ЛЕКЦИЯ 14

RabbitMQ, Apache Kafka, NATS. gRPC.

Point to point

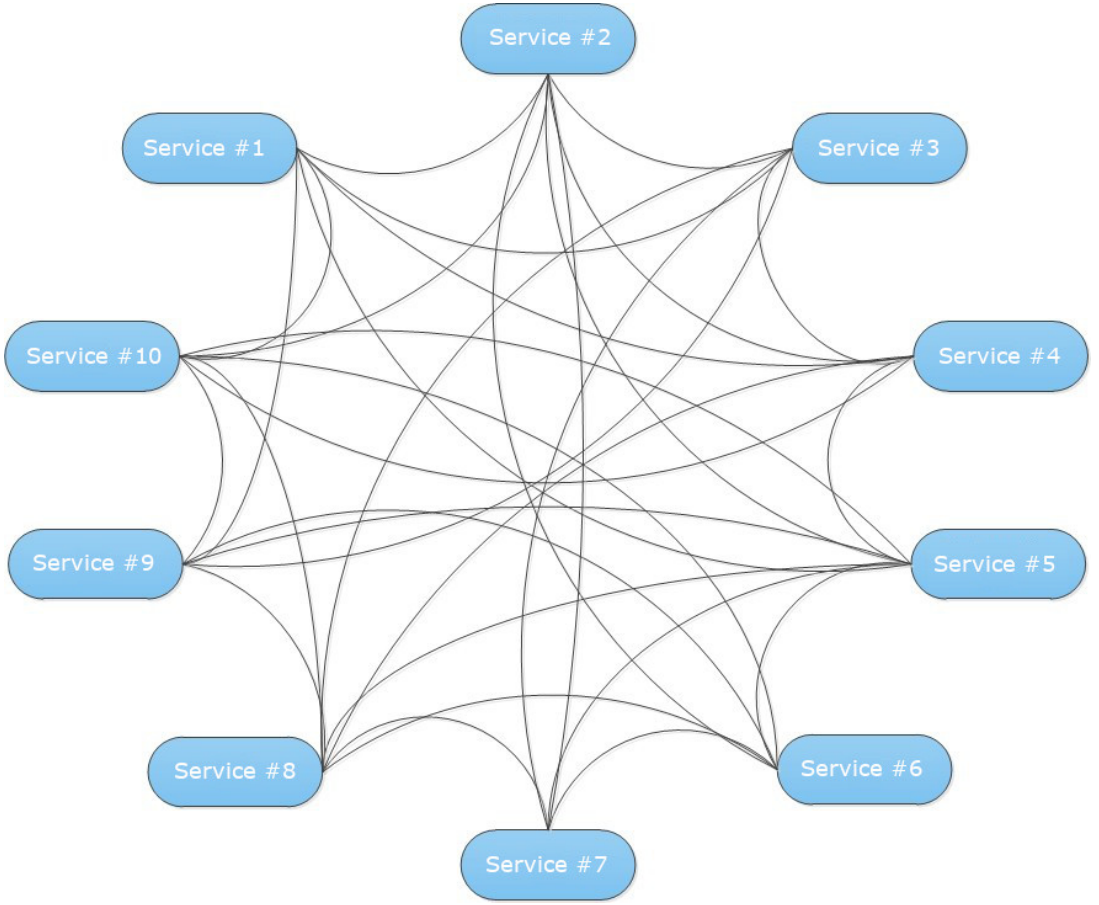
Point-To-Point:

Плюсы:

- Нет точки отказа в лице посредника
- Выше скорость коммуникации

Минусы:

- Сложность интеграции
- Сложность сопровождения и дальнейшего развития
- Проблемы с балансировкой

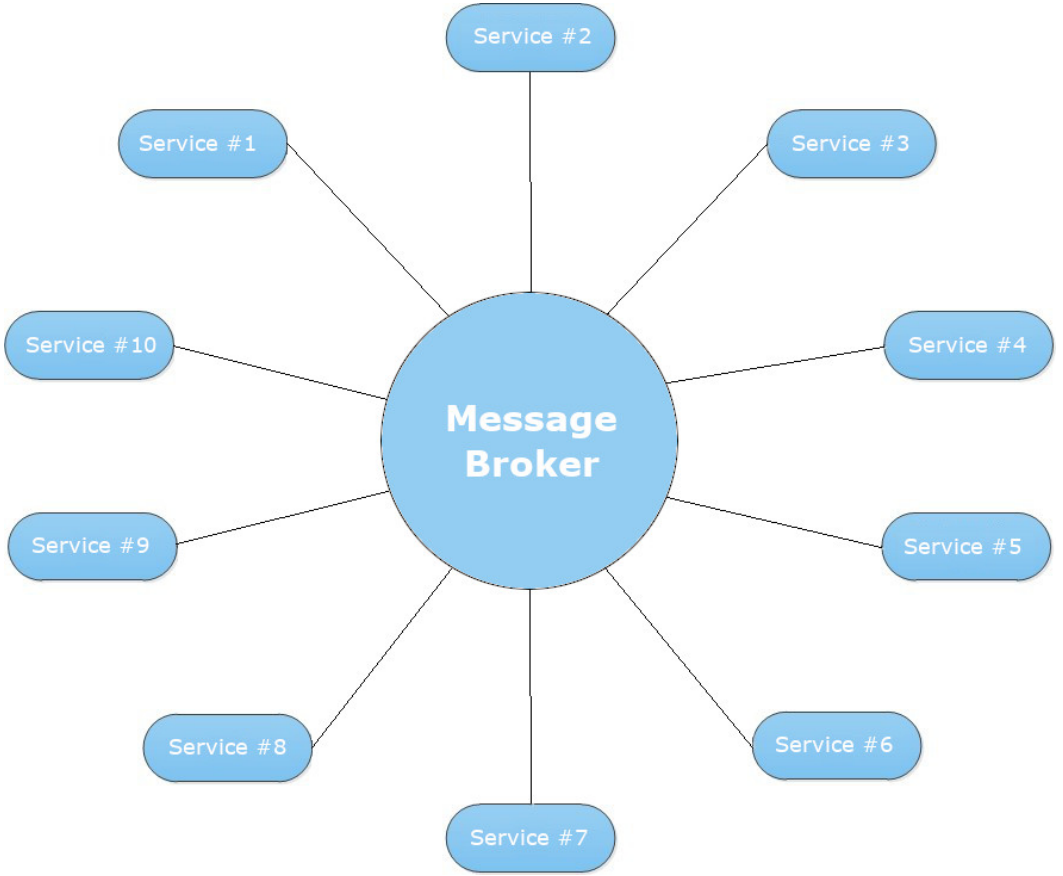


Message brokers

Message broker:

Плюсы:
- Простота интеграции
- Простота сопровождения и дальнейшего развития
- Нет проблем с балансировкой

Минусы:
- Точка отказа в лице брокера
- Ниже скорость коммуникации



Message brokers

Программный брокер сообщений на основе стандарта AMQP — тиражируемое связующее программное обеспечение, ориентированное на обработку сообщений. Поддерживается горизонтальное масштабирование для построения кластерных решений.

Создан на основе системы Open Telecom Platform, написан на языке Erlang, в качестве движка базы данных для хранения сообщений использует Mnesia (также написана на Erlang).

Изначально разрабатывался компанией SpringSource, после серии поглощений и разделений вошедшей в состав Pivotal; выпускается под Mozilla Public License.

AMQP

AMQP (Advanced Message Queuing Protocol):

Открытый протокол для передачи сообщений между компонентами системы.

Основная идея состоит в том, что отдельные подсистемы (или независимые приложения) могут обмениваться произвольным образом сообщениями через AMQP-брокер, который осуществляет маршрутизацию, возможно гарантирует доставку, распределение потоков данных, подписку на нужные типы сообщений.

Архитектуру протокола разработал John O’Hara из банка JP Morgan Chase & Co.

AMQP

AMQP основан на трёх понятиях:

Сообщение (message) — единица передаваемых данных, основная его часть (содержание) никак не интерпретируется сервером, к сообщению могут быть присоединены структурированные заголовки.

Точка обмена (exchange) — в неё отправляются сообщения. Точка обмена распределяет сообщения в одну или несколько очередей. При этом в точке обмена сообщения не хранятся. Точки обмена бывают трёх типов:

- fanout — сообщение передаётся во все прицепленные к ней очереди;
- direct — сообщение передаётся в очередь с именем, совпадающим с ключом маршрутизации (routing key) (ключ маршрутизации указывается при отправке сообщения);
- topic — нечто среднее между fanout и direct, сообщение передаётся в очереди, для которых совпадает маска на ключ маршрутизации, например, app.notification.sms.# — в очередь будут доставлены все сообщения, отправленные с ключами, начинающимися с app.notification.sms.

Очередь (queue) — здесь хранятся сообщения до тех пор, пока не будут забраны клиентом. Клиент всегда забирает сообщения из одной или нескольких очередей.

AMQP

AMQP параметры очереди:

Durable (true/false) – Если выставлен true, тогда при рестарте брокера, очередь сохранится (но не сообщения в ней!). Для сохранения сообщений при отправке нужно выбрать режим Persistence для параметра Delivery mode.

Exclusive (true/false) – Если выставлен true, то очередь будет поддерживать только одно соединение и будет удалена после его закрытия.

Auto-delete (true/false) – Если выставлен true, то очередь будет существовать только при наличии подписчиков (consumers), если не останется ни одного подписчика, она будет автоматически удалена.

Arguments – Опционально, можно передать параметры специфичные для брокера, например: message TTL, queue length и. т. д.

RabbitMQ

"streadway/amqp" producer:

```go
package main

import (
        "encoding/json"
        "github.com/masnun/gopher-and-rabbit"
        "github.com/streadway/amqp"
        "log"
        "math/rand"
        "time"
)

func handleError(err error, msg string) {
        if err != nil {
                log.Fatalf("%s: %s", msg, err)
        }

}
```

```go
func main() {

        conn, err := amqp.Dial(gopher_and_rabbit.Config.AMQPConnectionURL)
        handleError(err, "Can't connect to AMQP")
        defer conn.Close()

        amqpChannel, err := conn.Channel()
        handleError(err, "Can't create a amqpChannel")

        defer amqpChannel.Close()

        queue, err := amqpChannel.QueueDeclare("add", true, false, false, false, nil)
        handleError(err, "Could not declare `add` queue")

        rand.Seed(time.Now().UnixNano())

        addTask := gopher_and_rabbit.AddTask{
                Number1: rand.Intn(999),
                Number2: rand.Intn(999),
        }

        body, err := json.Marshal(addTask)
        if err != nil {
                handleError(err, "Error encoding JSON")
        }

        err = amqpChannel.Publish("", queue.Name, false, false, amqp.Publishing{
                DeliveryMode: amqp.Persistent,
                ContentType:  "text/plain",
                Body:         body,
        })

        if err != nil {
                log.Fatalf("Error publishing message: %s", err)
        }

        log.Printf("AddTask: %d+%d", addTask.Number1, addTask.Number2)
}
```

RabbitMQ

"streadway/amqp" consumer:

```go
package main

import (
        "encoding/json"
        "log"
        "os"

        gopher_and_rabbit "github.com/masnun/gopher-and-rabbit"
        "github.com/streadway/amqp"
)

func handleError(err error, msg string) {
        if err != nil {
                log.Fatalf("%s: %s", msg, err)
        }

}
```

```go
func main() {
        conn, err := amqp.Dial(gopher_and_rabbit.Config.AMQPConnectionURL)
        handleError(err, "Can't connect to AMQP")
        defer conn.Close()

        amqpChannel, err := conn.Channel()
        handleError(err, "Can't create a amqpChannel")

        defer amqpChannel.Close()

        queue, err := amqpChannel.QueueDeclare("add", true, false, false, false, nil)
        handleError(err, "Could not declare `add` queue")

        err = amqpChannel.Qos(1, 0, false)
        handleError(err, "Could not configure QoS")

        messageChannel, err := amqpChannel.Consume(
                queue.Name,
                "",
                false,
                false,
                false,
                false,
                nil,
        )
        handleError(err, "Could not register consumer")

        stopChan := make(chan bool)

        go func() {
                log.Printf("Consumer ready, PID: %d", os.Getpid())
                for d := range messageChannel {
                        log.Printf("Received a message: %s", d.Body)

                        addTask := &gopher_and_rabbit.AddTask{}

                        err := json.Unmarshal(d.Body, addTask)

                        if err != nil {
                                log.Printf("Error decoding JSON: %s", err)
                        }

                        log.Printf("Result of %d + %d is : %d", addTask.Number1, addTask.Number2,
                                addTask.Number1+addTask.Number2)

                        if err := d.Ack(false); err != nil {
                                log.Printf("Error acknowledging message : %s", err)
                        } else {
                                log.Printf("Acknowledged message")
                        }

                }
        }()

        // Stop for program termination
        <-stopChan

}
```

RabbitMQ

"djumanoff/amqp" server:

```
package main

import (
	"fmt"
	"github.com/djumanoff/amqp"
)

var cfg = amqp.Config{
	Host: "localhost",
	VirtualHost: "",
	User: "admin",
	Password: "admin",
	Port: 5672,
	LogLevel: 5,
}

var srvCfg = amqp.ServerConfig{
	//ResponseX: "response",
	//RequestX: "request",
}
```

```
func main() {

	sess := amqp.NewSession(cfg)

	if err := sess.Connect(); err != nil {
		fmt.Println(err)
		return
	}
	defer sess.Close()

	srv, err := sess.Server(srvCfg)
	if err != nil {
		fmt.Println(err)
		return
	}

	srv.Endpoint("request.get.test", func(d amqp.Message) *amqp.Message {
		fmt.Println("handler called")
		return &amqp.Message{
			Body: []byte("test"),
		}
	})

	if err := srv.Start(); err != nil {
		fmt.Println(err)
		return
	}
}
```

RabbitMQ

"djumanoff/amqp" client:

```
package main

import (
	"fmt"
	"github.com/djumanoff/amqp"
)

var cfg = amqp.Config{
	Host: "localhost",
	VirtualHost: "",
	User: "admin",
	Password: "admin",
	Port: 5672,
	LogLevel: 5,
}
```

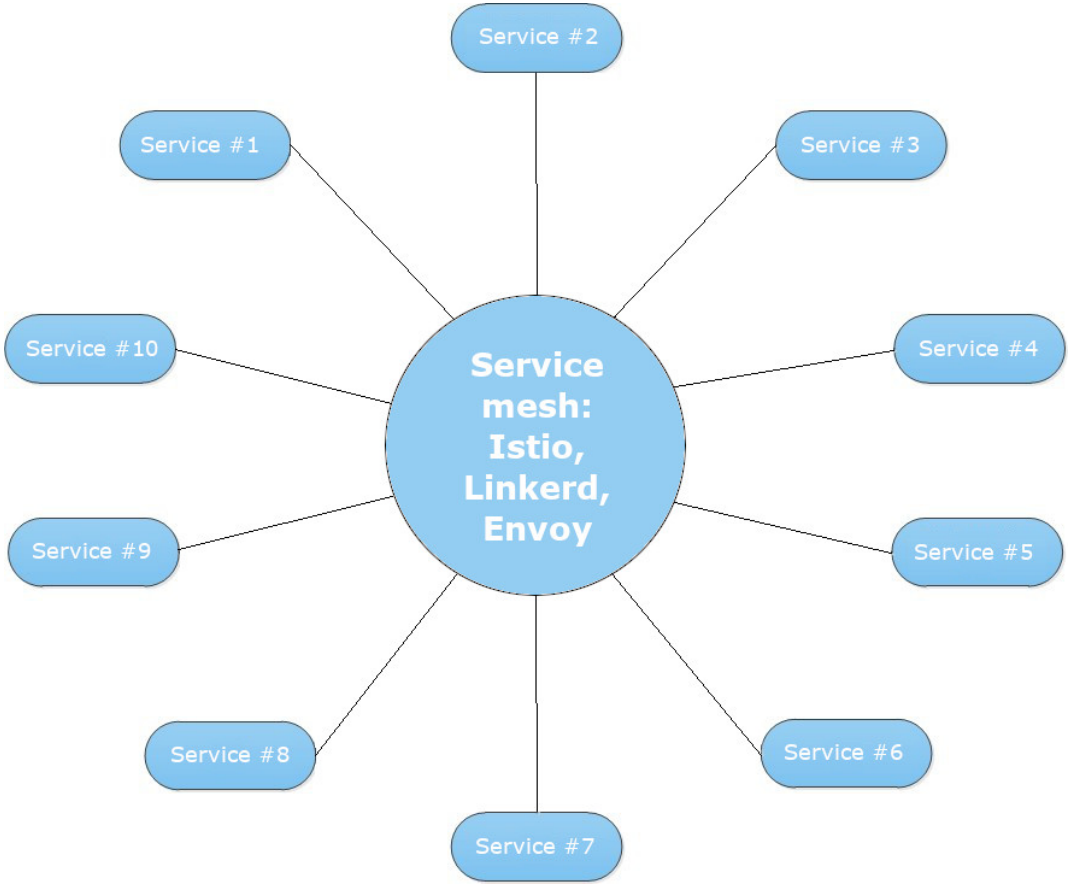
```
func main() {
	fmt.Println("Start")

	sess := amqp.NewSession(cfg)

	if err := sess.Connect(); err != nil {
		fmt.Println(err)
		return
	}
	defer sess.Close()

	var cltCfg = amqp.ClientConfig{
		//ResponseX: "response",
		//RequestX: "request",
	}

	clt, err := sess.Client(cltCfg)
	if err != nil {
		fmt.Println(err)
		return
	}

	reply, err := clt.Call("request.get.test", amqp.Message{
		Body: []byte("ping 1"),
	})
	if err != nil {
		fmt.Println(err)
		return
	}

	fmt.Println(reply)
	fmt.Println("End")
}
```

gRPC

gRPC:

Плюсы:
- Высокая скорость
- Нет точки отказа в лице брокера
- Нет проблем с балансировкой

Минусы:
- Сложность интеграции
- Повышенная связность



gRPC

Сначала создается .proto файл:

```
syntax = "proto3";

package reverse;

service Reverse {
    rpc Do(Request) returns (Response) {}
}

message Request {
    string message = 1;
}

message Response {
    string message = 1;
}
```

Затем выполняется генерация:

```
protoc –I . reverse.proto –go_out=plugins=grpc:.
```

Выполнение вышеприведенной команды создаст новый .go-файл, содержащий методы для создания клиента, сервера и сообщений, которыми они обмениваются.

gRPC. Client example

```go
package main

import (
	"context"
	"fmt"
	pb "github.com/matzhouse/go-grpc/proto"
	"google.golang.org/grpc"
	"google.golang.org/grpc/grpclog"
	"os"
)

func main() {
	opts := []grpc.DialOption{
		grpc.WithInsecure(),
	}
	args := os.Args
	conn, err := grpc.Dial("127.0.0.1:5300", opts...)

	if err != nil {
		grpclog.Fatalf("fail to dial: %v", err)
	}

	defer conn.Close()

	client := pb.NewReverseClient(conn)
	request := &pb.Request{
		Message: args[1],
	}
	response, err := client.Do(context.Background(), request)

	if err != nil {
		grpclog.Fatalf("fail to dial: %v", err)
	}

	fmt.Println(response.Message)
}
```

gRPC. Server example

```go
package main

import (
	"net"
	pb "github.com/matzhouse/go-grpc/proto"
	"golang.org/x/net/context"
	"google.golang.org/grpc"
	"google.golang.org/grpc/grpclog"
)

func main() {
	listener, err := net.Listen("tcp", ":5300")

	if err != nil {
		grpclog.Fatalf("failed to listen: %v", err)
	}

	opts := []grpc.ServerOption{}
	grpcServer := grpc.NewServer(opts...)

	pb.RegisterReverseServer(grpcServer, &server{})
	grpcServer.Serve(listener)
}

type server struct{}
```

```go
func (s *server) Do(c context.Context, request *pb.Request) (response
*pb.Response, err error) {
	n := 0
	// Create an array of runes to safely reverse a string.
	rune := make([]rune, len(request.Message))

	for _, r := range request.Message {
		rune[n] = r
		n++
	}

	// Reverse using runes.
	rune = rune[0:n]

	for i := 0; i < n/2; i++ {
		rune[i], rune[n-1-i] = rune[n-1-i], rune[i]
	}

	output := string(rune)
	response = &pb.Response{
		Message: output,
	}

	return response, nil
}
```

Хороший пример:
https://ewanvalentine.io/microservices-in-golang-part-1/