupling - components are exchangeable in principle

Crucially for a component is to accept that others do not react at all, react late, or react by sending an error. Additionally, a component tries to be always reachable for other components. We call such a system message-driven [Jon14].

### 2.1.2 Elastic

Elasticity is the ability to scale, precisely to delegate tasks to other participants of the system. Because of the lack of hierarchy, no start or shutdown can be triggered from others. Instead, a standby of applications is published waiting to take over tasks on request. The system can exchange measurements about the status of its components and react to it:

- send information to update
- send data
- send tasks to undertake
- other reactions, e.g., save data locally

To be elastic is a necessity of Reactive Systems [Jon14].

### 2.1.3 Resilient

Resilience is the ability to handle failures and even outages of the system's participants transparently. Other applications on standby will be triggered to fill in accordingly. Therefore a heartbeat-like mechanism keeps the system up to date. We need resilience; otherwise, the whole system could be unresponsive after a failure. A component needs to survive the failure of others, and it must be able to come back into the system after it had a failure itself. A Reactive System stays responsive in the face of failure [Jon14].

### 2.1.4 Responsive

Responsiveness is the final goal. It is vital to users' acceptance, and to a system able to react quickly and effectively to various situations like problems. Then, A Reactive System will deliver a consistent quality of service [Jon14].

## 2.2 Architecture Of Distributed Systems

*"... methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently ..."*

Quote by ROY THOMAS FIELDING [Roy00]

We have to decide first what are the possible architectural patterns concerning the needs of the whole software system.

### 2.2.1 Aspects Of Distinction

Before we see these patterns, we should introduce three aspects of distinction:

## Stateless and Stateful

*Stateless* means there is no memory of the past. Like a mathematical function, the same input always yields the same output. *Stateful* means that there is memory of the past and therefore persistent data is needed. The output depends not only on the input. Stateful reactive systems cannot guarantee responsiveness and consistency under all circumstances [Set02].

## Application and Infrastructure

A universal definition without referring to each other is complicated. But an intuitive understanding should be enough for the time being. In a language-agnostic way, we could say, the *infrastructure* should provide what the application has in common with other applications, and the *application* should implement what it has different. According to the aimed architecture [1.2], the bootstrap service is usable for all applications, and therefore, it is part of the infrastructure. The UI and the application layer is entirely part of the application. The membership layer is part of the infrastructure, but it is also possible to use it piggybacking for application purposes [Abh].

## Orchestration and Choreography

*Orchestration* needs an orchestrator to coordinate the action between the components. *Choreography* is an instruction set how the components should interact. Due to orchestrator as a single point of failure, the rule of thumb is as much choreography as possible and so much orchestration as needed. Additionally the orchestrator has to be resilient itself, i.e., has to be elected from among the others if needed. That is not trivial and causes the leader election problem [Gur92].

## 2.2.2 Infrastructural Aspects

We see three groups of infrastructural aspects, which share obviously most of the distributed systems in one way or the other:

### Processing

Modern distributed systems are resilient and elastic, i.e., they can handle failure and do scale horizontally. This behavior needs orchestration or choreography, respectively. The processing requirements, i.e., a required minimum and a sufficient maximum, should be coordinated with the available computational resources.

### Network

The network has not only the task to ensure communication between the components but also to manage meta information and to control the traffic.

## Storage

Knowing that failure can destroy the availability of resources, e.g., virtual or physical hosts, we may need storage which survives these scenarios and provides persistent data for new resources.

### 2.2.3 Sharing Architectures

The considerations and decisions about how and what data should be shared and accessed by different services and service instances is a significant design task and will be discussed in this section.

#### Shared Nothing Architecture

*"... shared nothing (SN), i.e. neither memory nor peripheral storage is shared among processors"*

Quote by MICHAEL STONEBRAKER [Mic86]

Analogously to a multiprocessor system architecture, a shared-nothing architecture (SN) in the domain of distributed computing contains nodes which are neither sharing disks nor memory. Nodes do have memory and disk storage, but they do not access others' data.

#### Shared Disk Architecture

In many systems, we need some consensus about the data shared between its nodes. We can decouple the storage from the processing nodes and share the data centrally, e.g., using a service. Concerning clusters using containers on different hosts, we can share disk storage per host or container [K8sb].

#### Space-Based or Cloud Architecture

This architecture consists of horizontally scalable service instances which hold processing capacities, a copy of the relevant data in memory, a replication engine, and optionally an asynchronous persistent store for failover. Additionally, an orchestrator is needed for coordinating the data replication [Mar15a].

### 2.2.4 Client-Server

In this classical, or old-fashioned, if you wish, paradigm, we see a monolith as a server interacting with clients and holding all that the clients need to know. Mainly, the clients are presenting and interacting with the user and the server. Therefore, the server is a single point of failure and the crucial part regarding scalability, with all the essential problems. We should avoid these limitations. Another difficulty of large server is maintainability. It is often too complex to handle it efficiently.

## 2.2.5 Cloud

A first approach to overcome the problems of a monolithic server was to introduce redundancy by having a cluster with more than one machine. But it is expensive to provide extra hardware for the case of failure or extreme load on a per server basis. Why not sharing resources more flexible? From virtual machines to container technology like Docker and orchestration platforms like Kubernetes, we can see how the flexibility happened.

## 2.2.6 Services And Microservices

*"The reason for calling it a Microservice [...] is that it has a small number of responsibilities: one. It does one thing, and does it well."*

Quote by VIKTOR KLANG [Vkl]

In practice, it is not always easy to decide what exactly is this one responsibility, this one thing, which makes a microservices. We will use the term microservice to emphasize the one responsibility aspect.

### Domain-Driven Design

Closely related to microservices is Domain-Driven Design and Bounded Context.

*"Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's <sup>1</sup>strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships."*

Quote by MARTIN FOWLER [Fow14]

Defining Bounded Context is the art of dividing a bigger application into microservices by making aware the context of the service to be determined.

One kind of functional decomposition of classical monoliths is into services, known as microservices. They have their own bounded contexts and give developers and operators a high grade of freedom. Microservices need an infrastructure to operate on and APIs to communicate with each other. We need a network functionality to route requests from external clients to the appropriate microservices. It can be done by one controller aka API gateway for the whole microservices application, one per host, or one per container. These approaches lead from orchestration to choreography [Mic18].

The controller has some tasks to fulfill [Lee18]:

---

<sup>1</sup>Domain-Driven Design [Dom]

## Traffic Control

It acts as an ingress controller to limit access to the cluster network via IP addresses and port numbers. It provides services discovery, i.e., it has to know how to connect to the services. Service discovery is needed as well for inter-service communication, i.e., internally between services.

## Performance Optimization

It uses load balancing and caching to optimize the performance, and it tries to minimize communication start-time by using health checks and keepalive functionality. If a component gets under stress, it should initialize a back-pressure, i.e., inform the controller about its diminished availability.

## Security

It should provide advanced SSL/TLS<sup>2</sup> features and access control like RBAC<sup>3</sup>. Additionally, the controller could implement request limiting as a mechanism against DDoS attacks<sup>4</sup>.

## Distribution and Aggregation

If needed, it distributes the requests to different services and parallel executions and aggregates the returned results accordingly.

## **API-Driven Development And How To Communicate Messages**

API-driven development starts with designing the APIs for the communication between the bounded contexts and its services. Then, we analyze the information to be communicated and how communication has to be organized.

We use two dimensions to categorize interaction via messages:

### One-to-one / One-to-many

A one-to-one interaction means a message has one sender and one recipient. On the other hand, one-to-many means one sender and possibly more than one recipient.

---

<sup>2</sup>SSL/TLS "Secure Sockets Layer/Transport Layer Security" [Rfcc]

<sup>3</sup>RBAC "Role-Based Access Control" [Fer92]

<sup>4</sup>DDoS attack "Distributed Denial-of-Service attack"



## Synchronous / Asynchronous

Synchronous interaction means the sender expects a timely response and waits possibly blocking. Asynchronous interaction decouples request and potentially response.

In these two-dimensional space you consider the following:

### Notification

A notification is merely a one-way message with no response expected.

### Request-response

A message is sent as a request to a recipient, and the sender waits for a response synchronously, i.e., keeps the connection open for the expected message. Maybe the sender's processing is not blocked, the connection is it. Alternatively, the connection is closed, but the sender is listening for a response asynchronously.

### Publish-subscribe

This one-to-many messaging pattern needs a sender which provides a registration service for its subscribers to publish messages accordingly. The interaction can be done in a synchronous or asynchronous way.

### Message queue

A message queue implements an asynchronous communication protocol. Sender and receiver do not need to communicate at the same time. A received message is stored until the recipient retrieves it. A message queue has limits concerning the size of the messages and the size of the queue.

## **As A Service**

If we are using a service from public cloud providers, we gain professionalism and convenience they offer and lose flexibility and control. There are many categories of services named 'as a service' - I want to emphasize the following:

### IaaS - Infrastructure as a service

We can create virtual machines, network, storage, and more as a service to have an infrastructure for our needs.

### PaaS - Platform as a service

We can have a complete platform with an orchestrator of resources like Kubernetes [Gke][Eks][Aks]. As a requirement, the applications have to be images to run in containers. Additionally, the platform provides a set of professional services for administration, support, CI/CD <sup>5</sup>, and others.

### FaaS - Function as a service

We can have only the runtime environment for our functions or microservices, respectively, as a service. These services are only available for specific programming languages and not all yet battle-proven <sup>6</sup>.

Finally, you can use services for a single predefined purpose. And you can combine all in various ways.

## 2.2.7 Peer-To-Peer

*"P2P computing fits naturally to this new era of userdriven, distributed applications utilizing resource-rich edge devices."*

Quote by H. M. N. DILUM BANDARA AND ANURA P. JAYASUMANA [H. 12]

The most complex system to develop is a P2P system where the distributed system runs on the devices of the end user. But, from the perspective of costs, scaling, resilience, and data privacy, it can have a unique selling proposition. Here we see an extension or even competition to the cloud computing as provided by the public cloud provider, especially. Aside from having other paradigms and methodologies, we have other terms and names, and problems [H. 12]. We cannot dissolve the client-server pattern completely, but as much as possible.

## 2.3 Kind Of Components

I want to pinpoint three essential kinds of components we see in reactive systems:

### 2.3.1 Clients

Client applications can run in the browser, as rich application, on mobile or embedded devices. Their remit varies from data collecting sensors over classical frontend apps until application clients, which include backend functionality. We will focus on client applications with backend functionality.

---

<sup>5</sup>CI/CD "Continuous Integration/Continuous Delivery" or "Continuous Integration/Continuous Deployment" [Steb]

<sup>6</sup>I did participate in the Cloud Functions for Go alpha testing, recently

### 2.3.2 Services

Centralized services are a core component we cannot avoid altogether. And we should not try to, anyway. But, a service should always follow the microservice pattern and do one thing well. We will need a bootstrap service [1.2] and introduce a service to coordinate multiple-client testing [6.1.5].

### 2.3.3 Container

If we need some functionality, which we want centralized, and our requirements, e.g. of performance, are not achievable by microservices, we have to think about using containers. Then, we can operate with images of software working together directly in the same environment and without the disadvantages of network connections. We will not use containers. Instead, we will use Cloud Functions [Gcf] to operate the bootstrap service.

## 2.4 Composition Of Functionality

The decomposition of the functionality of a monolith or how to design from scratch is crucial, and we have to differentiate. Here some distinctions we can take into account:

### 2.4.1 Layer

If we recognize a one-way dependency, we should see the components as layers. The lower layer provides functionality for the layer above. It does not know the layer above. It always has a network aspect, or you should consider it as a library. If it is the only one providing the feature, it is more likely a service.

### 2.4.2 Service

A service communicates over the network in a kind of centralized manner. While designing services, we should think about whether and how to divide or join [Fow14].

### 2.4.3 Libraries

If we can identify a functionality which can be used by others or replaced by another implementation, you can separate it as a library - but be aware of cyclic dependencies. In case of loose coupling, like an API, you can start implementing it as a library from the beginning.

## 2.5 Conclusion

*"Simplicity is complicated but the clarity is worth the fight."*

Quote by ROB PIKE [ROB]

The theoretical considerations are manifold, confusing, and sometimes contradictory. Not to lose motivation, we should avoid too complicated theories without concrete, practical relevance. This advice is more about the scholar, and not about the theories itself. If you are not comfortable with more straightforward ideas, you will lose your intuition dealing with more complex one - if you understand them at all. It is all about reducing complexity, and this is hard.

Thus, we want to have the most minimalistic and most generalized design, implementation, and testing possible for the system, and we decide to go for the following:

- as stateless as possible and as stateful as needed
- a clear separation between infrastructure and application
- as much choreography as possible and as much orchestration as needed
- fade out the infrastructural aspects if possible
- share as less as possible
- overcome client-server paradigm
- a restricted set of microservices
- deploy microservices on a serverless environment of a public cloud provider if needed
- focus on a production-ready library for peer-to-peer

We will learn creatively [4.1] and gain experience by creating prototypes [5]. Finally, we want to get a profound understanding of the applied theories and the possible ways forward towards the creation of more specialized applications.

## 3 Relevant Features and Libraries in Go

In this chapter we explain some essential parts we need for our prototypes and for the final software:

1. core elements of Go [Gol]
2. standard libraries of Go [Gosa]
3. other external libraries

All listings and code examples are self-created if not mentioned otherwise.

### 3.1 Language Specification

#### 3.1.1 Slices, Maps, and Structs

Slices, Maps, and Structs - these three types are the main constructs for collections of data in the language Go.

##### Slices as segments of dynamic arrays

*"A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Gosg]

Slices are like dynamic arrays. A slice administrates an underlying array internally. A slice has an element type, a length, i.e., the number of elements, and capacity, i.e., the length of the underlying array. Slices having an *empty interface* as element type can store elements of different types. After declaration, you can use the built-in function *append* to add new elements. If you initialize a slice with the built-in function *make* specifying the length and optionally the capacity, the elements have the default value of its type. Other built-in functions are *len* and *cap* showing the length and capacity, respectively.

The following example shows how you address elements of a slice - directly by index, as a range from one index to another inclusively or with no index from the beginning or until the end, respectively.

Listing 3.1: "slices.go"

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     sliceOfDigits := []string{
7         "zero", "one", "two", "three",
8         "four", "five", "six", "seven",
9         "eight", "nine", "ten"}
10
11     fmt.Printf("sliceOfDigits[0] : \t%\v\n", sliceOfDigits[0])
12     fmt.Printf("sliceOfDigits[1:6]: \t%\v\n", sliceOfDigits[1:6])
13     fmt.Printf("sliceOfDigits[6: ]: \t%\v\n", sliceOfDigits[6:])
14     fmt.Printf("sliceOfDigits[ :4]: \t%\v\n", sliceOfDigits[:4])
15 }

```

Listing 3.2: Compile and run "slices.go"

```

1 $ go run slices.go
2 sliceOfDigits[0] :      zero
3 sliceOfDigits[1:6]:    [one two three four five]
4 sliceOfDigits[6: ]:    [six seven eight nine ten]
5 sliceOfDigits[ :4]:    [zero one two three]

```

Although array is an available type in Go, it is recommended to use slices instead in nearly almost cases [Gob].

## Maps as key/value stores

*"A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Goma]

The key and the element types can be any type, including the *empty interface*. Maps having an *empty interface* as element type can store elements of different types.

The following example creates a map having an index as key and the English word of it as value. It shows how you assign the key-value pairs by looping over an anonymous slice of strings. If you only declare a map you have a nil map, i.e., the initial value is a nil pointer. To initialize a usable map, you use the built-in function *make*. Other built-in functions in this regard are *len* and *delete*.

Listing 3.3: "maps.go"

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     mapOfDigits := make(map[int]string)
7
8     for i, word := range []string{
9         "zero", "one", "two", "three",
10        "four", "five", "six", "seven",
11        "eight", "nine", "ten"} {
12        mapOfDigits[i] = word
13    }
14
15    delete(mapOfDigits, 10)
16
17    fmt.Printf("mapOfDigits has %d elements: %v\n", len(mapOfDigits), mapOfDigits)
18 }

```

Listing 3.4: "Compile and run "maps.go"

```

1 $ go run maps.go
2 $ mapOfDigits has 10 elements: map[4:four 8:eight 6:six 7:seven 9:nine 0:zero 1:one
3 2:two 3:three 5:five]

```

## Structs as basic structures to organize entities

*"A struct is a sequence of named elements, called fields, each of which has a name and a type."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Gosh]

The core use of a struct is to build a hierarchy of elements, so-called fields, which can be itself structs. Fields are exported outside of the package implicitly by the identifier. An identifiers starting with an uppercase letter, is an exported field. Otherwise, it is not visible outside of the package. You can use fields without names, so-called *embedded* fields. The name of the type is used as a field identifier, then. If an embedded field is a struct with unambiguous fields, we can call them without the type identifier. We call such a fields promoted.

The following example creates a struct with two fields. The fields can be accessed by the using a dot syntax.

Listing 3.5: "structs.go"

```
1 package main
2
3 import "fmt"
4
5 type rectangle struct {
6     width int
7     height int
8 }
9
10 func main() {
11     oneRectangle := rectangle{
12         width: 8,
13         height: 12,
14     }
15
16     fmt.Printf("oneRectangle: %+v\n", oneRectangle)
17     fmt.Printf("oneRectangle.width: %v, "+
18         "oneRectangle.height: %v\n",
19         oneRectangle.width, oneRectangle.height)
20 }
```

Listing 3.6: Compile and run "structs.go"

```
1 $ go run structs.go
2 oneRectangle: {width:8 height:12}
3 oneRectangle.width: 8, oneRectangle.height: 12
```

### 3.1.2 Functions, Methods, and Interfaces

Functions, Methods, and Interfaces - these three types are essential when it comes to processing.



## Functions

*"A function type denotes the set of all functions with the same parameter and result types."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Gof]

Functions are first-class citizen and do have various manifestations as named functions, anonymous functions, and closures. They support multiple return values and can be variadic, but do not accept default parameter values.

The following example shows a function which has a name, two parameters, and two return values. The second parameter, the dividends, is variadic, i.e., it can be none, one, or many of its type all represented in a slice. The second parameter is the divisor for all dividends, and the function returns the sum of the divisions and an error which is not nil if the divisor is zero.

Listing 3.7: 'named-functions.go'

```
1 package main
2
3 import (
4     "fmt"
5     "errors"
6 )
7
8 // named variadic function with multiple return values
9 func divide(divisor int, dividends ...int) (float64, error) {
10     if divisor == 0 {
11         return 0, errors.New("Division by zero")
12     } else {
13         dividend := 0
14         for _, d := range dividends {
15             dividend += d
16         }
17         return float64(dividend/divisor), nil
18     }
19 }
20
21 func main() {
22
23     // named variadic function call returning multiple values
24     quotient, err := divide(2, 1, 2, 3)
25
26     if err != nil {
27         fmt.Printf("Division failed: %v\n", err)
28     } else {
29         fmt.Printf("Quotient is %v\n", quotient)
30     }
31
32     // division by zero
33     quotient, err = divide(0, 1, 2, 3)
34
35     if err != nil {
36         fmt.Printf("Division failed: %v\n", err)
37     } else {
38         fmt.Printf("Quotient is %v\n", quotient)
39     }
40
41     // empty variable parameter uses type's default value, i.e. 0
```

```

42 quotient, err = divide(2, )
43
44 if err != nil {
45     fmt.Printf("Division failed: %v\n", err)
46 } else {
47     fmt.Printf("Quotient is %v\n", quotient)
48 }
49 }

```

Listing 3.8: Compile and run "named-functions.go"

```

1 $ go run named-functions.go
2 Quotient is 3
3 Division failed: Division by zero
4 Quotient is 0

```

Functions without a name are called function literals and called where defined. We have to pay attention to the scope of the variables inside the function body of them. If we do not pass the variable explicitly, a change of the variable persists outside the function. Otherwise, we work on a copy of the variable.

The following example shows two function literals each within a for-loop increasing a variable displayed from inside the function. To the first function, we do not explicitly pass the variable, to the second we do.

Listing 3.9: "anonymous-functions.go"

```

1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8
9     for i := 0; i < 3; i++ {
10
11         // evaluated anonymous function as closure;
12         // variable i is in the outer scope
13         // but can be changed from inside the function
14         func() {
15             fmt.Printf("Variable i: %v\n", i)
16             i++
17         }() // <<<<< no explicit variable passing, i.e., call-by-reference
18     }
19
20     fmt.Println()
21
22     for i := 0; i < 3; i++ {
23
24         // evaluated anonymous function with parameter passed by value;
25         // variable i is passed by value and only a copy is inside
26         func(i int) {
27             fmt.Printf("Variable i: %v\n", i)
28             i++
29         }(i) // <<<<< explicit variable passing, i.e., call-by-value
30     }
31 }

```

Listing 3.10: Compile and run "anonymous-functions.go"

```
1 $ go run anonymous-functions.go
2 Variable i: 0
3 Variable i: 2
4
5 Variable i: 0
6 Variable i: 1
7 Variable i: 2
```

The compiler does not support pure functional programming, i.e., it does not skip to load the same function repeatedly. Therefore tail recursion has no performance gain compared to other recursions [Fra].

## Methods

*"A type determines a set of values together with operations and methods specific to those values."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Gomb]

We can use any type as a so-called receiver in the definition and the calls of a function. Internally, the receiver is just the first function parameter and can be passed by reference or by value, i.e., the function works via a reference on the outside or on a copy inside. Functions with a receiver are called methods of it.

A type - like a struct - together with its method set is an OOP-like object. As fields, methods are exported outside of the package implicitly by the identifier. If the receiver is not exported, exporting one of its methods works fine, and we can access it from outside of the package.

The following example shows a struct with two fields and a method processing both of them.

Listing 3.11: "objects.go"

```
1 package main
2
3 import "fmt"
4
5 type rectangle struct {
6     width int
7     height int
8 }
9 func (rect *rectangle) area() int {
10     return rect.width * rect.height
11 }
12
13 func main() {
14     quadrat := rectangle{
15         width: 8,
```

```

16     height: 8,
17 }
18
19 fmt.Printf("quadrat.area(): %v\n", quadrat.area())
20 }

```

Listing 3.12: Compile and run "objects.go"

```

1 $ go run objects.go
2 quadrat.area(): 64

```

The orthogonal concept<sup>1</sup> of Go gives other types method sets as well.

The following example shows a slice of floating numbers and a method processing them.

Listing 3.13: "vector-methods.go"

```

1 package main
2
3 import (
4     "fmt"
5     "math"
6 )
7
8 type vector []float64
9
10 func (v vector) abs() float64 {
11     sumOfSquares := 0.0
12     for i := range v {
13         sumOfSquares += v[i] * v[i]
14     }
15     return math.Sqrt(sumOfSquares)
16 }
17
18 func main() {
19     v := vector{2, 4, 4}
20     fmt.Printf("Absolute value of vector %v: %v\n", v, v.abs())
21 }

```

Listing 3.14: "Compile and run "vector-methods.go"

```

1 $ go run vector-methods.go
2 Absolute value of vector [2 4 4]: 6

```

## Interfaces

*"An interface type specifies a method set called its interface. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to implement the interface."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Goi]

<sup>1</sup>"In a purely orthogonal design, operations do not have side effects; each action (...) changes just one thing without affecting others." [Eri03]

An interface can be used as a function parameter or for variable declaration. The Go language has no default implementations for interfaces. A type implements an interface implicitly, i.e., if it defines every method of the interface. Therefore, any type implements an empty interface. In other words, an empty interface can store any type.

The following example shows an interface with one method implemented by two types. Then, we use the interface as a function parameter and as a variable type for assignments. In both cases, we can handle all types implementing the interface.

Listing 3.15: "interfaces.go"

```
1 package main
2
3 import (
4     "fmt"
5     "math"
6 )
7
8 type rectangle struct{ width, height float64 }
9
10 type circle struct{ radius float64 }
11
12 // interface definition
13 type calculator interface {
14     calculateArea() float64
15 }
16
17 // implement interface as method
18 func (r rectangle) calculateArea() float64 {
19     return r.width * r.height
20 }
21
22 // implement interface as method
23 func (c circle) calculateArea() float64 {
24     return math.Pi * c.radius * c.radius
25 }
26
27 // interface as function parameter
28 func showArea(c calculator) {
29     fmt.Printf("%+v: area: %v\n",
30         c, c.calculateArea())
31 }
32
33 func main() {
34     r := rectangle{width: 3, height: 4}
35     c := circle{radius: 5}
36
37     // interface implementations as function parameter
38     showArea(r)
39     showArea(c)
40
41     // interface for variable declaration
42     var calc calculator
43     calc = r
44     fmt.Printf("%+v: area: %v\n", calc, calc.calculateArea())
45     calc = c
46     fmt.Printf("%+v: area: %v\n", calc, calc.calculateArea())
47 }
```

Listing 3.16: Compile and run "interfaces.go"

```

1  $ go run interfaces.go
2  {width:3 height:4}: area: 12
3  {radius:5}: area: 78.53981633974483
4  {width:3 height:4}: area: 12
5  {radius:5}: area: 78.53981633974483

```

A type can implement many interfaces, and interfaces can contain other interfaces. Therefore, we see a tendency to minimize the number of methods, because we can combine them flexibly later as needed. We find many interfaces, e.g., in the standard library, which have only one method.

## Go's OOP Model

*"...James Gosling (Java's inventor) was the featured speaker. ...someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes." he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible."*

Quote by STEVE FRANCA [Stec]

Go has neither objects nor implementation inheritance. Go has structs with fields, functions with receivers, and interfaces. So, how are relations between types implemented?

We can use structs with other structs to compose so-called has-a relationships.

The following example shows how we formulate a simple has-a relationship.

Listing 3.17: "oop-composition.go"

```

1  package main
2
3  import "fmt"
4
5  type Author struct {
6      Name string
7      City string
8  }
9
10 type Book struct {
11     Title string
12     Author Author
13 }
14
15 func main() {
16
17     b := Book{
18         Title: "Just A Book",
19         Author: Author{
20             Name: "W. Riter",
21             City: "Munich",
22         },

```

```

23 }
24
25 fmt.Printf("%q was written from %s in %s\n",
26     b.Title, b.Author.Name, b.Author.City)
27 }

```

Listing 3.18: Compile and run "oop-composition.go"

```

1 $ go run oop-composition.go
2 "Just A Book" was written from W. Riter in Munich

```

Although we can create complicated data structures, we would neither name them objects nor speak about inheritance in their respect. Methods are missing. Now, let's see how we can combine has-a relationships with methods.

The following example shows a struct which is a field type of two other structs. This struct and one of the field structs have the same method signature but different bodies. If we call the methods of these two structs, we see the implemented methods executed. The other field struct has no such method implemented. If we try to call its method, we recognize the method of the outer struct is performed. We will call this behavior *subtyping*.

Listing 3.19: "oop-subtyping.go"

```

1 package main
2
3 import "fmt"
4
5 type Artist struct { Name string }
6
7 type Writer struct { Artist }
8
9 type Painter struct { Artist }
10
11 func (a *Artist) Talk() {
12     fmt.Printf("I am an artist named %s\n", a.Name)
13 }
14
15 func (w *Writer) Talk() {
16     fmt.Printf("I am a writer named %s\n", w.Name)
17 }
18
19 func main() {
20     artist := Artist{Name: "Alex"}
21     writer := Writer{Artist: Artist{Name: "William"}}
22     painter := Painter{Artist: Artist{Name: "Paul"}}
23
24     artist.Talk()
25     writer.Talk()
26     painter.Talk()
27 }

```

Listing 3.20: Compile and run "oop-subtyping.go"

```

1 $ go run oop-subtyping.go
2 I am an artist named Alex
3 I am a writer named William
4 I am an artist named Paul

```

We can use structs and interfaces to compose so-called is-a relationships or interface inheritance.

The following example shows an interface and again a struct, which is a field type of two other structs. This struct and one of the field structs implement the same interface. You can call all three methods - even the not implemented one, which uses the implementation of its field type instead.

Listing 3.21: "oop-interfaces.go"

```
1 package main
2
3 import "fmt"
4
5 type Human struct { Name string }
6
7 type Man struct { Human }
8
9 type Women struct { Human }
10
11 type Talker interface { Talk() }
12
13 func (h *Human) Talk() {
14     fmt.Printf("I am an human named %s\n", h.Name)
15 }
16
17 func (m *Man) Talk() {
18     fmt.Printf("I am a man named %s\n", m.Name)
19 }
20
21 func main() {
22     jamie := Human{Name: "Jamie"}
23     claudie := Man{Human: Human{Name: "Claude"}}
24     cameron := Women{Human: Human{Name: "Cameron"}}
25
26     jamie.Talk()
27     claudie.Talk()
28     cameron.Talk()
29 }
```

Listing 3.22: Compile and run "oop-interfaces.go"

```
1 $ go run oop-interfaces.go
2 I am an human named Jamie
3 I am a man named Claude
4 I am an human named Cameron
```

In Go, subtyping and interfaces, respectively, establish a semantic is-a relationship, while inheritance would only reuse implementation syntactically.

Every type implements the empty interface 'interface{}'. Therefore it can be used as universal parameter type. Empty interface and type switch, a combination of the known switch statement and type assertion, is the kind of runtime generics Go provides. Due to run-time type discovery costs, avoid it if possible.



The following example shows how you use an empty interface as function parameter type. Now, you can call the function with every type. In the function body, we use a type switch to handle the parameter accordingly.

Listing 3.23: "empty-interface.go"

```
1 package main
2
3 import (
4     "errors"
5     "fmt"
6 )
7
8 func processExpectedTypes(in interface{}) (err error) {
9     switch value := in.(type) {
10    case string:
11        fmt.Printf("Processing the string: %q\n", value)
12    case int:
13        fmt.Printf("Processing the integer: %d\n", value)
14    default:
15        return errors.New(fmt.Sprintf("no expected type: %T", in))
16    }
17    return nil
18 }
19
20 func main() {
21     for _, value := range []interface{}{1, "2", 3.0} {
22         if err := processExpectedTypes(value); err != nil {
23             fmt.Printf("Error: %v\n", err)
24         }
25     }
26 }
```

Listing 3.24: Compile and run "empty-interface.go"

```
1 $ go run empty-interface.go
2 Processing the integer: 1
3 Processing the string: "2"
4 Error: no expected type: float64
```

*"The generic dilemma is this: do you want slow programmers, slow compilers and bloated binaries, or slow execution times?"*

Quote by RUSS COX [Rus09]

### 3.1.3 Goroutines, Channels, and Select Statements

Goroutines, Channels, and Select Statements - these are two types and one statements which handle the communication between threads.

#### Goroutines

*"A 'go' statement starts the execution of a function call as an independent concurrent thread of control, or goroutine, within the same address space."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Gor]

To start a thread, you call a function or method by prepending your expression with the keyword *go*. We call it a goroutine, and its execution is in the background, not blocking the process which started it.

The following example shows the comparison of two parallel executions of the same function - one as goroutine.

Listing 3.25: "go-routines.go"

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func count(str string) {
9     for i := 0; i < 5; i++ {
10         fmt.Printf("%s: %v\n", str, i)
11         time.Sleep(time.Microsecond)
12     }
13 }
14
15 func main() {
16     go count("go count")
17     count("count")
18 }
```

Listing 3.26: Compile and run "go-routines.go"

```
1 $ go run go-routines.go
2 count: 0
3 count: 1
4 count: 2
5 go count: 0
6 count: 3
7 go count: 1
8 count: 4
9 go count: 2
```

Go-routines terminate if either its function or the program ends.

## Channels

*"A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type."*

Quote from THE GO PROGRAMMING LANGUAGE SPECIFICATION [Goca]

Channels have an element type, a direction, and a buffer size. They are initialized and closed by the built-in functions *make* and *close*, respectively. They are used by the `<-` operator to send or receive. A buffered channel can store data until picked up by a recipient.

The following example shows how you start a goroutine trying to send through an unbuffered channel. As long as there is no recipient at the other side of the channel, nothing happens. If a recipient is ready, the communication takes place.

Listing 3.27: "channels.go"

```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     // Initialize a channel of int with default buffer size 0
8     channel := make(chan int)
9
10    // Ready to receive values in the background
11    go func() { channel <- 41 }()
12
13    // Sends the value, and synchronized communication takes place
14    fmt.Printf("Received: %v\n", <-channel)
15 }
```

Listing 3.28: Compile and run "channels.go"

```
1 $ go run channels.go
2 Received: 41
```

Channels can do synchronization tasks, as well.

The following example shows how you can synchronize two processes. The main process waits as the recipient in front of a channel until the goroutine sends something through it.

Listing 3.29: "channel-sync.go"

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     done := make(chan bool)
```

```

9
10 go func() {
11     fmt.Println("do some work until it's done")
12     done <- true
13 }()
14
15 fmt.Println("without channel it would be terminated here")
16 <-done
17 }

```

Listing 3.30: Compile and run "channel-sync.go"

```

1 $ go run channel-sync.go
2 without channel it would be terminated here
3 do some work until it's done

```

## Select Statement

*"A 'select' statement chooses which of a set of possible send or receive operations will proceed. It looks similar to a 'switch' statement but with the cases all referring to communication operations."*

Quote from THE GO STANDARD LIBRARY [Gosf]

The *select* statement is usually inside an infinite loop running in the background and processes incoming signals accordingly in parallel threads.

The following example shows how a select statement processes signals. If the first signal is coming through a specific channel, it returns from the for-loop.

Listing 3.31: "select.go"

```

1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     chanString := make(chan string)
10    chanInt := make(chan int)
11    chanDone := make(chan bool)
12
13    go func() {
14        for {
15            select {
16                case str := <-chanString:
17                    fmt.Printf("received a string: %q\n", str)
18                case i := <-chanInt:
19                    fmt.Printf("received an integer: %v\n", i)
20                case done := <-chanDone:
21                    fmt.Printf("received a done signal: %v\n", done)
22                return
23            }
24        }
25    }()
26 }

```

```

24     }
25 }()
26
27 go func() {
28     for {
29         chanInt <- 1
30     }
31 }()
32
33 go func() {
34     for {
35         chanString <- "a"
36     }
37 }()
38
39 time.Sleep(time.Microsecond * 2)
40 chanDone <- true
41 }

```

Listing 3.32: Compile and run "select.go"

```

1 $ go run select.go
2 received a string: "a"
3 received a string: "a"
4 received an integer: 1
5 received a string: "a"
6 received a done signal: true

```

## "Functional Streaming"

To illustrate what is possible having the concurrency primitives combined with functions here is an approach called functional streaming [Ste17]. At its core is an infinite loop around a select statement which makes the fields of a struct called *node* exchangeable while it is processing data streaming in and out defined by the same fields of this specific struct. There are an input channel and an output channel which can both be changed as well as the processing function.

The following example shows functional streaming with two nodes. We connect the two nodes, i.e., one node's output channel is the input channel of the other. We start streaming data from a producer to the first node, which processes the data by its function. The second node's function is just printing out. During the streaming, we change the processing function of the first node.

Listing 3.33: "functional-streaming.go"

```

1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 type NodeOfInt struct {
9     in chan int // Input channel

```

```

10  cin chan chan int // can be exchanged.
11
12  f func(int) int // Function
13  cf chan func(int) int // can be exchanged.
14
15  out chan int // Output channel
16  cout chan chan int // can be exchanged.
17 }
18
19 func (node *NodeOfInt) Start() {
20     go func() {
21         for {
22             select {
23
24                 case in := <-node.in:
25                     node.out <- node.f(in) // Handle data (DEADLOCKS!)
26
27                 case node.in = <-node.cin: // Change input channel
28                 case node.f = <-node.cf: // Change function
29                 case node.out = <-node.cout: // Change output channel
30             }
31         }
32     }()
33 }
34
35 func NewNodeOfInt() *NodeOfInt {
36     node := NodeOfInt{}
37     node.in = make(chan int)
38     node.cin = make(chan chan int)
39     node.f = func(in int) int { return in } // Default returns input value
40     node.cf = make(chan func(int) int)
41     node.out = make(chan int)
42     node.cout = make(chan chan int)
43     node.Start()
44     return &node
45 }
46
47 func (node *NodeOfInt) Connect(nextNode *NodeOfInt) *NodeOfInt {
48     node.cout <- nextNode.in
49     return nextNode
50 }
51
52 func (node *NodeOfInt) SetFunc(f func(int) int) { node.cf <- f }
53
54 func (node *NodeOfInt) Printf(format string) {
55     go func() {
56         for {
57             select {
58                 case in := <-node.out:
59                     fmt.Printf(format, in)
60             }
61         }
62     }()
63 }
64
65 func (node *NodeOfInt) ProduceAtMs(n time.Duration) *NodeOfInt {
66     go func() {
67         for {
68             select {
69                 default:
70                     node.in <- 0
71             } // Trigger permanently
72             time.Sleep(time.Millisecond * n) // with delay in ms
73         }
74     }()
75     return node

```

```

76 }
77
78 func main() {
79     node_1, node_2 := NewNodeOfInt(), NewNodeOfInt() // create nodes
80     var i int //
81     node_1.SetFunc(func(in int) int { i++; return in + i }) //
82     node_2.SetFunc(func(in int) int { return in * 2 }) //
83
84     node_1.Connect(node_2).Printf("%v ") // stream configuration
85     node_1.ProduceAtMs(50) // sending data
86     time.Sleep(time.Second)
87     fmt.Println()
88
89     node_2.SetFunc(func(in int) int { return in * 10 }) // change function
90     time.Sleep(time.Second)
91 }

```

Listing 3.34: Compile and run "functional-streaming.go"

```

1 $ go run functional-streaming.go
2 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
3 210 220 230 240 250 260 270 280 290 300 310 320 330 340 350 360 370 380
4 390 400

```

## 3.2 Standard Library

The standard library of Go contains nearly 200 packages. We will take a look at for us most relevant packages in this section.

### 3.2.1 Network Packages

*"Package net provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets."*

Quote from THE GO STANDARD LIBRARY [Gosd]

At the heart of the *net* package are two interfaces: *Conn* to send a request and *Listener* to listen on requests, accordingly, to connect via the network.

The following example shows how to set up a listener waiting for connections to reply to requests. We create a listener handling requests to a specified network address. The listener accepts connections inside an infinite for loop and starts a goroutine for each of them. The goroutine reads the request, replies, and closes the connection.

Listing 3.35: "net-listener.go"

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "log"
7     "net"
8 )
9
10 func main() {
11
12     // listen for TCP connections on localhost port 22365
13     listener, err := net.Listen("tcp", "localhost:22365")
14     if err != nil {
15         log.Fatal(err)
16     }
17     defer listener.Close()
18
19     // wait for connections
20     for {
21         // accept connection
22         conn, err := listener.Accept()
23         if err != nil {
24             log.Fatal(err)
25         }
26         // create a goroutine for connection
27         go func(conn net.Conn) {
28
29             // read and print the message
30             msg, err := bufio.NewReader(conn).ReadString('\n')
31             if err != nil {
32                 log.Fatal(err)
33             }
34             fmt.Printf("Message received: %s", msg)
35
36             // send reply
```



```

37     conn.Write([]byte(fmt.Sprintf("Message accepted: %s", msg)))
38
39     // close connection
40     conn.Close()
41 } (conn)
42 }
43 }

```

The following example shows how to send a request and to receive a reply. We dial a network address and get a connection as a result. Now, we can send and receive data over the connection.

Listing 3.36: "net-dial.go"

```

1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "log"
7     "net"
8 )
9
10 func main() {
11
12     // create TCP connection to localhost port 22365
13     conn, err := net.Dial("tcp", "localhost:22365")
14     if err != nil {
15         log.Fatal(err)
16     }
17
18     // send message
19     fmt.Fprintf(conn, "Hi, it's me :)\n")
20
21     // receive and print reply
22     reply, err := bufio.NewReader(conn).ReadString('\n')
23     fmt.Print(reply)
24
25     // close connection
26     conn.Close()
27 }

```

Listing 3.37: Compile and run "net-listener.go"

```

1 $ go run net-listener.go
2 Message received: Hi, it's me :)
3 ^C

```

Listing 3.38: Compile and run "net-dial.go"

```

1 $ go run net-dial.go
2 Message accepted: Hi, it's me :)

```

An important subpackage is *net/http* which provides HTTP client and server implementations.

The following example shows how a server can be defined by only one function, a handler, implementing an interface. It handles the request and writes the response. We start the server by a function call with the network address and the handler as arguments.

Listing 3.39: "webserver.go"

```

1 package main
2
3 import (
4     "fmt"
5     "net/http"
6 )
7
8 // function to handle a request, i.e. write a reply
9 func handler(w http.ResponseWriter, r *http.Request) {
10     fmt.Fprintf(w, "Hello, %s.", r.URL.Path[1:])
11 }
12
13 func main() {
14
15     // listen on TCP for localhost on port 8080
16     // and serve the handled request on incoming connections
17     http.ListenAndServe("localhost:8080", http.HandlerFunc(handler))
18 }

```

Listing 3.40: Compile and run "webserver.go"

```

1 $ go run go run webserver.go
2 request "/little%20webserver/gopher" received from : 127.0.0.1:51099
3 ^C

```

Listing 3.41: Curl the webserver

```

1 $ curl http://localhost:8080/little%20webserver/gopher
2 Hello, little webserver/gopher.

```

An HTTP request multiplexer does multiplexing between various URL paths and executes defined functions appropriately.

The following example shows how method calls pass the path and the function, which implements the handler interface, to the multiplexer. Then, we start the server with the multiplexer as the handler.

Listing 3.42: "net-mux.go"

```

1 package main
2
3 import (
4     "fmt"
5     "log"
6     "net/http"
7 )
8
9 func main() {
10     // Create handler to multiplex according to the URL path
11     handler := http.NewServeMux()
12
13     // Register anonymous handler function to URL path "/hello "
14     handler.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
15         fmt.Fprintln(w, "I say hello to you!")
16     })
17
18     // Register anonymous handler function to URL path "/good-morning "
19     handler.HandleFunc("/good-morning", func(w http.ResponseWriter, r *http.Request) {
20         fmt.Fprintln(w, "I say good morning to you!")
21     })
22
23     // Listen and serve on localhost:22365
24     log.Fatal(http.ListenAndServe(":22365", handler))
25 }

```

Listing 3.43: Compile and run "net-mux.go"

```

1 $ go run go run net-mux.go
2 ^C

```

Listing 3.44: Curl the webserver

```

1 $ curl localhost:22365/hello
2 I say hello to you!
3 $ curl localhost:22365/good-morning
4 I say good morning to you!

```

### 3.2.2 Context Package

*"Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes."*

Quote from THE GO STANDARD LIBRARY [Gosb]

The context package provides three ways to finish a running thread. These are, explicit via calling a cancel function, after a timeout or a deadline. We pass the context as a parameter and implement the intended ending. If the context is done all running threads of the context come to an end.

The following example shows how to use a context with cancel function. We create a context to be canceled by a function call. Then, we start goroutines waiting to receive a cancelation signal from their context. We call the cancel function, and the goroutines come to an end as implemented.

Listing 3.45: 'ctx-cancel.go'

```

1 package main
2
3 import (
4     "context"
5     "fmt"
6     "time"
7 )
8
9 func main() {
10
11     // create context and function for cancellation
12     ctx, cancel := context.WithCancel(context.Background())
13
14     // call go routines waiting for its cancellation
15     for i := 0; i < 4; i++ {
16         go func(i int) {
17             for {
18                 select {
19                     case <-ctx.Done():
20                         fmt.Printf("%v: ctx.Done(): %v\n", i, ctx.Err())
21                         return
22                 }
23             }
24         }(i)
25     }
26
27     // cancel context and go routines, respectively
28     cancel()
29
30     // wait for printed messages
31     time.Sleep(time.Millisecond)
32 }

```

Listing 3.46: Compile and run "ctx-cancel.go"

```

1 $ go run ctx-cancel.go
2 1: ctx.Done(): context canceled
3 0: ctx.Done(): context canceled
4 3: ctx.Done(): context canceled
5 2: ctx.Done(): context canceled

```

The following example shows how to use a context with a timeout. We create a context which cancels each thread after the same timeout. Then, we start goroutines waiting to receive a cancellation signal from their context. After the timeout, a goroutine comes to an end as implemented.

Listing 3.47: "ctx-timeout.go"

```

1 package main
2
3 import (
4     "context"
5     "fmt"
6     "time"
7 )
8
9 func main() {
10     // create context with timeout for cancellation
11     ctx, cancel := context.WithTimeout(context.Background(),
12                                     50*time.Millisecond)
13     defer cancel()

```

```

14
15 // call go routines waiting for its cancellation
16 for i := 0; i < 4; i++ {
17     time.Sleep(time.Second)
18     go func(i int) {
19         for {
20             select {
21                 case <-ctx.Done():
22                     fmt.Printf("%v: ctx.Done(): %v\n", i, ctx.Err())
23                     return
24             }
25         }
26     }(i)
27 }
28
29 // wait for deadline cancelling context and go routines, respectively
30 // and for printed messages
31 time.Sleep(time.Second * 5)
32 }

```

Listing 3.48: Compile and run "ctx-timeout.go"

```

1 $ go run ctx-timeout.go
2 2: ctx.Done(): context deadline exceeded
3 1: ctx.Done(): context deadline exceeded
4 3: ctx.Done(): context deadline exceeded
5 0: ctx.Done(): context deadline exceeded

```

As well, we can cancel a context by a deadline. Then, all threads are canceled at the same time. Here is an example of how to create such a context.

Listing 3.49: "Context with deadline"

```

1 ctx, cancel := context.WithDeadline(context.Background(), time.Now().Add(time.Second*5))

```

### 3.2.3 JSON Package

*"Package json implements encoding and decoding of JSON as defined in RFC 7159."*

Quote from THE GO STANDARD LIBRARY [Gosc]

The encoding/json package provides the transformation of Go values to binary JSON representations and the other way round.

The following example shows how we marshal a slice of structs into an array of bytes and how to display it formatted.

Listing 3.50: "marshal-json.go"

```

1 package main
2
3 import (
4     "encoding/json"
5     "log"
6     "fmt"
7     "os"
8     "bytes"
9 )
10
11 type Animal struct {
12     Name      string
13     Kind      string
14     NumberOfLegs int
15 }
16
17 func main() {
18     animals := []Animal{
19         Animal{
20             "alice",
21             "cat",
22             4,
23         },
24         Animal{
25             "bob",
26             "bird",
27             2,
28         },
29         Animal{
30             "curt",
31             "fish",
32             0,
33         },
34     }
35
36     // Marshal array of struct
37     if byteArray, err := json.Marshal(animals); err != nil {
38         log.Fatal(err)
39     } else {
40         // Unformatted JSON
41         fmt.Printf("%s\n", byteArray)
42
43         //Formatted JSON
44         var out bytes.Buffer
45         json.Indent(&out, byteArray, "", "\t")
46         out.WriteTo(os.Stdout)
47     }
48 }

```

Listing 3.51: Compile and run "marshal-json.go"

```

1 $ go run marshal-json.go
2 [{ "Name": "alice", "Kind": "cat", "NumberOfLegs": 4 }, { "Name": "bob", "Kind": "bir
3 d", "NumberOfLegs": 2 }, { "Name": "curt", "Kind": "fish", "NumberOfLegs": 0 }]
4 [
5     {
6         "Name": "alice",
7         "Kind": "cat",
8         "NumberOfLegs": 4
9     },
10    {
11        "Name": "bob",
12        "Kind": "bird",
13        "NumberOfLegs": 2

```

```

14         },
15         {
16             "Name": "curt",
17             "Kind": "fish",
18             "NumberOfLegs": 0
19         }
20     ]

```

The following example shows how we unmarshal an array of bytes into a slice of structs.

Listing 3.52: "unmarshal-json.go"

```

1 package main
2
3 import (
4     "encoding/json"
5     "fmt"
6 )
7
8 type Animal struct {
9     Name      string
10    Kind       string
11    NumberOfLegs int
12 }
13
14 func main() {
15     // Define JSON string
16     jsonAnimals := []byte(`
17     [{"Name": "alice", "Kind": "cat", "NumberOfLegs": 4},
18     {"Name": "bob", "Kind": "bird", "NumberOfLegs": 2},
19     {"Name": "curt", "Kind": "fish", "NumberOfLegs": 0}]`)
20
21     // Unmarshal JSON into array of struct
22     var animals []Animal
23     err := json.Unmarshal(jsonAnimals, &animals)
24     if err != nil {
25         fmt.Println("error:", err)
26     }
27     // Print type and values
28     fmt.Printf("%T: %+v", animals, animals)
29 }

```

Listing 3.53: Compile and run "unmarshal-json.go"

```

1 $ go run unmarshal-json.go
2 []main.Animal: [{Name:alice Kind:cat NumberOfLegs:4} {Name:bob Kind:bird
3 NumberOfLegs:2} {Name:curt Kind:fish NumberOfLegs:0}]

```

The package supports encoding and decoding for existing data (marshal/unmarshal) and data streams (encoder/decoder).

The following example shows how to use a JSON decoder from standard input and a JSON encoder to standard output. In between, we capitalize the two string fields of the structs.

Listing 3.54: "coder-json.go"

```

1 package main
2
3 import (
4     "encoding/json"
5     "log"
6     "os"
7     "strings"
8 )
9
10 func main() {
11
12     // Define decoder for reading JSON string
13     decoder := json.NewDecoder(os.Stdin)
14
15     // Define encoder for outputting JSON
16     encoder := json.NewEncoder(os.Stdout)
17
18     for {
19         // Decode string into map
20         var jsonMap map[string]interface{}
21         if err := decoder.Decode(&jsonMap); err != nil {
22             // EOF expected
23             return
24         }
25         // Range map to capitalize string values
26         for key := range jsonMap {
27             if convertedValue, ok := jsonMap[key].(string); ok {
28                 jsonMap[key] = strings.Title(convertedValue)
29             }
30         }
31
32         // Encode output
33         if err := encoder.Encode(&jsonMap); err != nil {
34             log.Println(err)
35         }
36     }
37 }

```

Listing 3.55: Build and run "coder-json.go"

```

1 $ go build -o coder-json
2 $ echo '{"Name": "alice", "Kind": "cat", "NumberOfLegs": "4"}
3 {"Name": "bob", "Kind": "bird", "NumberOfLegs": "2"}
4 {"Name": "curt", "Kind": "fish", "NumberOfLegs": "0"}' | ./coder-json
5 {"Kind": "Cat", "Name": "Alice", "NumberOfLegs": "4"}
6 {"Kind": "Bird", "Name": "Bob", "NumberOfLegs": "2"}
7 {"Kind": "Fish", "Name": "Curt", "NumberOfLegs": "0"}

```

By using the empty interface, we can handle arbitrary data structures, as well.

The following example shows how to decode JSON into a map of empty interfaces. We use a type switch to retrieve the type and to convert the value accordingly.



Listing 3.56: 'arbitrary-json.go'

```

1 package main
2
3 import (
4     "encoding/json"
5     "fmt"
6 )
7
8 func main() {
9     // Define JSON string
10    b := []byte(`{"Kind": "Mouse", "NumberOfLegs": 4, "Names": ["Bernard",
11    "Bianca"]}`)
12
13    // Unmarshal JSON into map of empty interfaces
14    var mapOfInterfaces map[string]interface{}
15    err := json.Unmarshal(b, &mapOfInterfaces)
16    if err != nil {
17        fmt.Println("error:", err)
18    }
19
20    // Range with type switch
21    for key, value := range mapOfInterfaces {
22        switch convertedValue := value.(type) {
23        case string:
24            fmt.Printf("%q (string): %s\n", key, convertedValue)
25        case float64:
26            fmt.Printf("%q (float64): %f\n", key, convertedValue)
27        case []interface{}:
28            fmt.Printf("%q ([] interface {}): %v\n", key, convertedValue)
29        }
30    }
31 }

```

Listing 3.57: Compile and run "arbitrary-json.go"

```

1 $ go run arbitrary-json.go
2 "Kind" (string): Mouse
3 "NumberOfLegs" (float64): 4.000000
4 "Names" ([] interface {}): [Bernard Bianca]

```

### 3.2.4 Testing Package

*"Package testing provides support for automated testing of Go packages."*

Quote from THE GO STANDARD LIBRARY [Gose]

The package *testing* provides the functionality to write test functions which named with "Test" prefix and defined with one of its types as the argument. By convention, we create a Go file named after the file to test. The compiler's subcommand *test* does the testing then.

Listing 3.58: "Test file listing"

```

1 $ ls -l
2 testing-example.go
3 testing-example_test.go

```

The following example shows a simple test function. We loop over test cases defined by two strings and test if the processed string is equal to the expected string.

Listing 3.59: "testing-example\_test.go"

```

1 package testingexample
2
3 import (
4     "testing"
5 )
6
7 func Test_StringReverse(t *testing.T) {
8     for _, testCase := range []struct {
9         origin, expected string
10     }{
11         {"I am not reversed", "desrever ton ma I"},
12         {"1234567890", "0987654321"},
13         {"", ""},
14     } {
15         if Reverse(testCase.origin) != testCase.expected {
16             t.Errorf("origin: %q => %q != expected: %q",
17                 testCase.origin,
18                 Reverse(testCase.origin),
19                 testCase.expected)
20         }
21     }
22 }

```

Listing 3.60: Test execution

```

1 $ go test -v
2 == RUN Test_StringReverse
3 --- PASS: Test_StringReverse (0.00s)
4 PASS
5 ok      bitbucket.org/stefanhans/go-thesis/4.2.5      0.005s

```

The *testing* package is a widely used way to various testings in Go.

## 3.3 Libraries for Distributed Systems

### 3.3.1 HashiCorp's 'memberlist'

The so-called Distributed Group Membership Protocols try to answer the permanent question who is alive in my group.

One state-of-the-art protocol is the SWIM++ protocol. The acronym SWIM [Abh] stands for **S**calable **W**eakly-consistent **I**nfection-style process group **M**embership - the '++' indicates some extra advantages. HashiCorp's 'memberlist' is an implementation of the SWIM++ protocol.

#### Weakly-Consistent

Chosen responsiveness over consistency, which usually is preferable, we can not have a strong consistency where all members have the same knowledge about the group membership at the same time. But, it reaches eventually consistency which we call weakly consistent.

## Infection-Style

An approach to address the "who is alive in my group" is to send a heartbeat to all group members. It does not scale very well. Infection-style is a better way. Instead of heartbeats, it uses a gossip or epidemic approach. A member communicates directly only with a subset of members, which does the same on their side. Finally, all are informed. The protocol disseminates not only the membership related information in an infection-style. It does this as well for failure detection information.

## Suspicion Mechanism For Failure Detection

If a member does not respond to a ping, it will not immediately be marked as dead. Other members are called to contact this suspicious member. Only, if this fails as well, the member is marked as dead.

## Round-Robin Probe Target Selection

The protocol does not select which member to contact next randomly. It chooses a subset of members as a whole and proceeds one member after the other in a loop. Therefore, we expect a fixed interval every member is probed. It inserts new additional members at a random place.

## The 'memberlist' Library

A battle-proven implementation of the SWIM++ protocol is 'memberlist' [Mem] by HashiCorp. Professional products widely use it [Has].

### 3.3.2 Protocol Lab's 'go-libp2p-kad-dht'

Protocol Lab's 'go-libp2p-kad-dht' is an implementation of Kademlia's Distributed Hash Table orchestration. 'libp2p' [Liba] is a stack of modules implementing various network protocols for transport, routing, and data exchange. It is the groundwork of modern decentralized systems like IPFS<sup>2</sup>. We will focus here on 'go-libp2p-kad-dht' [Go a] a library implementing Kademlia [Pet], a peer-to-peer protocol using distributed hash tables and an XOR-based metric for routing.

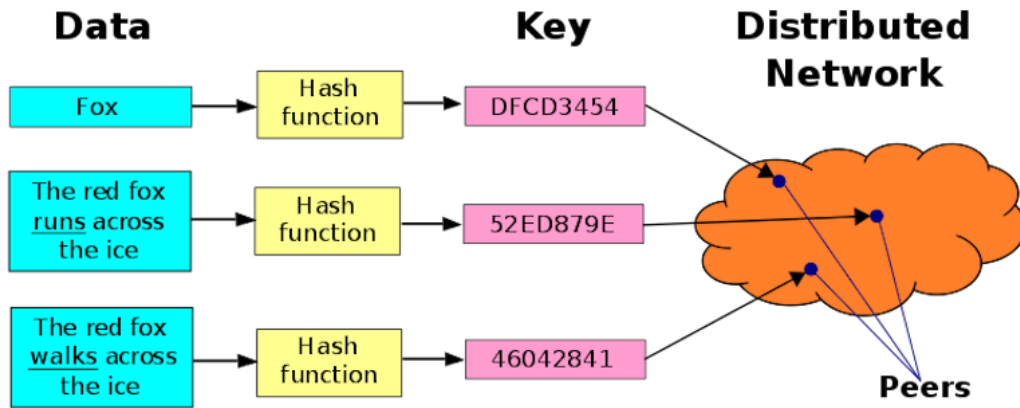
## Distributed Hash Table - DHT

We can use hash functions to create hashes out of data, or key/value pairs, respectively. These key/value pairs, i.e., the hashes and its data, are tamper-proof. If you change one bit of the data, you have a totally different hash key. Typically, all keys have the same length. In a distributed hash table, the key/value pairs are distributed between nodes of an overlay network.

---

<sup>2</sup>*InterPlanetary File System* is a decentralized peer-to-peer filesystem using content-addressing.

Figure 3.1: Distributed Hash Table [Dht]

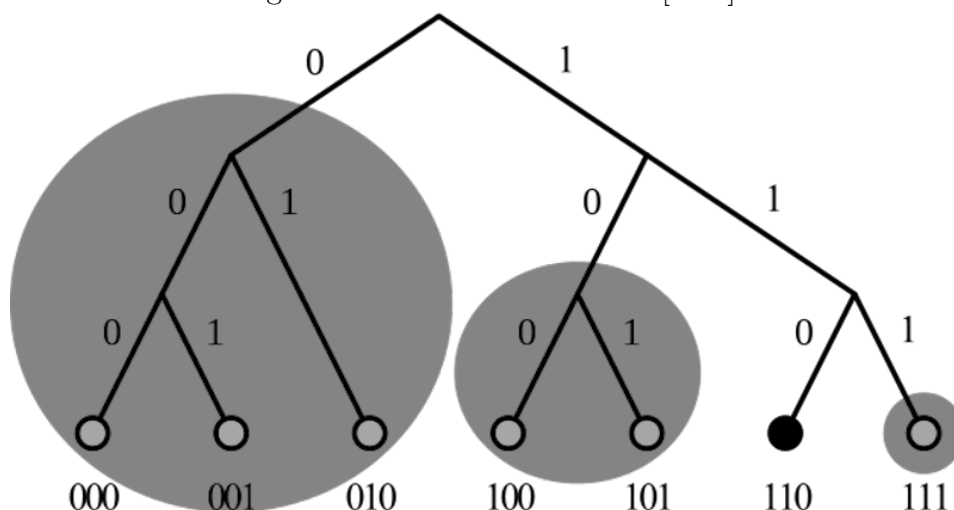


Every node has a unique identifier which is used to create a key/value pair for the node. The nodes and the data share the same key space. Information about other nodes and data are stored in regard to their keys following an XOR-based metric.

## XOR-based Metric

We use an XOR-based metric to process keys which represent other nodes, data or are search keys. To ask nodes for routing, we place nodes in a bucket where the first bit differs between the node's key and the other key. These buckets have a maximum number of nodes configured. If the bucket is full, you move the nodes along their path to new buckets. If you follow this algorithm recursively, you have a logarithmic complexity of  $O(\log 2)$ . The following image visualize the main idea.

Figure 3.2: XOR-based Metric [Kad]



The little black circle is the node with the key 110. The big grey circles are its buckets containing the other nodes according to their keys.

## 'go-libp2p-kad-dht'

We used the 'go-libp2p-kad-dht' library for a prototype of a chat application [Stea][Cha]. The prototype implementation was successful. But, the purpose of 'go-libp2p-kad-dht' was a bit misused, and thus we went on with 'memberlist'.

## 3.4 Other Libraries

### 3.4.1 Protocol Buffer and gRPC

gRPC[Grpa; Grpm], the RPC implementation by Google, and Protocol Buffer[Proh], a serializing framework for structured data, will be used in combination to realize the communication layer of a Reactive System prototype.

#### Protocol Buffers - Google's binary serialization toolset

*"Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data - think XML, but smaller, faster, and simpler."*

Quote from PROTOCOL BUFFERS WEBSITE [Proh]

Protocol Buffer provides a format to define data structures declaratively. Out of the declaration files, a compiler creates the appropriate code for the data serialization, i.e., functionality to create, serialize, and deserialize.

The following example shows a *.proto* file *info.proto* [Infb] with a simple message format type ...

Listing 3.61: "info.proto"

```
1 syntax = 'proto3';
2
3 package info;
4
5 message info {
6     string text = 1;
7     string from = 2;
8 }
```

... and how to compile it into a Go file *info.pb.go* [Infa].

Listing 3.62: Compile "info.proto" and create "info.pb.go"

```
1 $ ls
2   info.proto
3 $ protoc --go_out=. info.proto
4 $ ls
5   info.pb.go info.proto
```

The created go package contains all we need for serialization of the declared data structures.

The following example shows how to create a slice of the declared message type, marshal it into a binary representation, and save it to a file.

Listing 3.63: "write-info.go"

```

1 package main
2
3 import (
4     "encoding/binary"
5     "fmt"
6     "os"
7
8     "bitbucket.org/stefanhans/go-thesis/4.3.1/info-pb"
9     "github.com/golang/protobuf/proto"
10 )
11
12 func main() {
13
14     // Declare array with protobuffer messages
15     infos := []info_pb.Info{
16         info_pb.Info{
17             Text: "I am a painter",
18             From: "Marc Chagall",
19         },
20         info_pb.Info{
21             Text: "I am a writer",
22             From: "Edgar Allan Poe",
23         },
24     }
25
26     // Open file for appending info
27     filename := "storage"
28     file, err := os.OpenFile(filename,
29         os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0666)
30     if err != nil {
31         fmt.Errorf("could not open %s: %v", filename, err)
32     }
33
34     // Range over protobuffers
35     for _, v := range infos {
36
37         // Marshal into binary format
38         byteArray, err := proto.Marshal(&v)
39         if err != nil {
40             fmt.Errorf("could not encode info: %v", err)
41             os.Exit(1)
42         }
43
44         // Write binary representation
45         err := binary.Write(file, binary.LittleEndian,
46             int64(len(byteArray)))
47         if err != nil {
48             fmt.Errorf("could not encode length of message: %v", err)
49         }
50
51         // Write to file
52         _, err = file.Write(byteArray)
53         if err != nil {
54             fmt.Errorf("could not write info to file: %v", err)
55         }
56     }
57
58     // Close file
59     if err := file.Close(); err != nil {
60         fmt.Errorf("could not close file %s: %v", filename, err)
61     }
62 }

```

### Listing 3.64: Compile and run "write-info.go"

```
1 $ go run write-info.go
2 $ file storage
3 storage: data
```

The binary wire format for protocol buffer messages [Proi] begins with the first part of the information needed to understand the second part of the data.

The following example shows how to deserialize from the just created file. Firstly, we read the binary data into a byte array. Then, we decode the leading bytes to get the number of the relevant bytes and to be able to deserialize the following data into the known data structure. In this manner, we walk through the byte array until the end reached.

### Listing 3.65: "read-info.go"

```
1 package main
2
3 import (
4     "bytes"
5     "encoding/binary"
6     "fmt"
7     "io/ioutil"
8
9     "bitbucket.org/stefanhans/go-thesis/4.3.1/info-pb"
10    "github.com/golang/protobuf/proto"
11 )
12
13 func main() {
14     // Open file for reading info
15     filename := "storage"
16     byteArray, err := ioutil.ReadFile(filename)
17     if err != nil {
18         fmt.Errorf("could not read %s: %v", filename, err)
19     }
20
21     for {
22         // Check length of remaining bytes
23         if len(byteArray) == 0 {
24             break
25         } else if len(byteArray) < 8 {
26             fmt.Errorf("remaining odd %d", len(byteArray))
27         }
28
29         // Decode binary data and shift array forward
30         var length int64
31         err := binary.Read(bytes.NewReader(byteArray[:8]),
32                             binary.LittleEndian, &length)
33         if err != nil {
34             fmt.Errorf("could not decode message length: %v", err)
35         }
36         byteArray = byteArray[8:]
37
38         // Unmarshall info
39         var info info_pb.Info
40         if err := proto.Unmarshal(byteArray[:length], &info); err != nil {
41             fmt.Errorf("could not read info: %v", err)
42         }
43         byteArray = byteArray[length:]
44
45         fmt.Printf("%s: %q\n", info.From, info.Text)
```



```
46 | }  
47 | }
```

Listing 3.66: Compile and run "read-info.go"

```
1 $ go run read-info.go  
2 Marc Chagall: "I am a painter"  
3 Edgar Allan Poe: "I am a writer"
```

Very convenient is Protocol Buffers in conjunction with gRPC.

## gRPC - Google's Remote Procedure Call Framework

*"In gRPC a client application can directly call methods on a server application on a different machine as if it was a local object, making it easier for you to create distributed applications and services."*

Quote from GRPC WEBSITE [Grpa]

As an RPC system gRPC does the communicating between server and client transparently. It uses protocol buffers as the language to define its interface and to handle the serialization over the wire.

The following example shows a *.proto* file *info.proto* [Grpo] with a simple message format type and a simple service definition to read and write the messages.

Listing 3.67: "info.proto"

```
1 syntax = 'proto3';  
2  
3 package info;  
4  
5 message Info {  
6     string text = 1;  
7     string from = 2;  
8 }  
9  
10 message InfoList {  
11     // creates a slice of Info  
12     repeated Info infos = 1;  
13 }  
14  
15 // Empty message type used for Read method  
16 message Void {}  
17  
18 // Service definition for gRPC plugin  
19 service Infos {  
20     rpc Read(Void) returns (InfoList) {}  
21     rpc Write(Info) returns (Info) {}  
22 }
```

The protocol buffer compiler uses a gRPC plugin to generate the service interface code and stubs in a Go file *info.pb.go* [Grpn].

Listing 3.68: Compile `"/info/info.proto"` and create `"/info/info.pb.go"`

```

1  $ ls
2  info.proto
3  $ protoc --go_out=plugins=grpc:. info.proto
4  $ ls
5  info.pb.go info.proto

```

We have to declare a receiver implementing the service interface for the server part.

The following example shows how we implement the methods. We marshal the message into a byte array and write it to a file, and we read from a file into a byte array, decode the leading bytes, deserialize the appropriate data and unmarshal into the message.

Listing 3.69: `"/cmd/server/main.go"`

```

1  package main
2
3  import (
4      "bytes"
5      "encoding/binary"
6      "fmt"
7      "io/ioutil"
8      "log"
9      "net"
10     "os"
11
12     "bitbucket.org/stefanhans/go-thesis/4.3.1/info-gRPC/info"
13     "github.com/golang/protobuf/proto"
14     "golang.org/x/net/context"
15     "google.golang.org/grpc"
16 )
17
18 func main() {
19     // Create and register server
20     var infos infoServer
21     srv := grpc.NewServer()
22     info.RegisterInfosServer(srv, infos)
23
24     // Create listener
25     l, err := net.Listen("tcp", ":8888")
26     if err != nil {
27         log.Fatal("could not listen to :8888: %v", err)
28     }
29     // Serve messages via listener
30     log.Fatal(srv.Serve(l))
31 }
32
33 // Receiver for implementing the server service interface
34 type infoServer struct{}
35
36 // Server's Write implementation
37 func (s infoServer) Write(ctx context.Context, info *info.Info)
38     (*info.Info, error) {
39
40     // Marshall message
41     b, err := proto.Marshal(info)
42     if err != nil {
43         return nil, fmt.Errorf("could not encode info: %v", err)
44     }
45
46     // Open file
47     f, err := os.OpenFile("storage",

```

```

48                                     os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0666)
49     if err != nil {
50         return nil, fmt.Errorf("could not open storage: %v", err)
51     }
52
53     // Encode message and write to file
54     err := binary.Write(f, binary.LittleEndian, int64(len(b)))
55     if err != nil {
56         return nil, fmt.Errorf("could not encode length of message: %v",
57                                 err)
58     }
59     _, err = f.Write(b)
60     if err != nil {
61         return nil, fmt.Errorf("could not write info to file: %v", err)
62     }
63
64     // Close file
65     if err := f.Close(); err != nil {
66         return nil, fmt.Errorf("could not close file storage: %v", err)
67     }
68     return info, nil
69 }
70
71 // Server's Read implementation
72 func (s infoServer) Read(ctx context.Context, void *info.Void)
73                             (*info.InfoList, error) {
74
75     // Read file
76     b, err := ioutil.ReadFile("storage")
77     if err != nil {
78         return nil, fmt.Errorf("could not read storage: %v", err)
79     }
80
81     // Iterate over read bytes
82     var infos info.InfoList
83     for {
84         if len(b) == 0 {
85             // Return result
86             return &infos, nil
87         } else if len(b) < 8 {
88             return nil, fmt.Errorf("remaining odd %d bytes", len(b))
89         }
90
91         // Decode message
92         var length int64
93         err := binary.Read(bytes.NewReader(b[:8]), binary.LittleEndian,
94                             &length)
95         if err != nil {
96             return nil, fmt.Errorf("could not decode message length: %v",
97                                     err)
98         }
99         b = b[8:]
100
101         // Unmarshall message and append it
102         var info info.Info
103         if err := proto.Unmarshal(b[:length], &info); err != nil {
104             return nil, fmt.Errorf("could not read info: %v", err)
105         }
106         b = b[length:]
107         infos.Infos = append(infos.Infos, &info)
108     }
109 }

```

Now, we build and start the server.

### Listing 3.70: Build and start server

```

1  $ go build -o server ./cmd/server
2  $ ls
3  cmd      info      server
4  $ ./server
5  ^C

```

We have nothing to implement for the client part. However, we should use it.

The following example shows how we implement the client. We use two little wrappers, to create the message and send it, and to receive the message and print it. Finally, we implement two subcommands to call the client.

### Listing 3.71: `./cmd/client/main.go`

```

1  package main
2
3  import (
4      "flag"
5      "fmt"
6      "log"
7      "os"
8      "strings"
9
10     "bitbucket.org/stefanhans/go-thesis/4.3.1/info-grpc/info"
11     "golang.org/x/net/context"
12     "google.golang.org/grpc"
13 )
14
15 func main() {
16     // Check command args
17     flag.Parse()
18     if flag.NArg() < 1 {
19         fmt.Fprintln(os.Stderr, "missing subcommand: read or write")
20         os.Exit(1)
21     }
22
23     // Create client with insecure connection
24     conn, err := grpc.Dial(":8888", grpc.WithInsecure())
25     if err != nil {
26         log.Fatalf("could not connect to backend: %v", err)
27     }
28     client := info.NewInfosClient(conn)
29
30     // Switch subcommands and call wrapper function
31     switch cmd := flag.Arg(0); cmd {
32     case "read":
33         err = read(context.Background(), client)
34     case "write":
35         if flag.NArg() < 4 {
36             fmt.Fprintln(os.Stderr,
37                 "missing parameter: write <from> <text>...")
38             os.Exit(1)
39         }
40         err = write(context.Background(), client, flag.Arg(1),
41             strings.Join(flag.Args()[2:], " "))
42     default:
43         err = fmt.Errorf("unknown subcommand %s", cmd)
44     }
45     if err != nil {
46         fmt.Fprintln(os.Stderr, err)
47         os.Exit(1)
48     }
49 }

```

```

48 }
49 }
50
51 // Write wrapper function
52 func write(ctx context.Context, client info.InfosClient, from string,
53           text string) error {
54     // Write to gRPC client
55     _, err := client.Write(ctx, &info.Info{From: from, Text: text})
56     if err != nil {
57         return fmt.Errorf("could not add info in the backend: %v", err)
58     }
59     return nil
60 }
61
62 // Read wrapper function
63 func read(ctx context.Context, client info.InfosClient) error {
64     // Read from gRPC client
65     l, err := client.Read(ctx, &info.Void{})
66     if err != nil {
67         return fmt.Errorf("could not fetch info: %v", err)
68     }
69 }
70
71 // Print messages
72 for _, t := range l.Infos {
73     fmt.Printf("%s: %s\n", t.From, t.Text)
74 }
75 return nil
76 }

```

Here now, we run the client to test the functionality.

Listing 3.72: Compile and run client'

```

1 $ go run ./cmd/client/main.go write Marc I am a painter
2 $ go run ./cmd/client/main.go write Edgar I am a writer
3 $ go run ./cmd/client/main.go read
4 Marc: I am a painter
5 Edgar: I am a writer

```

## 3.4.2 TUI

For the creation of a terminal chat, an appropriate text-based user interface (TUI) library is convenient to use. We use minimalist Go package [Gocb] aimed at creating console user interfaces (CUI). The package itself is using the low-level TUI library *nsf/termbox-go* [Ter].

The following example shows how we define a layout with two views. One view has a permanent focus and is editable. The return key sends the text to the other view, which displays the text in a new line, and clears it.

Listing 3.73: "simple-tui.go"

```

1 package main
2
3 import (
4     "log"
5     "github.com/jroimartin/gocui"
6 )
7
8 // Content to be displayed in the GUI
9 func layout(g *gocui.Gui) error {
10
11     // Get size of the terminal
12     maxX, maxY := g.Size()
13
14     // Creates view "messages"
15     messages, err := g.SetView("messages", 0, 0, maxX-1, maxY-3)
16     if err != nil {
17         if err != gocui.ErrUnknownView {
18             return err
19         }
20         messages.Autoscroll = true
21         messages.Wrap = true
22     }
23
24     // Creates view "input"
25     input, err := g.SetView("input", 0, maxY-4, maxX-1, maxY-1)
26     if err != nil {
27         if err != gocui.ErrUnknownView {
28             return err
29         }
30         input.Wrap = true
31         input.Editable = true
32     }
33
34     // Set view "input" as the current view with focus and cursor
35     if _, err := g.SetCurrentView("input"); err != nil {
36         return err
37     }
38
39     // Show cursor
40     g.Cursor = true
41
42     return nil
43 }
44
45 // Quit the GUI
46 func quit(g *gocui.Gui, v *gocui.View) error {
47     return gocui.ErrQuit
48 }
49
50 // Send content from the bottom window to the top window
51 func send(g *gocui.Gui, v *gocui.View) error {
52
53     // Write the buffer of the bottom view to the top view
54     if m, err := g.View("messages"); err != nil {
55         log.Fatal(err)
56     } else {
57         m.Write([]byte(v.Buffer()))
58     }
59
60     // Clear the bottom window and reset the cursor
61     v.Clear()
62     if err := v.SetCursor(0, 0); err != nil {
63         log.Fatal(err)
64     }
65

```

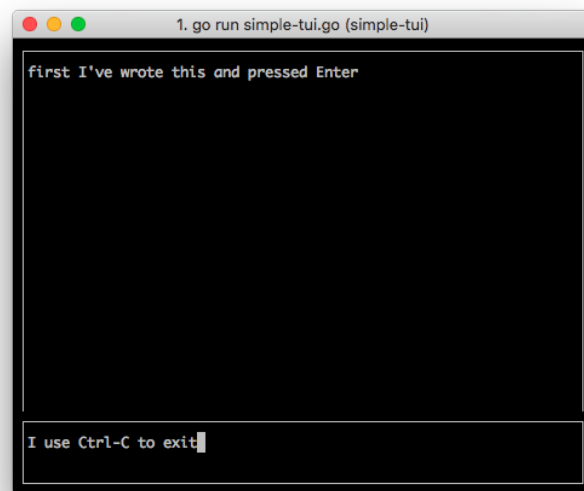
```

66     return nil
67 }
68
69 func main() {
70     // Create the terminal GUI
71     g, err := gocui.NewGui(gocui.OutputNormal)
72     if err != nil {
73         log.Fatal(err)
74     }
75     defer g.Close()
76
77     // Set function to manage all views and keybindings
78     g.SetManagerFunc(layout)
79
80     // Bind keys with functions
81     g.SetKeybinding(" ", gocui.KeyCtrlC, gocui.ModNone, quit)
82     g.SetKeybinding("input", gocui.KeyEnter, gocui.ModNone, send)
83
84     // Start main event loop of the GUI
85     g.MainLoop()
86 }
87 }

```

Here, a snapshot of the example in action.

Figure 3.3: Picture of TUI



### 3.4.3 Liner

*Liner* [Lina] is a line editor we use for the creation of an application internal command line to test, analyze, and play with libraries and implementations. We work as used on the command line, but we are running in our application. So, we have the commands our application provides plus command completion and command history.

The following example shows how we put commands into a slice of strings and use this slice in a function for code completion. The function to execute the commands by processing the line of command has defined a command to exit the application explicitly. Regarding the other commands, it passes them to the shell, the application is running in, and adds them to the command history if the execution returns true. For simplicity, we do not support arguments for commands.

Listing 3.74: "simple-liner.go"

```
1 package main
2
3 import (
4     "bytes"
5     "fmt"
6     "io"
7     "os"
8     "os/exec"
9     "strings"
10    "time"
11
12    "github.com/peterh/liner"
13 )
14
15 // Define a set of commands
16 var commands = []string{"ls", "pwd", "date", "hostname", "quit"}
17
18 func prompt() string {
19     return fmt.Sprintf("<%s %s> ", time.Now().Format("Jan 2 15:04:05.000"), "user")
20 }
21
22 func executeShellScript(cmdline string) bool {
23
24     if cmdline == "quit" {
25         os.Exit(0)
26     }
27
28     words := strings.Fields(cmdline)
29
30     if len(words) == 0 {
31         fmt.Print("\n")
32         return false
33     }
34
35     binary, lookErr := exec.LookPath(words[0])
36     if lookErr != nil {
37         fmt.Printf("%v\n", lookErr)
38         return false
39     }
40
41     var cmd *exec.Cmd
42     if len(words) == 1 {
43         cmd = exec.Command(binary)
44     } else {
45         fmt.Printf("command with arguments not supported\n")
```



```

46     return false
47 }
48
49 var out bytes.Buffer
50 cmd.Stdout = &out
51 cmd.Env = os.Environ()
52 err := cmd.Run()
53 if err != nil {
54     fmt.Printf("could not run: %v\n", err)
55     return false
56 }
57
58 fmt.Printf(out.String())
59
60 return true
61 }
62
63 func main() {
64     state := liner.NewLiner()
65     state.SetTabCompletionStyle(liner.TabPrints)
66     state.SetCompleter(func(line string) (ret []string) {
67         for _, c := range commands {
68             if strings.HasPrefix(c, line) {
69                 ret = append(ret, c)
70             }
71         }
72         return
73     })
74     defer state.Close()
75
76     for {
77         p, err := state.Prompt(prompt())
78         if err == io.EOF {
79             return
80         }
81         if err != nil {
82             panic(err)
83         }
84         if executeShellScript(p) {
85             state.AppendHistory(p)
86         }
87     }
88 }

```

# 4 Designing Reactive Systems

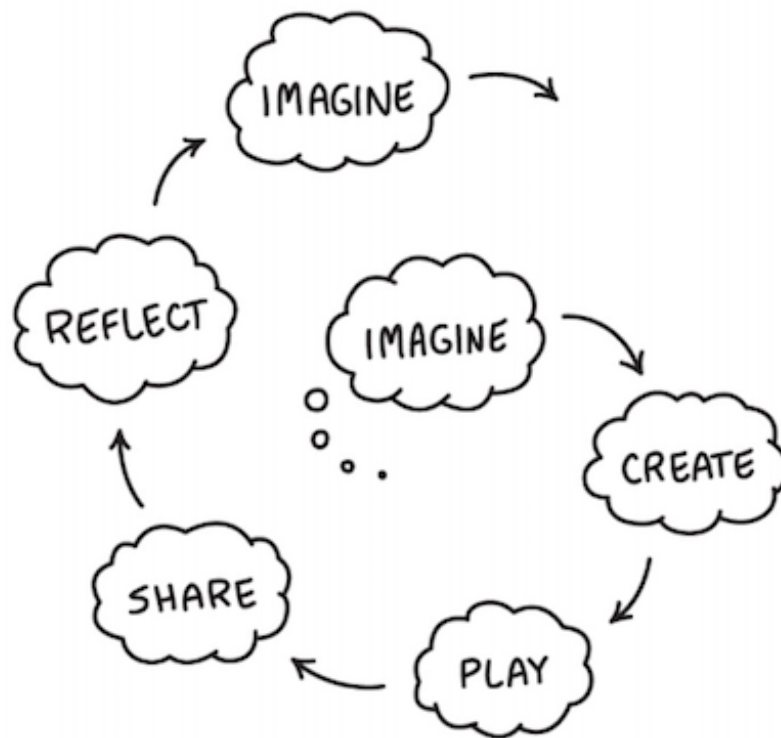
## 4.1 Approach

According to the MIT Media Lab, learning should evolve in spiraling cycles. They call it Creative Learning [Mit07]. This is how they think brains learn best:

*"...learning happens best when people are actively constructing knowledge through creative experimentation and the design of sharable objects.*

Quote by ML LEARNING INITIATIVE [ML ]

Figure 4.1: Creative Learning [Lcl]



We want to follow ideas and create prototypes to play. We have to learn from gained experiences, reflect, and write down the findings or share them in another way. Then we start a new cycle.

We should not only have the goal to create a reactive system. During the spiral cycles of prototypes development, we design, implement, and test two things in parallel:

- an example application
- a development system prototype

It is not planned to reach production-readiness for any of them, but a sufficiently deep understanding of how to achieve it.

### 4.1.1 Chat as an example application

One of the most prominent application to have in distributed client systems is the chat. We use it as an example application with its basic functionality, i.e., join, leave, and send messages from one to all. We need a bootstrap service to organize the members of the system to join.

### 4.1.2 Development system prototype

Distributed systems are about the interaction between their members, which adds complexity to the development process. We want to be able to work with multiple members running at the same time. Therefore, creating functions flexibly and call them interactively during execution is the central feature of the development system. Additionally, we need a service to coordinate the testing of multiple members of an application.

We use the development system while we are developing it. Here, we see the use case of creative learning. We should end up with a development system prototype which reflects our experiences in how to design, implement, and test reactive systems in general - ready to reused and enhanced, not at least, by Creative Learning.

The chat and the development system share the same UI.

## 4.2 Requirements

### 4.2.1 Requirements of the chat application

#### Isomorphic clients

In this thesis, the term isomorphic client depicts a client application which unites the functionality of server and client. Isomorphic clients who are free to go online and offline, and flexible concerning its network address need an entry-point to connect to the others. According to the open/closed principle<sup>1</sup>, we want the part which provides the generic and infrastructural functionality closed, and its use by the application open. So, in the end, the parts with infrastructural aspects, which are independent of the application

---

<sup>1</sup>"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" [Mey88]

should be closed for modifications, and be reusable for the applications. Furthermore, it should be possible to create new features, e.g., distributed storage or monitoring, as extensions.

### **Choreography to organize chat membership**

In a group of members of one distributed reactive system, we need permanently an answer to the question "who is currently reachable in our group." Due to the decentralized nature of the system, the members itself have to organize it as choreography. For this complex task, we will integrate existing implementations.

### **Bootstrap service**

As entry-point, we want to establish a bootstrap service which guarantees its responsiveness, being integrated into a public cloud accordingly. The service shall only provide network addresses of application members dynamically and not being a member itself.

## **4.2.2 Requirements of the development system**

### **Playing with internal functionality of client applications online**

If we are trying to get familiar with a new library or to test our ideas, we do not want to change and compile code possibly for every step we make or on different versions of the application clients. It will not be practicable. We need to act interactively from inside applications running online in parallel. The methodology to do so has to be easy to use and flexible to change.

### **Testing interaction between client applications and services**

Concerning services we want to implement APIs as libraries and test them with high code coverage for two reasons:

- to reuse the tests for new service implementations
- to help to solve problems of other parts by excluding the API as a reason for them

Additionally, end-to-end tests shall be able to use the service indirectly.

### **Testing interaction between multiple client applications**

End-to-end tests, which are using multiple client applications, has to follow an "as simple as possible" approach and, therefore, a central orchestration instance. As well, it is appropriate to implement an intermediate layer with commands to define test cases. The disadvantages of orchestration for production are not relevant in a controlled test environment.

## Tooling for logging, debugging and configuring

Of course, to make a developer feel comfortable and to work effectively, tailored tooling is a must have. Because we develop for our individual preferences here, we can follow the "you aren't gonna need it" <sup>2</sup> principle for the time being. That means, we implement the features we ourselves want to have and see, then, if we are using it and how we want them to be enhanced possibly.

## 4.3 Methodical Proceedings

To keep it simple, we start from idealized preconditions and evolve step-by-step towards a more realistic environment. To describe the scenarios we will pass, we distinguish between the information the application needs and the situations the application has to handle:

1. Complete information in an idealized world
2. Incomplete information in an idealized world
3. Incomplete information in a realistic world

### 4.3.1 Complete information in an idealized world

In the beginning, we agree to the simplest possible scenario to get an idea for the core functionality and the bounded context of the application. In practice, complete information means we assume that the user or tester, respectively, knows facts at the border of the bounded context. Additionally, we start in an idealized world environment in which we assume no faults do occur.

We will create some prototypes to accomplish the following:

- create an isomorphic client
- create the UI for the chat
- create an interactive command line
- try out different ways of communication

---

<sup>2</sup>"you aren't gonna need it" principle (YAGNI) [Yag]

### 4.3.2 Incomplete information in an idealized world

Now, we introduce the uncertainty of membership to the simplest possible scenario. In practice, this means we assume that the user or tester, respectively, has to handle real application members who have initially unknown communication data and can go online and offline at any time. Apart from that, we stay in an idealized environment in which we exclude routing problems by being in one single network, and we take a stable network for granted.

We will create some prototypes to accomplish the following:

- create a bootstrap service
- create the UI for the chat
- analyze different ways of communication

### 4.3.3 Incomplete information in a realistic world

In the final scenario, we know nothing about other members before the start of the application, and we try to react to the *Fallacies of Distributed Computing* [Arn]:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

The goal is not a professional decentralized reactive chat application. Instead, we want a complete fundamental knowledge of how to accomplish. We have no focus on UX<sup>3</sup>, test coverage, or reusability. But, we want to get the development system prototype up to a status at which we are sure that and how we can achieve these. That means that non-basic features of a professional chat application have not to be implemented if they need no new approach. i.e., we are sure how to do it.

---

<sup>3</sup>UX - User Experience [Eff09]

# 5 Programmed Prototypes

*"Our ultimate goal is to foster a world full of playfully creative people, who are constantly inventing new possibilities for themselves and their communities. "*

Quote from THE LIFELONG KINDERGARTEN GROUP OF THE MIT MEDIA LAB [Mit]

Here is an overview of the prototypes we describe in this chapter:

	Prototype	Chat Server	Group Membership	Bootstrap	Internal Command	Protocol Buffers	UI
5.1	gRPC-chat [Grpb]	First client	-	Well-known address	-	Yes	'TUI'
5.2	protobuf-tcp-chat [Prob]	First client	-	Well-known address	Yes	Yes	'TUI'
5.2	protobuf-udp-chat [Prof]	First client	-	Well-known address	Yes	Yes	'TUI'
5.3	cf-chat [Cfa]	All clients	Cloud Functions	Cloud Functions	Yes	Yes	'TUI'
5.4	liner-memberlist-chat [Linb]	All clients	'memberlist'	Cloud Functions	Yes	Yes	'liner'
5.5	libp2p-chat [Libc]	All clients	'libp2p'	Bootstrap peers	Yes	-	'TUI'

Explanation of the table columns:

1. **5.\*** - reference to the prototype section
2. **Prototype** - name of the root directory in the code repository
3. **Chat Server** - client publishing the chat messages
4. **Group Membership** - solution for the Distributed Group Membership Problem
5. **Bootstrap** - whom to ask in the first place
6. **Internal Command** - do we have internal commands in the chat
7. **Protocol Buffers** - do we use Protocol Buffers for the chat
8. **UI** - library for the UI

## 5.1 Chat With Protocol Buffers And gRPC

Code Repository: '*gRPC-chat*' [Grpb]

### 5.1.1 Design

A starting point in the Go programming domain to develop distributed systems like microservices is the combining of Protocol Buffers and gRPC. We define the data structures we want to use as so-called *Protocol Buffer message types* and the API as so-called *gRPC services*. We use a compiler to get Go code for the client and server part. Finally, we have to define what to do with incoming messages by implementing the already created Go interfaces of the server.

All applications have parts of the client and the server, and all run a chat message receiving server called *Displayer*. But, only the first application started runs the server part named *Publisher* to publish chat messages to all others.

### 5.1.2 Implementation

We define our message types and services in one .proto file [Grpd] and use the compiler *protoc* to create the Go code [Grpc].

Listing 5.1: Compile "chat-group.proto" and create "chat-group.pb.go"

```
1 $ protoc --go_out=plugins=grpc:. chat-group.proto
```

For the gRPC client sides all is prepared, but for the gRPC server interfaces we have to implement the functions.

Let's start to implement the Publisher interface [Grpf]:

- Subscribe
- Unsubscribe
- Publish



And the Displayer interface [Grpe]:

- DisplayText
- DisplaySubscription
- DisplayUnsubscription

The gRPC Publisher client interfaces are wrapped to dial the publisher [Grpl]:

- Subscribe
- Unsubscribe
- Publish
- dialPublisher

We provide a well-known address as constant configuration:

Listing 5.2: Publisher Address

```
1 // publishing service on a commonly known address
2 const (
3     serverIp    string = "localhost"
4     serverPort  string = "22365"
5 )
```

The main function gives an overview about how we start [Grph]. The interesting calls are:

1. startPublisher - try to start the publisher aware of a already running one [Grpj]
2. Subscribe - one of the wrapper functions to call the publisher for subscription [Grpk]
3. startDisplayer - start the displayer to receive chat messages from the publisher [Grpi]
4. runTUI - run the text-based UI [Grpg]

## 5.2 Chat With Protocol Buffers Via TCP Or UDP

Code Repositories: *'protobuf-tcp-chat'* [Prob] and *'protobuf-udp-chat'* [Prof]

### 5.2.1 Design

We want to evolve the chat, as an example application, and a development system parallel and together in one software.

For the next **application prototype**, we want to dismiss gRPC and implement native TCP/UDP for the communication between the isomorphic clients instead. The advantages of the gRPC regarding convenience are valid, but we want to explore more in-depth the network layer. In the last prototype, we had also not found a way to get the address of the sender of a message. So, if there is a possibility to get the sender's address, it is not a standard feature presented to the user. Therefore, we move the idea of different service methods from gRPC into Protocol Buffer's message types.

To begin implementing the **development system**, we want to introduce an internal command line in the chat. Then, we can analyze behavior and functionality more efficiently during the execution of multiple applications.

### 5.2.2 Implementation

To implement using Protocol Buffer without gRPC, we define our message types in one .proto file [Proc]. The difference between creating gRPC services and request/reply pairs inside of the messages becomes apparent by a look in the code. We see, instead of defining services, we define an enumeration of request/reply pairs used inside the messages. Before sending, we have to specify the purpose of a message, and after receiving, we have to read it and act accordingly.

Listing 5.3: Service Definitions

```
1 // Service definition for gRPC plugin to publish messages and handle subscriptions
2 service Publisher {
3     rpc Subscribe(Member) returns (Member) {}
4     rpc Unsubscribe(Member) returns (Member) {}
5     rpc Publish(Message) returns (MemberList) {}
6 }
7
8 // Service definition for gRPC plugin to display messages
9 service Displayer {
10     rpc DisplayText(Message) returns (Message) {}
11     rpc DisplaySubscription(Member) returns (Member) {}
12     rpc DisplayUnsubscription(Member) returns (Member) {}
13 }
```

Listing 5.4: Enumeration of request/reply pairs

```
1 // Services are mapped by request/reply message types
2 message Message {
3     enum MessageType {
4         // messages to handle subscriptions at the publishing service
5         SUBSCRIBE_REQUEST = 0;
6         SUBSCRIBE_REPLY = 1;
7
8         // messages to handle unsubscriptions at the publishing service
9         UNSUBSCRIBE_REQUEST = 2;
10        UNSUBSCRIBE_REPLY = 3;
11
12        // messages to publish chat messages via the publishing service
13        PUBLISH_REQUEST = 4;
14        PUBLISH_REPLY = 5;
15    }
16    MessageType msgType = 1;
17    Member sender = 2;
18    string text = 3;
19    MemberList memberList = 4;
20 }
```

Therefore, we have to implement a multiplexer function, which distributes the messages according to its message type.

handleDisplayerRequest [Prod]:

- handleSubscribeReply
- handleUnsubscribeReply
- handlePublishReply

handlePublisherRequest [Proe]:

- handleSubscribeRequest
- handleUnsubscribeRequest
- handlePublishRequest

Two maps are used to link the message type and the message handler:

Listing 5.5: Maps for message handlers

```
1 var requestActionMap = map[chatgroup.Message_MessageType] func(*chatgroup.Message,
2                                     net.Addr) error {
3     chatgroup.Message_SUBSCRIBE_REQUEST:    handleSubscribeRequest,
4     chatgroup.Message_UNSUBSCRIBE_REQUEST:  handleUnsubscribeRequest,
5     chatgroup.Message_PUBLISH_REQUEST:      handlePublishRequest,
6 }
7
8 var replyActionMap = map[chatgroup.Message_MessageType] func(*chatgroup.Message) error {
9     chatgroup.Message_SUBSCRIBE_REPLY:      handleSubscribeReply,
10    chatgroup.Message_UNSUBSCRIBE_REPLY:     handleUnsubscribeReply,
11    chatgroup.Message_PUBLISH_REPLY:         handlePublishReply,
12 }
```

If we introduce an internal command line to the chat, we need a distinction between standard chat messages and commands to be executed. We use a leading backslash for it:

Listing 5.6: Distinguish a message from a command

```
1 // Distinguish between command and chat mode by '\'-prefix
2 if strings.HasPrefix(input, "\\") {
3
4     // Interpret "input" as command
5     executeCommand(input)
6
7 } else {
8     // Send "input" to publish
9     Publish(input)
10 }
```

We split the command line into words and select the appropriate function by the first word. The rest of the command line represent the arguments for the function to be called:

Listing 5.7: Process command line

```
1 // Trim prefix and split string by white spaces
2 commandFields := strings.Fields(strings.Trim(commandline, "\\"))
3
4 ...
5
6 // Switch according to the first word and call appropriate function with the rest as
7                                     arguments
8 switch commandFields[0] {
9
10 case "list":
11     list(commandFields[1:])
12 ...
13
14 }
```

Using UDP as the transport protocol, only minor changes have to be done, e.g., to the displayingService [Prog]. We take care of the buffer size according to the IP protocol [Rfcb].

#### Listing 5.8: Optimize buffer size

```
1 // The maximum safe UDP payload is 508 bytes.
2 // This is a packet size of 576 (IPv4 minimum reassembly buffer size),
3 // minus the maximum 60-byte IP header and the 8-byte UDP header.
4 bufferSize = 508
```

To dial a connection needs only a change of the network parameter.

#### Listing 5.9: Dial a UDP connection

```
1 // Connect to publishing service
2 conn, err := net.Dial("udp", publishingService)
```

## 5.3 Chat Using Cloud Functions As List Of Members Service

Code Repository: '*cf-chat*' [Cf a]

### 5.3.1 Design

The prototype is the first one to handle incomplete information in an idealized world, i.e., at the start, no chat member has address information about any other, and network failures are not taken into account.

Any member of the chat needs an entry point to join. We introduce a so-called *List Of Members service*, which let a new member join and provides a list of already joined members in return. The service approach is a raw model for other decentralized applications. In addition to the chat, as an example application, and the development system, we recognize a bootstrap service as the third central aspect.

The List Of Members as **bootstrap service** needs to be responsive to guarantee the reactivity of the whole system. Operating on Google Cloud Functions [Clo], a runtime environment as a service gives the reliability needed. Google Cloud Firestore [Fir], a NoSQL database, provides the persistence for the list of members. We implement an API for the service and test API and service completely.

The **application prototype** communicates the chat messages directly from the sender to all the recipients. It starts by requesting the *List Of Members service*. Using the returned list of already joined members, it broadcasts a subscription message to all other members. Now, all members have the same complete information. If a member wants to leave, it sends an unsubscription message to the service and all other members. Of course, this works only in an idealized world without communication problems.

Concerning the **development system**, we implement the complete API of the *List Of Members service* as a set of commands for the internal command line. To evolve an approach we call Distributed Testing, we extend existing parts to be able to test processes between separated network units.

## 5.3.2 Implementations

### List Of Members Service

The service, as we see it, is divided into two parts - the Cloud Functions and the library or API, respectively, for the chat to use.

Cloud Functions:

We have the struct for the data of the service to be stored in JSON format [Cf d]:

Listing 5.10: Data structure for a member

```
1 // IPAddress is the struct for the collection
2 type IPAddress struct {
3     Name      string 'firestore:"name,omitempty"'
4     Ip        string 'firestore:"ip,omitempty"'
5     Port      string 'firestore:"port,omitempty"'
6     Protocol  string 'firestore:"protocol,omitempty"' // "tcp" or "udp"
7 }
```

And the functions deployed as Cloud Functions:

- Subscribe [Cf e]
- Unsubscribe [Cf f]
- List - only necessary for testing [Cf b]
- Reset - only necessary for testing [Cf c]

Every function is stored in a file and has to be deployed separately, e.g., the Subscribe function:

Listing 5.11: Deploy "Subscribe" as Cloud Function

```
1 gcloud alpha functions deploy subscribe --region $GCP_REGION \
2 --entry-point Subscribe \
3 --runtime go111 \
4 --trigger-http
```

## API:

The API provides an appropriate data structure with the same struct as the service[Cf s] and a data structure to store it [Cf u]:

Listing 5.12: Create a 'memberlist'

```
1 // Create returns the memberlist for oneself
2 func Create(self *IpAddress) (*Memberlist, error) {
3
4     serviceUrl := os.Getenv("GCP_SERVICE_URL")
5     if serviceUrl == "" {
6         return nil, fmt.Errorf("GCP_SERVICE_URL environment variable unset or missing")
7     }
8
9     id := uuid.NewV4().String()
10
11     ipAddresses := make(map[string]*IpAddress)
12     ipAddresses[id] = self
13
14     return &Memberlist{
15         ServiceUrl: serviceUrl,
16         Uuid:       id,
17         Self:       self,
18     }, nil
19 }
```

And the functions for the chat:

- Subscribe [Cf w]
- Unsubscribe [Cf x]
- List [Cf t]
- Reset [Cf v]

## Chat

The chat uses the Protocol Buffer message types from the former prototype [5.2.2] to define the chat's core struct and to initialize the chat [Cf n]:

Listing 5.13: Initialize the chat

```
1 // Chat is the core struct for the chat
2 type Chat struct {
3     self          *chatgroup.Member
4     memberlist    []*chatgroup.Member
5     message       *chatgroup.Message
6 }
7
8 // CreateChat returns a new chat instance
```

```

9| func CreateChat(name, ip string) *Chat {
10|
11|     self := &chatgroup.Member{
12|         Name: name,
13|         Ip:    ip,
14|     }
15|
16|     var memberlist []*chatgroup.Member
17|
18|     return &Chat{
19|         self:    self,
20|         memberlist: memberlist,
21|         message: &chatgroup.Message{
22|             MsgType: chatgroup.Message_SUBSCRIBE_REQUEST,
23|             Sender:    self,
24|         },
25|     }
26| }

```

We want to show the other changes in the chat using a **session's lifecycle** [Cf o]:

1. create the list for the chat members [Cf j]
2. start chat listener to get its network address [Cf p]
3. send subscription request to the List Of Members service [Cf w]
4. initialize the chat [Cf n]
5. send complete subscriber list to all chat members [Cf h]
6. send chat message to all other members [Cf i]
7. all other members receive the message [Cf k]
8. send unsubscribe request to all chat members and the List Of Members service [Cf g]

All the communication between the chat members uses the data structure and the multiplexer functions similar to the last prototype [5.2.2].

### 5.3.3 Testing

Testing contains two parts. We **test the API** of the List Of Members service up to high test coverage and, thus, reliability for further development. We start to test the interaction between multiple chat members using an approach we call **distributed testing**.

#### API Tests

Having the two additional functions List and Reset, which we do not use in the chat itself, we can define tests to subscribe and unsubscribe using the '*testing*' package [3.2.4] from Go's standard library.



## Distributed Testing

For distributed testing we introduce a parallel message communication using the same methodology. Then we can test all members currently in the chat and its communication functionality.

We have two ways implemented:

- TEST\_PUBLISH is a two hop round trip over every chat member to every chat member each and then back to the tester.
- TEST\_CMD is a simple call for commands to every chat member sending the results back to the tester.

First, we need new message types [Cf r]:

Listing 5.14: Request/reply pairs for testing

```
1 // messages to test publishing chat messages
2 TEST_PUBLISH_REQUEST = 6;
3 TEST_PUBLISH_REPLY = 7;
4
5 // messages to test chat commands
6 TEST_CMD_REQUEST = 8;
7 TEST_CMD_REPLY = 9;
```

The chat needs two new handlers to response at the tests:

- handleTestPublishRequest [Cf m]
- handleTestCmdRequest [Cf l]

The rest we implement in the test file, i.e., the listener, the reply handlers, and the test functions itself [Cf q].

To execute the test, we open as many terminals as wanted, at least two. Then, we do the following in each terminal:

Listing 5.15: Start a chat terminal

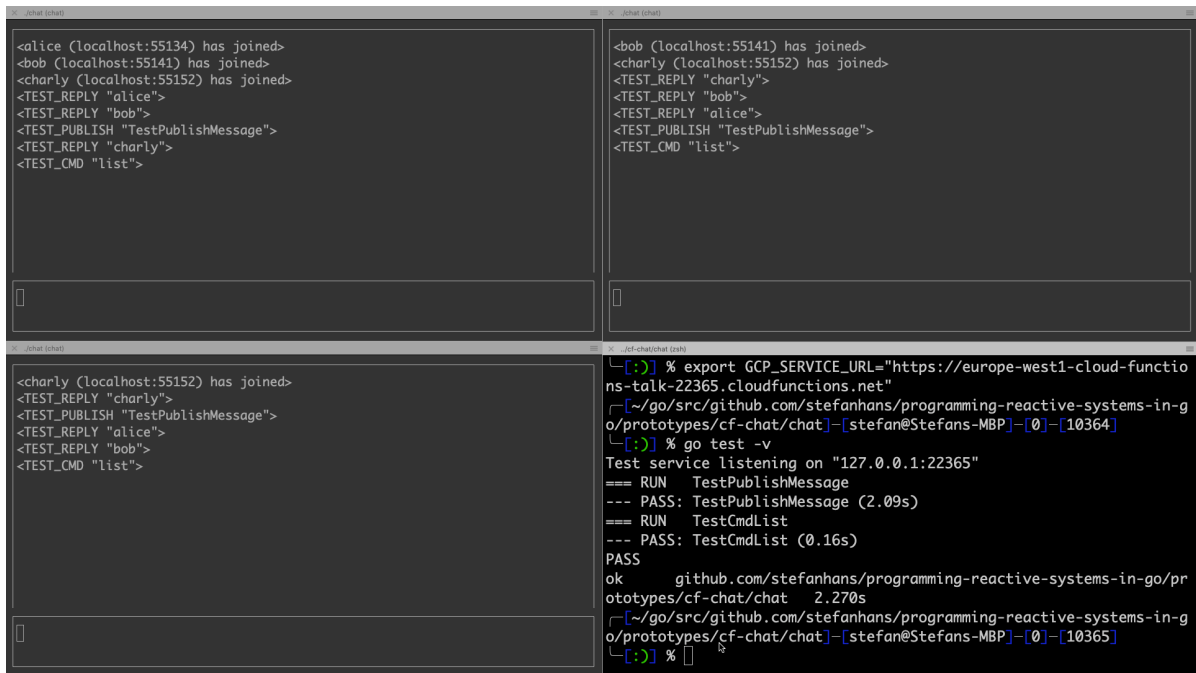
```
1 export GCP_SERVICE_URL="https://europe-west1-XXX.cloudfunctions.net"
2
3 ./chat alice localhost
```

Now, we can do the distributed testing:

Listing 5.16: Start the test

```
1 export GCP_SERVICE_URL="https://europe-west1-XXX.cloudfunctions.net"
2
3 go test -v
```

Figure 5.1: Screenshot of distributed testing



```
<alice (localhost:55134) has joined>
<bob (localhost:55141) has joined>
<charly (localhost:55152) has joined>
<TEST_REPLY "alice">
<TEST_REPLY "bob">
<TEST_PUBLISH "TestPublishMessage">
<TEST_REPLY "charly">
<TEST_CMD "list">

<bob (localhost:55141) has joined>
<charly (localhost:55152) has joined>
<TEST_REPLY "charly">
<TEST_REPLY "bob">
<TEST_REPLY "alice">
<TEST_PUBLISH "TestPublishMessage">
<TEST_CMD "list">

<charly (localhost:55152) has joined>
<TEST_REPLY "charly">
<TEST_PUBLISH "TestPublishMessage">
<TEST_REPLY "alice">
<TEST_REPLY "bob">
<TEST_CMD "list">

[[:~]] % export GCP_SERVICE_URL="https://europe-west1-cloud-functio
ns-talk-22365.cloudfunctions.net"
[~/go/src/github.com/stefanhans/programming-reactive-systems-in-g
o/prototypes/cf-chat/chat]-[stefan@Stefans-MBP]-[0]-[10364]
[[:~]] % go test -v
Test service listening on "127.0.0.1:22365"
=== RUN   TestPublishMessage
--- PASS: TestPublishMessage (2.09s)
=== RUN   TestCmdList
--- PASS: TestCmdList (0.16s)
PASS
ok      github.com/stefanhans/programming-reactive-systems-in-go/pr
ototypes/cf-chat/chat    2.270s
[~/go/src/github.com/stefanhans/programming-reactive-systems-in-g
o/prototypes/cf-chat/chat]-[stefan@Stefans-MBP]-[0]-[10365]
[[:~]] %
```

We see in the terminals of each chat member how the other members join the chat. Then, we start the test in another terminal to see the results. In the chat terminals we see the message types of the test and the message text in the prompt. We have one test sending a message to all and getting replies as acknowledge. One test is executing the *'list'* command and checking the return.

## 5.4 Pre-Chat Using 'memberlist'

Code Repository: '*liner-memberlist-chat*' [Linb]

### 5.4.1 Design

The prototype is the first one to handle incomplete information in a realistic world, i.e., at the start, no chat member has address information about any other, and network failures are taken into account. For this, we need three layers: the bootstrap layer, the memberlist layer, and the chat layer. Each layer has its network addresses for the time being.

To handle the Distributed Group Membership problem, we use the '**memberlist**' library [Mem], which implements the SWIM++ [Abh] protocol. For the exploration of this library, we introduce the '**liner**' library [Lina], a line editor with command completion and history. So, regarding the UI, we move the chat functionality out of focus and replace the 'gocui' [Gocb] by the 'liner' library. Then, we can more conveniently use our internal command line. On the other hand, the application starts from scratch and all steps to get the chat up and running has to be done on the command line.

We implement a true **bootstrap service**, which holds a limited number of members to contact at the beginning. These members receive requests to join the distributed group of members provided by 'memberlist.' Additionally to the implementations of a bootstrap server as Cloud Functions and an API, we introduce a bootstrap server providing the same API, but, can be used for offline development running on localhost.

Concerning the internal commands, the prototype shows **enhanced tooling** to log, debug, and to administrate configurations, and we establish the **execution of scripts** called from inside the running application.

The **bootstrap service** holds a limited list of members, i.e., their network addresses of the memberlist layer. A new member will be added until the limit is reached. If a member is on the bootstrap list and is leaving or left the memberlist layer, it will be removed. The memberlist layer requests the removal, by the leaving member or by another member who noticed that a member left accidentally. The list can be refilled if requested.

An API completes the bootstrap service. Both have to be tested thoroughly.

The deployment of the bootstrap service on the Cloud Functions runtime is perfect for production, but, to work on the development of the other parts, we have to be online. Therefore, we implement an HTTP server, which stores the data on the file system, for offline development.

On the '**memberlist**' layer [Mem], regularly, an exchange of synchronization information happens. Other layers can use it as piggyback transport and get events notification about changes to the list of members. In our design, we separate the chat layer from the memberlist layer, i.e., the chat messages are not sent on the memberlist layer, and

changes on the memberlist layer do not directly influence the chat layer and vice versa. We have to implement it explicitly.

The **enhanced tooling** has the following features already, or they can be implemented quickly:

- interactive logging, i.e., we can switch logging and log files freely at any time during the execution
- interactive debugging, i.e., any structs or information can be saved interactively for later investigation
- interactive configuration, i.e., runtime configurations can be saved and loaded, from the command line

The **execution of scripts** means we can write simple scripts using, built-in and individual, function calls, and execute them from the command line.

## 5.4.2 Implementation

### The Bootstrap Layer

The bootstrap layer has two parts - the bootstrap service and the API library, respectively, used by the other layers. Both service implementations [Lind] [Line] and the API [Linc] use the same data structure:

Listing 5.17: Data structure for the bootstrap service

```
1 // Peer is the struct for the collection
2 type Peer struct {
3     ID      string 'json:"id,omitempty"' // UUID
4     Name    string 'json:"name,omitempty"' // chat name
5     Ip      string 'json:"ip,omitempty"'
6     Port    string 'json:"port,omitempty"'
7     Protocol string 'json:"protocol,omitempty"' // "tcp" or "udp"
8     // todo get rid of unused field status
9     Status  string 'json:"status,omitempty"'
10    Timestamp string 'json:"timestamp,omitempty"' // Unix time in seconds
11 }
12
13 // Config has the configuration of the bootstrap service.
14 type Config struct {
15     MaxPeers      int 'json:"maxpeers,omitempty"'
16                                     // Max number of bootstrap peers to be saved
17     MinRefillCandidates int 'json:"minrefillcandidates,omitempty"'
18                                     // Number used to decide peer send refill request
19     NumPeers      int 'json:"numpeers,omitempty"'
20                                     // Number of bootstrap peers
21 }
22
23 // BootstrapData is a complete data structure
24 type BootstrapData struct {
25     Config Config
26     Peers  map[string]*Peer
27 }
```

The service has the following functions:

- Join - inserts into the list up to a specified number and returns the list [Linw]
- Leave - removes the peer from the list, if appropriate [Linx]
- List - returns the list of bootstrap members [Linac]
- Refill - is like a Join with additional check for replacing a member from the list according to a 'Least Recently Used' [Sta] like eviction policy [Linz]
- Ping - just returns "OK" [Liny]
- ConfigUpdate - sets the limit for the number of bootstrap members [Linab]
- Reset - empties the list of bootstrap members [Linaa]

Here the handler function from the HTTP server [Lins]:

Listing 5.18: Run the service with its handlers

```
1 // Run starts the service.
2 func Run() {
3
4     http.HandleFunc("/join", Join)
5     http.HandleFunc("/leave", Leave)
6     http.HandleFunc("/refill", Refill)
7     http.HandleFunc("/list", List)
8     http.HandleFunc("/reset", Reset)
9     http.HandleFunc("/ping", Ping)
10    http.HandleFunc("/config", ConfigUpdate)
11
12    log.Fatal(http.ListenAndServe(":8080", nil))
13 }
```

## The 'memberlist' Layer

The 'memberlist' layer is a gossip layer, on which all members communicate with each other in an infectious style, i.e., not everyone to all others directly. It is like, you tell it a few others, they do the same, and in the end, everybody knows it. On this layer, regular heart-beat-like communication takes place. Additionally, we can communicate explicitly, using delegates on the layer. And we can get notifications about changes in the member list by event delegates.

There are two interfaces of the 'memberlist' layer we implement.

- 'Delegate' to hook into the layer and send messages to all members. Here the function doing the member registration [Linq]:

### Listing 5.19: Handle member registration

```

1 // NotifyMsg is called when a user-data message is received.
2 func (d *delegate) NotifyMsg(b []byte) {
3     if len(b) == 0 {
4         return
5     }
6
7     var newChatMember *chatmember.Member
8
9     if err := json.Unmarshal(b, &newChatMember); err != nil {
10        return
11    }
12
13    switch newChatMember.MsgType {
14    case chatmember.Member_JOIN:
15        mtx.Lock()
16        chatMembers[newChatMember.Name] = newChatMember
17        mtx.Unlock()
18    case chatmember.Member_LEAVE:
19        mtx.Lock()
20        delete(chatMembers, newChatMember.Name)
21        mtx.Unlock()
22    }
23 }

```

- 'EventDelegate' is used only to receive notifications about members joining or leaving. Here the function triggered by a leaving member [Linp]:

### Listing 5.20: Handle left member

```

1 // NotifyLeave is invoked when a node is detected to have left.
2 func (d *eventDelegate) NotifyLeave(node *memberlist.Node) {
3
4     // Get the current list of bootstrap peers
5     bootstrapPeers = bootstrapApi.List()
6
7     // Loop for the bootstrap server possibly left
8     var leftBootstrapNode string
9     for k, v := range bootstrapPeers {
10        if v.Name == node.Name {
11            leftBootstrapNode = k
12        }
13    }
14
15    // Update bootstrap servers service and locally
16    if leftBootstrapNode != "" {
17        bootstrapApi.Leave(leftBootstrapNode)
18        delete(bootstrapPeers, leftBootstrapNode)
19    }
20
21    // Request the bootstrap servers service to refill with the requesting node
22    bootstrapConfig = bootstrapApi.Refill()
23 }

```

## The 'chat' Layer

A new chat member joins the chat by starting the chat listener [Linn]. Implicitly it broadcasts a message on the 'memberlist' layer [Linj], which is processed from its recipients [Linq].

To send a chat message, it loops over the chat members to send directly to each one [Linu].

A leaving chat member [Linl] stops the chat listener, empties its list of chat members, and broadcasts a message on the 'memberlist' layer [Linm], which is processed from its recipients [Linq].

If an event about a member who left appears on the 'memberlist' layer [Linp], it checks the freshly loaded list of bootstrap members for needed removal of the member [Link]. If the list is not full, it sends a refill request to add itself as a bootstrap member [Linu]. Finally, it removes the member from the local list of chat members, if it cannot be pinged [Lini].

## Enhanced Tooling

We have interactive logging, which allows to start and stop logging into a specified file during the execution at any time [Linu].

We can save [Lint] and load [Lino] configurations in JSON format on, respectively from, the filesystem. For the time being, it is implemented only for 'memberlist' configurations.

## Creating Commands

To implement new commands conveniently, here are the steps:

1. use the 'play' command [Linr] to implement ad-hoc without creating and integrating a new function, e.g., by calling existing functions and displaying related information
2. adapt the function to be used as a command by using a wrapper or changing the argument specification of the function
3. add the command as key and a short description as its value in the 'commands' map [Ling]
4. add a case statement for the command in the 'executeCommand' function [Linh]
5. try your command on the internal command line

## The Execution Of Scripts

We can execute lines of commands from a file by calling 'execute' with the filename as an argument. Additionally, we provide some built-in function as we know it from shell scripting:

- echo
- sleep

## Pre-Chat

The application starts from scratch, and we have to initialize a chat functionality on the command line as follows:

1. *memberlistconfigure* creates a memberlist configuration
2. *memberlistcreate* creates the memberlist specified by the configuration
3. *bootstrapjoin* joins calling peer to bootstrap peers
4. *memberliststart* starts broadcasting between the members
5. *memberlistjoin* joins bootstrap peers to memberlist
6. *chatjoin* start chat listener and join the chat
7. *msg hi* sends a message to all chat members

To quit the application gently, you do the following:

1. *chatleave* stops the chat listener and broadcasts a leave message on the 'memberlist' layer
2. *memberlistleave* again broadcasts a leave message on the 'memberlist' layer (optional)
3. *memberlistshutdown* stops being a member of the 'memberlist' network
4. *bootstrapleave* sends a leave request to the bootstrap service
5. *quit* exits the application

To boot up to different layers we can use scripts:

- *execute bootstrap.txt*
- *execute memberlist.txt*
- *execute chat.txt*

If we kill the application hard, the 'memberlist' layer will notice the changed situation and events trigger the appropriate action on the other layers to keep the system healthy.



## 5.5 Pre-Chat Using 'libp2p'

Code Repository: '*libp2p-chat*' [Libc]

The purpose of this prototype is to experiment with the 'libp2p' stack [Libb], including other projects from Protocol Labs [Proa] as needed. The goal is to use our development system approach to find out how usable the libraries are for our chat application.

Here the steps:

1. we fork the repository of the library we are interested in [Stea]
2. we create a subdirectory with the development system [Steb]
3. we add the commands for the library calls and try them out
4. we implement a prototype [Libc]

Concerning the development system, we find the prototype very useful to explore libraries for decentralized systems. The concept of the analysed library is promising but challenging as well.

The implemented pre-chat application [Libc] proves that we can create a chat using the libraries. But, it has worked out that we need a more profound understanding of how to do it. Without further investigations of us, we can only estimate how reactive the final application would be.

## 5.6 Findings

What did we learn after implementing these prototypes?

### 5.6.1 Simplicity

All implementations concerning distributed and decentralized systems have a high risk to get too complex and hard to handle. Therefore, we should always search for solutions being as simple as possible. If you are facing real problems implementing an approach, step back and look for a more straightforward solution. This is known as the "Keep It Simple Stupid" (KISS) principle [Kis].

As well, you should avoid implementing features which you think you will probably need later on. If you need the feature later, you usually have a better understanding of it and a more stable application in its other parts. But, often you will not need it at all, which is known as the "You Ain't Gonna Need It" (YAGNI) principle [Yag].

I advise you to prefer readability over efficiency [Abe66]. If code repetition makes code easier to understand, it is better to repeat it than to save some lines of code. Similarly, I would be cautious with segregation of functionality into libraries or introducing interfaces

for more flexibility by exchanging parts connected by loose coupling. According to the KISS and YAGNI principles, you should do these refactoring, if you really need it and if the complexity is still acceptable or it can be hidden from the later developer.

### 5.6.2 Services

Developing, testing and operating of decentralized systems need mostly some centralized functionality which you should encapsulate as a microservice. Various protocols like HTTP/S, UDP or TCP, and data-serialization formats like JSON, XML or Protocol Buffer [Proh] are applicable for the task. You should take the following into account, if you choose between different protocols and data formats to handle communication:

- current and future applications as consumer

If the service is for a large variety of applications, I would prefer HTTP as protocol and JSON as data format. On the internet, HTTP is the most common application protocol, and JSON probably the most used data-serialization format.

- performance requirements

A service having high-performance requirements should use Protocol Buffer over TCP or UDP.

- data creation and processing

Plain text can be used if the application and the service create and process data directly as strings or words, respectively.

- human-readability

If nothing stands against it, data, readable for humans, have their obvious advantage.

You should see the service and its API as a pair and test the service via API. We can implement and test a microservice and its API locally. Later we can deploy the service to a so-called serverless runtime in a public cloud.

### 5.6.3 Command Line

In many cases, it is a great help if you can use a command line in the running application. You can call functions of your application while it is online connected with other peers. A big plus is the flexibility. You can now act during development without always compiling and executing again. We can use it in different ways:

- to investigate and get familiar with external libraries
- to implement and enhance functionality
- to test behavior
- to debug code

To enhance this, we can introduce additional features like scripting, advanced logging, or saving and loading of configurations.

## 6 The Final System Prototype

Developing decentralized systems is a complex project. Without a central instance which is always available we face general problems:

- who to contact in the first place
- which peers are currently online

To experiment with different libraries and to gain experience about the subject, it is useful to have a command line running in an online connected peer. A chat as an example application gives us a realistic use case layer.

With this basic understanding, we started to develop the prototypes, and, step-by-step, we ended up with the following two prototypes, which we combined as our final one:

- Chat using Cloud Functions as List of Members service [5.3]  
We take the text-based user interface, and the experience of the testing approach, which we discarded.
- Pre-Chat Using 'memberlist' [5.4]  
We take most of it, except the command-line interface<sup>1</sup>.

Finally, we added a multi-client testing framework prototype to it.

---

<sup>1</sup>Integration of the command-line interface into the text-based user interface was only partially successful.

Here we see the architectures describing the development system and the chat application:

Figure 6.1: Architecture of implemented development system

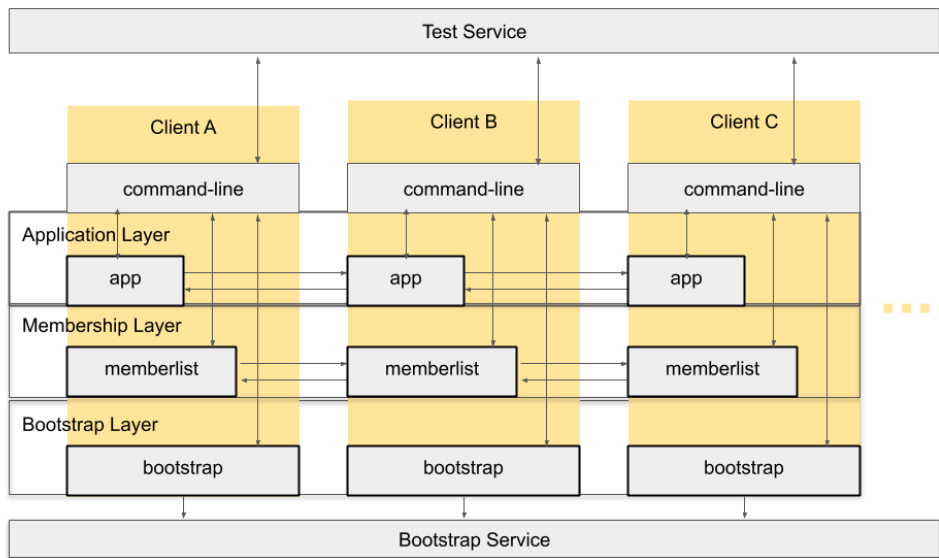
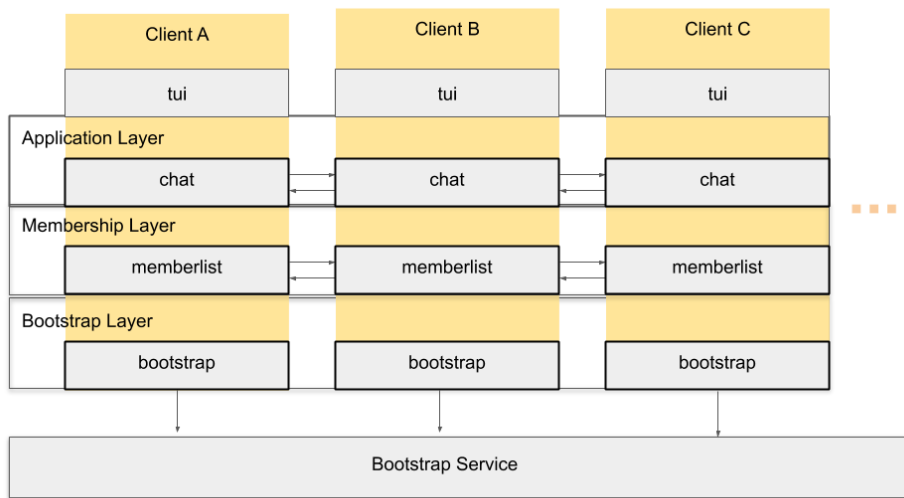


Figure 6.2: Architecture of implemented chat application



## 6.1 Design

The design shows a multi-layered, isomorphic client using services. We combine this with a UI for the application, and for developing and testing.

Here the major features we need:

- command line interface within the running peer
- chat application
- distributed group membership protocol
- bootstrap data service

In our prototype, the application and the development system share the same user interface. In other scenarios, we could use different methodologies [7.5].

### 6.1.1 Bootstrap Service

The bootstrap service provides information about which peer to contact for joining the decentralized overlay network. An API completes the bootstrap service. We tested the API against two different backends:

- a HTTP server running on localhost
- GCP Cloud Functions

There is only one JSON Data Object holding the configuration and the list of bootstrap peers. Here is an example:

Listing 6.1: Bootstrap data example

```
1 {
2   "Config": {
3     "maxpeers": 2,
4     "minrefillcandidates": 2,
5     "numpeers": 2
6   },
7   "Peers": {
8     "71c3c0ea-e1bd-4112-97ac-1cfееaa07bc3": {
9       "id": "71c3c0ea-e1bd-4112-97ac-1cfееaa07bc3",
10      "name": "bob-Stefans-MBP-94d785ce-734a-11e9-a0c2-acde4
11          8001122",
12      "ip": "127.0.0.1",
13      "port": "56471",
14      "protocol": "tcp",
```

```

14     "status": "0",
15     "timestamp": "1557509977"
16 },
17 "a559fd29-0410-4cac-b1f1-109a9513f22f": {
18     "id": "a559fd29-0410-4cac-b1f1-109a9513f22f",
19     "name": "alice-Stefans-MBP-939dbaf2-734a-11e9-847f-
        acde48001122",
20     "ip": "127.0.0.1",
21     "port": "56461",
22     "protocol": "tcp",
23     "status": "0",
24     "timestamp": "1557509975"
25 }
26 }
27 }

```

The interface of the backends is quite straightforward and returns the JSON object - except ping which returns only "OK". Here is the function containing all HTTP handlers:

Listing 6.2: Run the service with its handlers

```

1 // Run starts the service.
2 func Run() {
3     http.HandleFunc("/join", Join)
4     http.HandleFunc("/leave", Leave)
5     http.HandleFunc("/refill", Refill)
6     http.HandleFunc("/list", List)
7     http.HandleFunc("/reset", Reset)
8     http.HandleFunc("/ping", Ping)
9     http.HandleFunc("/config", ConfigUpdate)
10
11     log.Fatal(http.ListenAndServe(":8080", nil))
12 }

```

Only *refill* needs some explanation. As configured by *maxpeers*, we store up to a maximum of peers. If one of these bootstrap peers is leaving the network, all others are triggered to send a request to refill. The server chooses the peer with the oldest timestamp or uptime, respectively, to complete the list [Sta].

### 6.1.2 Distributed Group Membership Protocol Layer

For the distributed group membership protocol layer, we use the 'memberlist' library as SWIM++ protocol implementation. It provides each peer with a maintained list of all other online peers. As a gossip protocol, it is continuously exchanging data between the peers. We use the layer to exchange communication data of the application layer, and the other layers get informed about changes in the group membership. The bootstrap service provides information to initially access to it.

### 6.1.3 Integrated Command Line Interpreter

The frontend is a terminal UI with an integrated command line. It gives us flexible, interactive access to the internals of the running client. We can now, step-by-step, investigate the libraries and experiment freely and being open to failure. We gain experience in how different components interact with each other in an online network of peers. We try to have seamless transitions from getting familiar, developing, testing, and being production-ready. Additional to some internal CLI<sup>2</sup> features, we offer script execution and multi-client testing.

### 6.1.4 The Chat As Example Application

A terminal chat is the example application for the isomorphic client. The functionality of the chat is simplistic:

- join the chat
- send a message to all other peers
- leave the chat

The chat shows members who joined, are leaving or left the chat. If a client left unintentionally, this will be displayed as well.

The initialization of the chat implies all three layers as follows:

1. initialize the 'memberlist' layer to get an address by which your node is listening
2. request to join the bootstrap service with your 'memberlist' address
3. enter the 'memberlist' by contacting the addresses returned from the bootstrap service
4. initialize the chat layer by broadcasting the information of the started listener on the 'memberlist' layer

A proper leaving from the chat propagates the information about it accordingly on all three layers. The 'memberlist' layer recognizes an accidental disconnecting from the chat and informs the other layers.

---

<sup>2</sup>Command-line interpreter [Ker84]



### 6.1.5 Multi-Client Testing

Aside from regular testing of the APIs, we want to test the application in an end-to-end fashion with multiple clients running in parallel. For this, we need a coordinating instance and a practical way to write tests. Here are the main ingredients of the design:

- a wrapper around the integrated command execution for test commands
- a test service providing the test commands
- an integrated test service API
- filter to collect events from the application, e.g. received messages in the message view of the UI
- additional commands to execute the test and all related actions

Now, you can write tests using all internal commands and setting a filter to catch expected events. The commands and filters are assigned to the client peer by the name as the first word of each line. The coordination is the serial execution of commands as ordered by the lines of the test script. The clients send back the test results to the server. The test server provides a summary after the test. The design allows to enhance the multi-client testing with variables, new filter types, and data comparison.

## 6.2 Implementation

The implementation consists of the parts, which are logically separated, but the code is not yet segregated<sup>3</sup>. It has defined and tested service APIs. The chat application, especially the UI, provides no good UX.

On the other hand, it comes with great flexibility to serve as a development framework for decentralized applications. And its purpose, to prove decentralized systems can be reactive, it supports perfectly.

### 6.2.1 Bootstrap API

The bootstrap API [Fina] implements functions to send strings as POST parameters to the service and loads the returned results into appropriate data structures. It uses an environment variable [Finb] for the connection information of the service. Global variables [Fins] ensure that there is only one API instance and no redundant bootstrap information:

---

<sup>3</sup>This work has to be done to professionalize the development system but was not needed for the thesis. The same is valid for possible interfaces or APIs.

Listing 6.3: "chat/bootstrap.go" using the bootstrap API to join

```

1 var (
2     bootstrapApi *bootstrap_data_api.BootstrapDataAPI
3     bootstrapData *bootstrap_data_api.BootstrapData
4 )
5
6 func initializeBootstrapApi() {
7
8     if mlist == nil {
9         displayError("cannot create bootstrap API without local node data of memberlist")
10        return
11    }
12
13    bootstrapApi, err = bootstrap_data_api.Create(&bootstrap_data_api.Peer{
14        Name:      mlist.LocalNode().Name,
15        Ip:        mlist.LocalNode().Addr.String(),
16        Port:      fmt.Sprintf("%d", mlist.LocalNode().Port),
17        Protocol:  "tcp",
18    })
19    if err != nil {
20        displayError("failed to create bootstrap peer API", err)
21    }
22 }
23
24 func joinBootstrapPeers(arguments []string) {
25
26     if bootstrapApi == nil {
27         initializeBootstrapApi()
28         if bootstrapApi == nil {
29             return
30         }
31     }
32
33     bootstrapData = bootstrapApi.Join()
34
35     displayBootstrapData(bootstrapData)
36 }

```

## 6.2.2 Bootstrap Server

The bootstrap server [Find] was used for development purpose only. As it is running on localhost, we do not need to be online. It implements precisely the same interface, and the same API can be used. For production, you can deploy it as a container image onto a platform like Kubernetes, and it will be a reactive service.

## 6.2.3 Bootstrap Cloud Functions

For production, we use an implementation of the bootstrap service as Cloud Functions [Finc] using *Firestore* [Fir] for storage. It comes with all the advantages of running serverless concerning scale, resilience, and cost benefits<sup>4</sup>.

---

<sup>4</sup>"2 million invocations per month free" - Google Cloud Platform Free Tier [Gcp]

## 6.2.4 'memberlist' Integration

We configure, initialize, and administrate the 'memberlist' with the appropriate functions [Finn] and use broadcast functionality [Fine] and delegates [Finp] [Fino] for our purposes. It broadcasts data about chat members in Protocol Buffer format and receives them as delegates. Additionally, we receive event delegates about changes of the group membership itself. Currently, we use only the default local configuration [Fing]. To use it on the internet, we have to overcome routing problems via NAT<sup>5</sup>. Cutting-edge network stacks like 'libp2p' solve these problems in an automated way [Go b].

## 6.2.5 Terminal UI

As UI, we use an external library for terminals. We have two views - one to write the messages and commands [Finu], and one to display the received messages and the command output [Finv]. Unfortunately, we can not scroll the views. The displayed output is not thread-safe, so, we can not be sure the order of writing output is displayed correctly. For our purpose, it is acceptable. Finally, the chat as an application, as well as, the UI are just examples. Text-based user interfaces are not widely accepted anyway [7.5].

## 6.2.6 Chat

The chat is tightly connected with the 'memberlist' layer and the terminal UI. Joining [Fink] and leaving [Finl] the chat is done by the 'memberlist'. Sending [Finr] and receiving [Finw] chat messages is done in the chat's communication layer using the list of online peers maintained by 'memberlist'. The messages are input and output in the two views of the terminal UI.

## 6.2.7 Command Line

The command line is implemented as a rudimentary parser [Fint] which parses all lines with a leading front-slash. The first word of the command line is interpreted as a command and the other words as arguments. The commands are processed by a switch to find a proper internal function to be executed. Additionally, we have other internal feature commands for logging [Finf], to run scripts [Finh], to save [Finq] and load [Finn] configurations, and a command line help [Finj]. A separate set of commands is for testing purposes. We can extend the set of commands to chat application related ones. Finally, the commands related to development can be made inaccessible by default.

## 6.2.8 Test API

The test API is still an integrated part of the client. It sends requests to the test server in a loop [Fini] to have always the current test command queue. If the next command is

---

<sup>5</sup>Network address translation [Rfca]

for the client, the command will be executed. The client sends the result to the server and, if successful, the server removes the line with the run command from the queue. If the queue is empty, the client leaves the loop, and the server prepares the test summary [Finy].

### 6.2.9 Test Server

The test server is implemented as an HTTP server running on localhost [Finz]. It will not be needed in production. The data exchange with the client uses text strings and JSON format. It reads the test script at the initialization of a test run [Finx], holds all data during the run in global variables [Finab], and writes all to the filesystem before the next run [Finaa].

## 6.3 Testing

Testing the interactions of isomorphic clients is complex. Therefore, the testing framework has to be as reliable and straightforward as possible.

- provide a test file with each line defining one command for one client
- use a command to set a text filter on the terminal view of the chat messages

Now, you can make end-to-end tests like this:

Listing 6.4: Example of a test file

```
1 alice init
2 bob init
3 charly init
4 alice testfilter messagesView 1 <bob> test
5 alice testfilter messagesView 1 <charly> test
6 bob testfilter messagesView 1 <alice> test
7 bob testfilter messagesView 1 <charly> test
8 charly testfilter messagesView 1 <alice> test
9 charly testfilter messagesView 1 <bob> test
10 alice msg test
11 bob msg test
12 charly msg test
```

In this test, we have three clients, alice, bob, and charly. All do the same. They initialize the chat, set a filter expecting a message from the other two, and send a message themselves. The filter specifies the expected number of events.

A successful test shows a summary like the following:

Listing 6.5: Test summary example

```
1 Summary of "default" (8e389cce-7301-11e9-b4f3-acde48001122)
2
3 command "alice init" OK
4 command "alice testfilter messagesView 1 <bob> test" OK
5 event "alice testfilter messagesView 1 <bob> test" OK
6 command "alice testfilter messagesView 1 <charly> test" OK
7 event "alice testfilter messagesView 1 <charly> test" OK
8 command "alice msg test" OK
9 command "bob init" OK
10 command "bob testfilter messagesView 1 <alice> test" OK
11 event "bob testfilter messagesView 1 <alice> test" OK
12 command "bob testfilter messagesView 1 <charly> test" OK
13 event "bob testfilter messagesView 1 <charly> test" OK
14 command "bob msg test" OK
15 command "charly init" OK
16 command "charly testfilter messagesView 1 <alice> test" OK
17 event "charly testfilter messagesView 1 <alice> test" OK
18 command "charly testfilter messagesView 1 <bob> test" OK
19 event "charly testfilter messagesView 1 <bob> test" OK
20 command "charly msg test" OK
```

You see the commands with their return values and the events showing the match with their filters.

Currently, it is more a proof of concept. For instance, to test the behavior of the system if a client exits hard, we have a problem. The client can not remove the executed command line on the test server, and all following lines of test commands are blocked permanently. We could introduce a new test function which removes the command line before it executes the command. We will see more examples for enhancement [7.5].

## 6.4 Status

We have achieved a robust base system for a real decentralized and reactive system. According to the needs and requirements of the application, we can enhance the system appropriately. Even if we have only a prototype system, we are sure how to make isomorphic clients a production-ready system for a large variety of applications:

- integrate the new application in the system
- complete the multi-client testing in a localhost environment
- open up the network environment by developing against the multi-client tests
- change the UI if wanted

As well we know how to develop the approach further to a development framework:

- integrate the application communication in the 'memberlist' layer
- generalize the application's communication using an interface with Protocol Buffer
- segregate the terminal UI and introduce an appropriate interface for a presentation layer

## 7 Summary

During the work on the prototypes, including the final result, we kept the complexity low and stayed focused on the design goals [2.5]. The experience we made indicates that the promise of Creative Learning is real. The development system shows an open and transparent structure and proved its usability in our test case, i.e., developing in parallel the development system and the chat application. Finally, we can determine the chat application, as an example implementation, is reactive, and the design is appropriate to produce reactive applications in general.

Here the results regarding the design goals:

- it is stateless except regarding the bootstrap service and the test server
- a clear separation between infrastructure and application is manifested, even not fully implemented yet
- the only orchestration does the test server
- the infrastructural aspects can be mitigated to APIs used by the applications
- it shares only the member list and the connection information of the services
- only a minimum of services are left, i.e., it overcomes the client-server paradigm
- a restricted set of only one production and one testing microservice
- deployed the productive bootstrap microservice on Cloud Functions
- using 'memberlist' as a production-ready library for peer-to-peer

### 7.1 Creative Learning

During the development of the prototypes, the concept of Creative Learning made it possible to be always motivated and continually learning without losing the understanding of the current subject, or the creativity. Disappointments about how it should be implemented or work, then, turned out not to block us finding other solutions.

For instance, when we found no solution how to get the address of the sender of a message in the gRPC framework, we found a reasonable and practical solution [5.2.2]. When we had chosen a too complicated approach regarding the multi-client testing [5.3.3], we stepped back and found a more straightforward solution introducing a test server [6.3].

So, evolving new software by creating prototypes in spiraling cycles made it even possible to realize a complex task with multiple aspects successfully, and with pleasure.

## 7.2 Chat As An Example Application

The final client we developed is genuinely isomorphic. It acts as a server on the Memberlist Layer and the Chat Layer, as well, as a client on both plus additionally on the Bootstrap Layer. We have to integrate the communication of the Chat Layer in the Memberlist Layer by introducing a new message type [5.4.2.] Then, we can lose connectivity between the clients only on the Memberlist Layer, and this can be handled by the SWIM++ protocol appropriately. Additionally, to its professional use [3.3.1], the 'memberlist' implementation showed its stability during all the development and testing, we had done. Finally, the bootstrap service having a small functional footprint and being deployed on Cloud Functions makes the get together of the clients high available.

## 7.3 Development System Prototype

The development system prototype we created provides a very flexible and handy way of work as a developer of decentralized systems. Especially if we want to explore new ideas or unfamiliar libraries, we can make our first steps in an unknown area with multiple running clients acting interactively and freely. We implement based on the findings, then, and evolve to reach the goal of the prototype. It is a perfect fit for the approach of Creative Learning to create prototypes.

Finally, our application reached a certain maturity, and we start to test the interaction of multiple clients. Here, we have not yet implemented all features for a high test coverage [7.5]. For debugging and, later in production, for monitoring, the basic features are implemented, like flexible logging, configuration handling, and, last but not least, scripting to automate tasks.

## 7.4 Prove It As Reactive System

When we started, we emphasized the goal to implement a system to be reactive as defined by the Reactive Manifesto. Let us now see if we succeeded.

### 7.4.1 Message Driven

Is our system message-driven?

The layers of our system are communicating via messages:

- the bootstrap service via HTTP in JSON format and plain text
- the 'memberlist' library via TCP and UDP using Protocol Buffer



- the chat application via TCP in plain text

All ensures loose coupling, isolation and location transparency. The system sends errors as messages.

### 7.4.2 Elastic

Does the system scale accordingly?

From a hardware resources perspective, the scaling is system immanent. The requirements scale with the number of peers as the resources do. But, for communication traffic, it is not self-evident. The bootstrap service receives only one request per starting client to provide a short list of bootstrap peers. But, in case of a left bootstrap peer, the service gets requests from all online peers to refill the list. The serverless runtime of the public cloud provider does ensure the needed scaling in our implementation. The SWIM++ implementation provides scalability by design. The application itself does not necessarily have proper scaling, as long as a member sends to all others directly. The application can use the Memberlist Layer to communicate and enhance its scalability appropriately.

### 7.4.3 Resilient

Is the system resilient?

Only the bootstrap service has a centralized role and is a single point of failure. Aside from the resilience of the serverless runtime of the public cloud provider, the clients could additionally remember the list of former bootstrap peers to get more reliability.

### 7.4.4 Responsive

Is the system really responsive?

We rely on the properties described above to ensure responsiveness. If the system uses asynchronous messages, it can not be blocked, and react still. If it can scale and be elastic, it is available accordingly. And, finally, if it can handle failure and is resilient, it is responsive as a reactive system.

## 7.5 Outlook

We have seen how to develop decentralized systems and how a development system to do this can seem. Still, there is a way to go before we can catch up with the established development frameworks following the client-server paradigm. We do an outlook concerning what are the next steps to go according to Creative Learning style.

There are some todos to implement first:

1. enhance the multi-client testing and scripting, respectively, e.g., new test functions and variables
2. write test scripts for sufficient test coverage and to be used during further refactoring and implementing

Now, we can refactor:

1. move the communication of the Application Layer entirely to the Memberlist Layer
2. separate the Application Layer plus UI from Bootstrap and Memberlist Layer
3. make Bootstrap and Memberlist Layer a package with API and interfaces
4. separate the Application Layer from the UI

After some polishing like documentation and code quality, we can start new explorations:

- how to move to the internet and other network scenarios
- other UIs like mobile first or in the browser using e.g., Kotlin [Kot], Swift [Swi] or WebAssembly [Web]
- persistence layer using decentralized or cloud storage
- security layer using encryption or blockchain, respectively
- applications

# Books & Articles

- [Abe66] Abelson and Sussman. *"Structure and Interpretation of Computer Programs"*. Programs must be written for people to read, and only incidentally for machines to execute. 1966. ISBN: 3-540-63898-9.
- [Bri81] Brian W. Kernighan. "Software Tools in Pascal." In: (1981). URL: <http://www.catb.org/~esr/writings/taouu/html/ch02.html>.
- [Ker84] Kernighan, Brian Wilson; Pike, Robert C. *"The UNIX Programming Environment"*. Englewood Cliffs: Prentice-Hall, 1984. ISBN: 0-13-937699-2.
- [Mic86] Michael Stonebraker. "The Case for Shared Nothing." In: (1986). URL: <http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf>.
- [Mey88] Meyer, Bertrand. *"Object-Oriented Software Construction"*. Prentice Hall. ISBN 0-13-629049-3, 1988.
- [Fer92] Ferraiolo, D.F. and Kuhn, D.R. *"Role-Based Access Control"*. 1992. URL: <http://csrc.nist.gov/groups/SNS/rbac/documents/ferraiolo-kuhn-92.pdf>.
- [Gur92] Gurdip Singh. "Leader Election in Complete Networks." In: (1992). "Leader election is a fundamental problem in distributed computing and has a number of applications." URL: <http://www.cis.ksu.edu/~singh/papers/election.ps.gz>.
- [Set02] Seth Gilbert and Nancy Lynch. *"Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services"*, pg. 51. "... consistency, availability, and partition tolerance. It is impossible to achieve all three." ACM SIGACT News, Volume 33 Issue 2, 2002.
- [Eri03] Eric Steven Raymond. *"The Art of Unix Programming" page 113*. 2003. URL: <https://nakamotoinstitute.org/static/docs/taoup.pdf>.
- [Eri04] Eric Steven Raymond. "The Art of Unix Usability" (Chapter 2: "A Brief History of User Interfaces"). In: (2004). URL: <http://www.catb.org/~esr/writings/taouu/html/ch02.html>.
- [Hoa04] Hoare, C. A. R. *"Communicating Sequential Processes"*. Prentice Hall International. ISBN 978-0-13-153271-7, (2004) [1985].
- [Mit07] Mitchel Resnick. "All I Really Need to Know (About Creative Thinking) I Learned (By Studying How Children Learn) in Kindergarten." In: *MIT Media Lab* (2007). URL: <https://web.media.mit.edu/~mres/papers/kindergarten-learning-approach.pdf>.

- [Eff09] Effie L-C. Law etc. *"Understanding, Scoping and Defining User eXperience: A Survey Approach"*. 2009.
- [H. 12] H. M. N. Dilum Bandara, Anura P. Jayasumana. *"Collaborative Applications over Peer-to-Peer Systems - Challenges and Solutions"*. 2012. URL: <https://arxiv.org/pdf/1207.0790.pdf>.
- [Mar15a] Mark Richards. "Software Architecture Patterns", Chapter 5: "Space-Based Architecture." In: (2015). URL: <http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf>.
- [Mar15b] Martin L. Abbott and Michael T. Fisher. *"The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise" (2nd Edition)*. Pearson Education Inc. 0-134-03280-2, 2015.
- [Lee18] Lee Calcote. *"The Enterprise Path to Service Mesh Architectures"*. O'Reilly Media, Inc., 2018.
- [Mic18] Michael Hausenblas. *"Container Networking"*. O'Reilly Media, Inc., 2018.
- [Arn] Arnon Rotem-Gal-Oz. *"Fallacies of Distributed Computing Explained"*. URL: <http://www.rgoarchitects.com/Files/fallacies.pdf>.
- [Sta] Stavros Nikolaou, Robbert van Renesse, Nicolas Schiper. "Cooperative client caching strategies for social and web applications." In: (). URL: <https://www.cs.cornell.edu/~snikolaou/files/LADIS13.pdf>.

# Websites

- [Roy00] Roy Thomas Fielding. *"Architectural Styles and the Design of Network-based Software Architectures"*, XVI. 2000. URL: [http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation\\_2up.pdf](http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf).
- [Rus09] Russ Cox. *"The Generic Dilemma"*. 2009. URL: <https://research.swtch.com/generic>.
- [And10] Andrew Gerrand. *"Share Memory By Communicating" from the official Go blog*. . . . . 2010. URL: <https://blog.golang.org/share-memory-by-communicating>.
- [Fow14] Martin Fowler. *Blog "BoundedContext"*. . . . . 2014. URL: <https://www.martinfowler.com/bliki/BoundedContext.html>.
- [Jam14] James Lewis and Martin Fowler. *"Microservices - a definition of this new architectural term"*. 2014. URL: <https://www.martinfowler.com/articles/microservices.html>.
- [Jon14] Jonas Boner, Dave Farley, Roland Kuhn, and Martin Thompson. *"The Reactive Manifesto"*. 2014. URL: <https://www.reactivemanifesto.org>.
- [Ste17] Stefan Hans. *"Core Functional Streaming in Go" Talk*. 2017. URL: <https://go-talks.appspot.com/github.com/stefanhans/go-present/slides/FunctionalStreaming/functional-streaming.slide>.
- [Gol] *Introduction of "The Go Programming Language Specification"*. 2018. URL: <https://golang.org/ref/spec>.
- [Abh] Abhinandan Das, Indranil Gupta, Ashish Motivala. *"SWIM++" - Scalable Weakly-consistent Infection-style Process Group Membership Protocol*. URL: [https://www.cs.cornell.edu/Info/Projects/Spinglass/public\\_pdfs/SWIM.pdf](https://www.cs.cornell.edu/Info/Projects/Spinglass/public_pdfs/SWIM.pdf).
- [Eks] *Amazon Elastic Container Service for Kubernetes*. URL: <https://aws.amazon.com/eks/>.
- [Dom] *Articles on martinfowler.com*. URL: <https://martinfowler.com/tags/domain%20driven%20design.html>.
- [Aks] *Azure Kubernetes Service*. URL: <https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes>.
- [Goca] *Channel types of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Channel\\_types](https://golang.org/ref/spec#Channel_types).

- [Fra] Francesc Campoy Flores. *"Functional Go?" Talk at dotGo 2015*. URL: <https://www.youtube.com/watch?v=ouyHp2nJl0I>.
- [Gof] *Function types of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Function\\_types](https://golang.org/ref/spec#Function_types).
- [Gosa] *Go Standard Libraries*. URL: <https://golang.org/pkg/#stdlib>.
- [Gor] *"go" statements of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Go\\_statements](https://golang.org/ref/spec#Go_statements).
- [Fir] *"Google Cloud Firestore" - Google's flexible, scalable NoSQL cloud database*. URL: <https://firebase.google.com/docs/firestore/>.
- [Gcf] *Google Cloud Functions*. URL: <https://cloud.google.com/functions/docs/>.
- [Clo] *"Google Cloud Functions" - Google's serverless runtime*. URL: <https://cloud.google.com/functions/docs/>.
- [Gcp] *Google Cloud Platform Free Tier*. URL: <https://cloud.google.com/free/docs/gcp-free-tier>.
- [Gke] *Google Kubernetes Engine*. URL: <https://cloud.google.com/kubernetes-engine/docs/>.
- [Grpa] *gRPC, A high performance, open-source universal RPC framework*. URL: <http://www.grpc.io/>.
- [Grpm] *gRPC-Go, The Go implementation of gRPC*. URL: <https://github.com/grpc/grpc-go/>.
- [Goi] *Interface types of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Interface\\_types](https://golang.org/ref/spec#Interface_types).
- [Kis] *"KISS Principle" - Keep It Simple Stupid (KISS)*. URL: [http://principles-wiki.net/principles:keep\\_it\\_simple\\_stupid](http://principles-wiki.net/principles:keep_it_simple_stupid).
- [K8sb] *Kubernetes' Concept of Volumes*. URL: <https://kubernetes.io/docs/concepts/storage/>.
- [Liba] *"libp2p" - A modular network stack*. URL: <https://libp2p.io>.
- [Goma] *Map types of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Map\\_types](https://golang.org/ref/spec#Map_types).
- [Gomb] *Method expressions of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Method\\_expressions](https://golang.org/ref/spec#Method_expressions).
- [Vkl] *"Microservices definition" from blog on "Klang Heavy Industries"*. URL: <https://viktorklang.com/blog/Microservices-definition.html>.
- [Lcl] *MindShift: 10 Tips for Creating a Fertile Environment for Kids' Creativity and Growth*. URL: <https://www.kqed.org/mindshift/49362/10-tips-for-creating-a-fertile-environment-for-kids-creativity-and-growth>.

- [Mit] *MIT Lifelong kindergarten*. URL: <https://www.media.mit.edu/groups/lifelong-kindergarten/overview/>.
- [ML] *ML Learning*. URL: <https://www.media.mit.edu/groups/ml-learning/overview/>.
- [Gosb] *Package "context" of Go's standard library*. URL: <https://golang.org/pkg/context/>.
- [Gosc] *Package "json" of Go's standard library*. URL: <https://golang.org/pkg/encoding/json/>.
- [Gosd] *Package "net" of Go's standard library*. URL: <https://golang.org/pkg/net>.
- [Gose] *Package "testing" of Go's standard library*. URL: <https://golang.org/pkg/testing/>.
- [Pet] Petar Maymounkov and David Mazieres. *"Kademlia" - A Peer-to-peer Information System Based on the XOR Metric*. URL: <http://www.scs.stanford.edu/~dm/home/papers/kpos.pdf>.
- [Proa] *Projects of Protocol Labs*. URL: <https://protocol.ai/projects/>.
- [Cnc] *Projects of the Cloud Native Computing Foundation*. URL: <https://www.cncf.io/projects/>.
- [Proh] *Protocol Buffers, A language-neutral, platform-neutral extensible mechanism for serializing structured data*. URL: <https://developers.google.com/protocol-buffers/>.
- [Proi] *Protocol Buffers, The binary wire format for protocol buffer messages*. URL: <https://developers.google.com/protocol-buffers/docs/encoding>.
- [Rfca] *RFC 2663 "IP Network Address Translator (NAT) Terminology and Considerations"*. URL: <https://tools.ietf.org/html/rfc2663>.
- [Rfcb] *"RFC 791" - INTERNET PROTOCOL*. URL: <https://tools.ietf.org/html/rfc791>.
- [Rfcc] *RFC 8446 "The Transport Layer Security (TLS) Protocol Version 1.3"*. URL: <https://tools.ietf.org/html/rfc8446>.
- [Rob] Rob Pike. *"Simplicity is Complicated" Talk on dotGo 2015*. URL: <https://talks.golang.org/2015/simplicity-is-complicated.slide#29>.
- [The] *Scale Cube: Simplified Scale Model for Microservices*. URL: <https://dzone.com/articles/scale-cube-simplified-scale-model>.
- [Gosf] *"select" statements of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Select\\_statements](https://golang.org/ref/spec#Select_statements).
- [Gosg] *Slice types of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Slice\\_types](https://golang.org/ref/spec#Slice_types).

- [Stea] Stefan Hans. *Talk: "go-libp2p" - Introduction to libp2p for Golang Developer*. URL: <https://go-talks.appspot.com/github.com/stefanhans/go-present/slides/libp2p/go-libp2p.slide>.
- [Steb] Sten Pittet. *"Continuous integration vs. continuous delivery vs. continuous deployment"*. URL: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>.
- [Stec] Steve Francia. *"Inheritance Is Best Left Out" from "Is Go An Object Oriented Language?"*. URL: <https://spf13.com/post/is-go-object-oriented/>.
- [Gosh] *Struct types of "The Go Programming Language Specification"*. URL: [https://golang.org/ref/spec#Struct\\_types](https://golang.org/ref/spec#Struct_types).
- [Gob] *The Go Blog "Arrays, slices (and strings): The mechanics of 'append' "*. URL: <https://blog.golang.org/slices>.
- [Kot] *The Kotlin Programming Language*. URL: <https://kotlinlang.org/docs/reference/>.
- [Swi] *The Swift Programming Language*. URL: <https://swift.org/documentation>.
- [Web] *WebAssembly - A Binary Instruction Format*. URL: <https://webassembly.org/>.
- [Dht] *Wikipedia: Distributed hash table*. URL: [https://en.wikipedia.org/wiki/Distributed\\_hash\\_table](https://en.wikipedia.org/wiki/Distributed_hash_table).
- [Kad] *Wikipedia: Kademlia*. URL: <https://en.wikipedia.org/wiki/Kademlia>.
- [Yag] *"YAGNI Principle" - You Ain't Gonna Need It (YAGNI)*. URL: <https://www.martinfowler.com/bliki/Yagni.html>.



# Repositories

- [Doc] *Docker CE*. URL: <http://github.com/docker/docker-ce>.
- [Stea] *Forked "A Kademlia DHT implementation on go-libp2p"*. URL: <https://github.com/stefanhans/go-libp2p-kad-dht>.
- [Go a] *"go-libp2p-kad-dht" - A Kademlia DHT implementation on go-libp2p*. URL: <https://github.com/libp2p/go-libp2p-kad-dht>.
- [Go b] *"go-libp2p-nat" - NAT port mapping library for go-libp2p*. URL: <https://github.com/libp2p/go-libp2p-nat>.
- [Gocb] *"GOCUI - Go Console User Interface" - GOCUI - Go Console User Interface*. URL: <https://github.com/jroimartin/gocui>.
- [K8sa] *Kubernetes*. URL: <https://github.com/kubernetes/kubernetes>.
- [Libb] *"libp2p" - Modular peer-to-peer networking stack (used by IPFS and others)*. URL: <https://github.com/libp2p/>.
- [Steb] *libp2p-chat*. URL: <https://github.com/stefanhans/go-libp2p-kad-dht/tree/master/examples/cmdtool>.
- [Lina] *"liner" - Pure Go line editor with history, inspired by linenoise*. URL: <https://github.com/peterh/liner>.
- [Mem] *"memberlist" - Golang package for gossip based membership and failure detection*. URL: <https://github.com/hashicorp/memberlist>.
- [Eth] *Official Golang implementation of the Ethereum protocol*. URL: <https://github.com/ethereum/go-ethereum>.
- [Cha] *p2p chat app with libp2p [with peer discovery]*. URL: <https://github.com/libp2p/go-libp2p-examples/tree/master/chat-with-rendezvous>.
- [Ipf] *Reference Implementation of IPFS*. URL: <https://github.com/ipfs/go-ipfs>.
- [Has] *Repositories of HashiCorp*. URL: <https://github.com/hashicorp>.
- [Ter] *Termbox is a library that provides a minimalistic API which allows the programmer to write text-based user interfaces*. URL: <https://github.com/nsf/termbox-go>.

# Code

[Cf a]	<i>"cf-chat" - Chat Using Cloud Functions As List Of Members Service.</i> . . . . . prototypes/cf-chat.
[Cf b]	<i>cf-chat/cf: List.</i> prototypes/cf-chat/cf/list.go.
[Cf c]	<i>cf-chat/cf: Reset.</i> prototypes/cf-chat/cf/reset.go.
[Cf d]	<i>cf-chat/cf: service data structure in JSON.</i> prototypes/cf-chat/cf/types.go.
[Cf e]	<i>cf-chat/cf: Subscribe.</i> prototypes/cf-chat/cf/subscribe.go.
[Cf f]	<i>cf-chat/cf: Unsubscribe.</i> prototypes/cf-chat/cf/unsubscribe.go.
[Cf g]	<i>cf-chat/cf: Unsubscribe.</i> prototypes/cf-chat/chat/sender.go (Line: 12).
[Cf h]	<i>cf-chat/chat:</i> prototypes/cf-chat/chat/chat.go (Line: 81).
[Cf i]	<i>cf-chat/chat:</i> prototypes/cf-chat/chat/sender.go (Line: 42).
[Cf j]	<i>cf-chat/chat: CreateMemberlist.</i> . . . . . prototypes/cf-chat/chat/memberlist.go (Line: 14).
[Cf k]	<i>cf-chat/chat: handlePublishReply.</i> prototypes/cf-chat/chat/receiver.go (Line: 138).
[Cf l]	<i>cf-chat/chat: handleTestCmdRequest.</i> . . . . . prototypes/cf-chat/chat/test_handlers.go (Line: 60).
[Cf m]	<i>cf-chat/chat: handleTestPublishRequest.</i> . . . . . prototypes/cf-chat/chat/test_handlers.go (Line: 12).
[Cf n]	<i>cf-chat/chat: Initialize.</i> prototypes/cf-chat/chat/chat.go (Line: 46).
[Cf o]	<i>cf-chat/chat: main.</i> prototypes/cf-chat/chat/main.go (Line: 20).
[Cf p]	<i>cf-chat/chat: startChatListener.</i> prototypes/cf-chat/chat/receiver.go (Line: 13).
[Cf q]	<i>cf-chat/chat: test functions.</i> prototypes/cf-chat/chat/main_test.go.
[Cf r]	<i>cf-chat/chat/chat-group: MessageType.</i> . . . . . prototypes/cf-chat/chat/chat-group/chat-group.proto (Line: 16).
[Cf s]	<i>cf-chat/memberlist: 'IpAddress' struct for service.</i> . . . . . prototypes/cf-chat/memberlist/api.go (Line: 52).
[Cf t]	<i>cf-chat/memberlist: List.</i> prototypes/cf-chat/memberlist/api.go (Line: 120).
[Cf u]	<i>cf-chat/memberlist: 'Memberlist' struct.</i> . . . . . prototypes/cf-chat/memberlist/api.go (Line: 16).

[Cf v]	<i>cf-chat/memberlist: Reset.</i> prototypes/cf-chat/memberlist/api.go (Line: 144).
[Cf w]	<i>cf-chat/memberlist: Subscribe.</i> prototypes/cf-chat/memberlist/api.go (Line: 70).
[Cf x]	<i>cf-chat/memberlist: Unsubscribe.</i> prototypes/cf-chat/memberlist/api.go (Line: 99).
[Fina]	<i>Final Prototype Bootstrap API.</i> bootstrap-data-api/api.go.
[Finb]	<i>Final Prototype Bootstrap API: serviceUrl.</i> bootstrap-data-api/api.go (Line: 52).
[Finc]	<i>Final Prototype Bootstrap Cloud Functions.</i> bootstrap-data-cloudfunctions.
[Find]	<i>Final Prototype Bootstrap Server: Run.</i> . . . . . bootstrap-data-server/bootstrap-server/run.go.
[Fine]	<i>Final Prototype Chat: broadcast.go.</i> cli-chat/chat/broadcast.go.
[Finf]	<i>Final Prototype Chat: cmdLogging.</i> cli-chat/chat/logging.go (Line: 64).
[Fing]	<i>Final Prototype Chat: DefaultLocalConfig.</i> cli-chat/chat/memberlist.go (Line: 30).
[Finh]	<i>Final Prototype Chat: executeScript.</i> cli-chat/chat/script.go (Line: 15).
[Fini]	<i>Final Prototype Chat: for !testend.</i> cli-chat/chat/run.go (Line: 83).
[Finj]	<i>Final Prototype Chat: help.go.</i> cli-chat/chat/help.go.
[Fink]	<i>Final Prototype Chat: joiningChat.</i> cli-chat/chat/broadcast.go (Line: 194).
[Finl]	<i>Final Prototype Chat: leavingChat.</i> cli-chat/chat/broadcast.go (Line: 228).
[Finm]	<i>Final Prototype Chat: loadMemberlistConfiguration.</i> . . . . . cli-chat/chat/memberlist.go (Line: 166).
[Finn]	<i>Final Prototype Chat: memberlist.go.</i> . . . . . cli-chat/chat/memberlist.go.
[Fino]	<i>Final Prototype Chat: NotifyLeave.</i> cli-chat/chat/event_delegate.go (Line: 23).
[Finp]	<i>Final Prototype Chat: NotifyMsg.</i> cli-chat/chat/delegate.go (Line: 42).
[Finq]	<i>Final Prototype Chat: saveMemberlistConfiguration.</i> . . . . . cli-chat/chat/memberlist.go (Line: 130).
[Finr]	<i>Final Prototype Chat: send.</i> cli-chat/chat/tui.go (Line: 82).
[Fins]	<i>Final Prototype Chat: var bootstrapApi.</i> cli-chat/chat/bootstrap.go (Line: 10).
[Fint]	<i>Final Prototype Chat: var bootstrapApi.</i> cli-chat/chat/commander.go (Line: 20).
[Finu]	<i>Final Prototype Chat: view "input".</i> cli-chat/chat/tui.go (Line: 48).
[Finv]	<i>Final Prototype Chat: view "messages".</i> cli-chat/chat/tui.go (Line: 38).

[Finw]	<i>Final Prototype Chat: wait for connections.</i> cli-chat/chat/chat.go (Line: 71).
[Finx]	<i>Final Prototype Test Server: InitRun.</i> test-server/cli-test-server/testrun.go (Line: 15).
[Finy]	<i>Final Prototype Test Server: PrepareTestSummary.</i> . . . . . test-server/cli-test-server/summary.go (Line: 43).
[Finz]	<i>Final Prototype Test Server: Run.</i> test-server/cli-test-server/run.go (Line: 61).
[Finaa]	<i>Final Prototype Test Server: saveTestJsonData.</i> . . . . . test-server/cli-test-server/summary.go (Line: 12).
[Finab]	<i>Final Prototype Test Server: var currentTestRun.</i> . . . . . test-server/cli-test-server/types.go (Line: 59).
[Grpb]	<i>"gRPC-chat" - Chat Using Protocol Buffers And gRPC.</i> prototypes/gRPC-chat.
[Grpc]	<i>gRPC-chat: chat-group.pb.go.</i> . . . . . prototypes/gRPC-chat/chat-group/chat-group.pb.go.
[Grpd]	<i>gRPC-chat: chat-group.proto.</i> . . . . . prototypes/gRPC-chat/chat-group/chat-group.proto.
[Grpe]	<i>gRPC-chat: implements DisplayerServer.</i> prototypes/gRPC-chat/displayer.go (Line 36).
[Grpf]	<i>gRPC-chat: implements PublisherServer.</i> prototypes/gRPC-chat/publisher.go (Line 49).
[Grpg]	<i>gRPC-chat: run TUI.</i> prototypes/gRPC-chat/tui.go (Line: 15).
[Grph]	<i>gRPC-chat: start chat.</i> prototypes/gRPC-chat/main.go (Line: 24).
[Grpi]	<i>gRPC-chat: start displayer.</i> prototypes/gRPC-chat/displayer.go (Line: 12).
[Grpj]	<i>gRPC-chat: start publisher.</i> prototypes/gRPC-chat/publisher.go (Line: 19).
[Grpk]	<i>gRPC-chat: subscribe.</i> prototypes/gRPC-chat/publisher_client.go (Line: 12).
[Grpl]	<i>gRPC-chat: wrapper to dial.</i> prototypes/gRPC-chat/publisher_client.go.
[Grpn]	<i>gRPC: .pb.go file example.</i> . . . . . thesis-code-snippets/3.4.1/info-gRPC/info/info.pb.go.
[Grpo]	<i>gRPC: .proto file example.</i> . . . . . thesis-code-snippets/3.4.1/info-gRPC/info/info.proto.
[Libc]	<i>"libp2p-chat" - Pre-Chat Using 'libp2p'.</i> prototypes/libp2p-chat.
[Linb]	<i>"liner-memberlist-chat" - Pre-Chat Using 'memberlist'.</i> . . . . . prototypes/liner-memberlist-chat.
[Linc]	<i>liner-memberlist-chat: Bootstrap API.</i> . . . . . prototypes/liner-memberlist-chat/bootstrap-data-api/api.go.

[Lind]	<i>liner-memberlist-chat: Bootstrap Cloud Functions.</i> . . . . . prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions.
[Line]	<i>liner-memberlist-chat: Bootstrap Server.</i> . . . . . prototypes/liner-memberlist-chat/bootstrap-data-server/bootstrap-server.
[Linf]	<i>liner-memberlist-chat: cmdLogging.</i> . . . . . prototypes/liner-memberlist-chat/commander.go (Line: 351).
[Ling]	<i>liner-memberlist-chat: commandsInit.</i> . . . . . prototypes/liner-memberlist-chat/commander.go (Line: 22).
[Linh]	<i>liner-memberlist-chat: executeCommand.</i> . . . . . prototypes/liner-memberlist-chat/commander.go (Line: 86).
[Linl]	<i>liner-memberlist-chat: isChatMemberReachable.</i> prototypes/liner-memberlist- chat/broadcast.go (Line: 158).
[Linj]	<i>liner-memberlist-chat: joiningChat.</i> . . . . . prototypes/liner-memberlist-chat/broadcast.go (Line: 180).
[Link]	<i>liner-memberlist-chat: Leave.</i> . . . . . prototypes/liner-memberlist-chat/bootstrap-data-api/api.go (Line: 120).
[Linl]	<i>liner-memberlist-chat: leaveChat.</i> . . . . . prototypes/liner-memberlist-chat/chat.go (Line: 87).
[Linm]	<i>liner-memberlist-chat: leavingChat.</i> . . . . . prototypes/liner-memberlist-chat/broadcast.go (Line: 211).
[Linn]	<i>liner-memberlist-chat: listenStream.</i> . . . . . prototypes/liner-memberlist-chat/chat.go (Line: 18).
[Lino]	<i>liner-memberlist-chat: loadMemberlistConfiguration.</i> . . . . . prototypes/liner-memberlist-chat/memberlist.go (Line: 157).
[Linp]	<i>liner-memberlist-chat: NotifyLeave.</i> . . . . . prototypes/liner-memberlist-chat/event_delegate.go (Line: 21).
[Linq]	<i>liner-memberlist-chat: NotifyMsg.</i> . . . . . prototypes/liner-memberlist-chat/delegate.go (Line: 40).
[Linr]	<i>liner-memberlist-chat: play.</i> . . . . . prototypes/liner-memberlist-chat/commander.go (Line: 389).
[Lins]	<i>liner-memberlist-chat: Run.</i> . . . . . prototypes/liner-memberlist-chat/bootstrap-data-server/bootstrap-server/ run.go (Line: 8).
[Lint]	<i>liner-memberlist-chat: saveMemberlistConfiguration.</i> . . . . . prototypes/liner-memberlist-chat/memberlist.go (Line: 125).
[Linu]	<i>liner-memberlist-chat: sendMessage.</i> . . . . . prototypes/liner-memberlist-chat/commander.go (Line: 266).

- [Linv] *liner-memberlist-chat/bootstrap-data-api: Refill.* prototypes/liner-memberlist-chat/bootstrap-data-api/api.go (Line: 153).
- [Linw] *liner-memberlist-chat/bootstrap-data-cloudfunctions: Join.* prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/join.go (Line: 15).
- [Linx] *liner-memberlist-chat/bootstrap-data-cloudfunctions: Leave.* prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/leave.go (Line: 15).
- [Liny] *liner-memberlist-chat/bootstrap-data-cloudfunctions: Ping.* prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/ping.go (Line: 12).
- [Linz] *liner-memberlist-chat/bootstrap-data-cloudfunctions: Refill.* prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/refill.go (Line: 15).
- [Linaa] *liner-memberlist-chat/bootstrap-data-cloudfunctions: Reset.* prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/reset.go (Line: 13).
- [Linab] *liner-memberlist-chat/bootstrap-data-cloudfunctions: Config.* prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/config.go (Line: 16).
- [Linac] *liner-memberlist-chat/bootstrap-data-cloudfunctions: List.* . . . . . prototypes/liner-memberlist-chat/bootstrap-data-cloudfunctions/list.go . . . (Line: 12).
- [Prob] *"protobuf-tcp-chat" - Chat Using Protocol Buffers Via TCP.* . . . . . prototypes/protobuf-tcp-chat.
- [Proc] *protobuf-tcp-chat: chat-group.proto.* . . . . . prototypes/protobuf-tcp-chat/chat-group/chat-group.proto.
- [Prod] *protobuf-tcp-chat: handleDisplayerRequest.* . . . . . prototypes/protobuf-tcp-chat/displayer.go (Line: 39).
- [Proe] *protobuf-tcp-chat: handlePublisherRequest.* . . . . . prototypes/protobuf-tcp-chat/publisher.go (Line: 71).
- [Prof] *"protobuf-udp-chat" - Chat Using Protocol Buffers Via UDP.* . . . . . prototypes/protobuf-udp-chat.
- [Prog] *protobuf-udp-chat: startDisplayer.* prototypes/protobuf-udp-chat/displayer.go (Line: 12).
- [Infa] *Protocol Buffers: .pb.go file example.* thesis-code-snippets/3.4.1/info.pb.go.
- [Infb] *Protocol Buffers: .proto file example.* thesis-code-snippets/3.4.1/info.proto.

# Acronyms

**aaS** as a Service. 6

**API** Application Programming Interface. 5, 15, 16, 19, 44, 68, 72, 77–80, 83, 84, 91, 94, 97–99, 103, 106

**CI/CD** Continuous Integration / Continuous Delivery. 18

**CLI** Command-Line Interface. 96

**CSP** Communicating Sequential Processes. 10

**CUI** Console User Interface. 62

**DDD** Domain-Driven Design. 15

**DDoS** Distributed Denial-of-Service. 16

**FaaS** Function as a service. 18

**gRPC** Google’s Remote Procedure Call framework. 4, 54, 57, 58, 72–74, 103

**HTTP** HyperText Transfer Protocol. 83, 86, 90, 94, 95, 100, 104

**IaaS** Infrastructure as a Service. 17

**IoT** Internet of Things. 6

**IP** Internet Protocol. 16, 40, 76

**IPFS** InterPlanetary File System. 9, 51

**JSON** JavaScript Object Notation. 3, 46, 78, 88, 90, 94, 95, 100, 104

**NAT** Network Address Translation. 99

**P2P** Peer-to-Peer. 18

**PaaS** Platform as a Service. 17

**RBAC** Role-Based Access Control. 16

**SN** Shared Nothing. 14

**SSL/TLS** Secure Socker Layer / Transport Layer Security. 16

**SWIM** Scalable Weakly-consistent Infection-style Process Group Membership Protocol.  
50, 51, 83, 95, 104, 105

**TCP** Transmission Control Protocol. 4, 40, 74, 90, 91, 104, 105

**TUI** Text-based User Interface. 4, 62, 64, 71

**UDP** User Datagram Protocol. 4, 40, 74, 76, 77, 90, 91, 104

**UI** User Interface. 5, 8, 13, 67, 69, 70, 72, 73, 83, 94, 96, 97, 99, 102, 106

**UX** User eXperience. 70, 97

**XML** eXtensible Markup Language. 54, 90