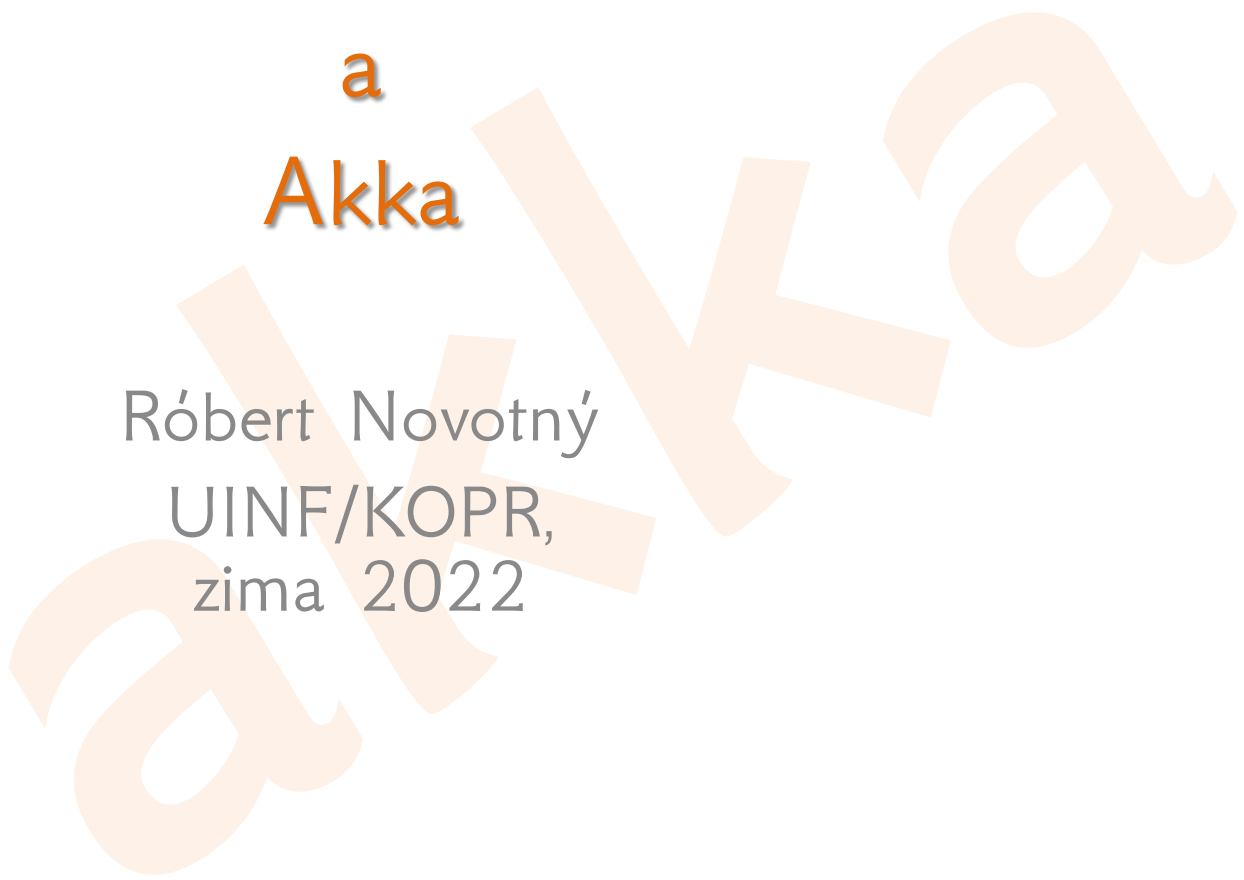
Aktory
a
Akka

Róbert Novotný
UINF/KOPR,
zima 2022



Tony dát a hordy klientov

- obrovské dáta
- hordy klientov
- veľký prietok
- nutnosť nízkej latencie

Škálovateľnosť má hranice

Aktorový model

- stiera sa rozdiel medzi distribuovaným a konkurentným
- žiadne zdieľané dáta!
- len sa posielajú správy bez čakania na odpoveď

Aktor = výpočtová entita

since 1973

- vie prijímať a odosielať správy
- po prijatí správy môže:
  - odoslať konečný počet správ iným aktorom
  - vytvoriť konečný počet nových aktorov
  - zmeniť svoj stav
  - určiť správanie pre správy prijaté v budúcnosti
- možnosti sa môžu diať v ľubovoľnom poradí
  - a dokonca paralelne

Asynchronicita

- nečaká sa na prijatie správy
- nečaká sa na odpoveď
- nezáleží na poradí prijatých správ

fire-and-forget

Správa

- ľubovoľný **nemenný** objekt posielaný medzi aktormi
- jediný spôsob výmeny informácií a zdieľania dát medzi aktormi
- medzi aktormi neexistujú žiadne zdieľané dáta!

Akka

aktorový model na platforme Java

Ako programovať v Akke?

použijeme: syntax Java + Akka Typed + OOP

- odporúčaná Scala
- odporúčaný funkcionálny model

Jestvujú výkonné Akka systémy

- SOAP/REST/HTTP endpointy
- paralelné a distribuované výpočty
- dávkové spracovanie a BigData
- microservices
- transakčné operácie

DEMO› POČÍTANIE FREKVENCIÍ SLOV

Frekvencia slov v dokumentoch

Napadlo mu: zlom dobro zlom.
V krajine Mu napadlo mnoho snehu.

napadlo:2, mu:2, zlom:2, dobro:1, v:1,
krajine:1, mnoho:1, snehu:1

Ako na deľbu práce?

- Jednotlivé vety rozhadzujeme na **uzly**.
- **Uzol** zráta frekvencie slov v jednej vete.
- Samostatný **uzol** berie čiastkové výsledky a kumuluje ich.

uzol = aktor

Príkazy a udalosti

- **command**: správa smerovaná na aktora
  - rozkaz!
  - zodpovedá „volaniu metódy“
- **event**:
  - udalosť po vykonaní rozkazu
  - zodpovedá „odpovedi na volanie metódy“

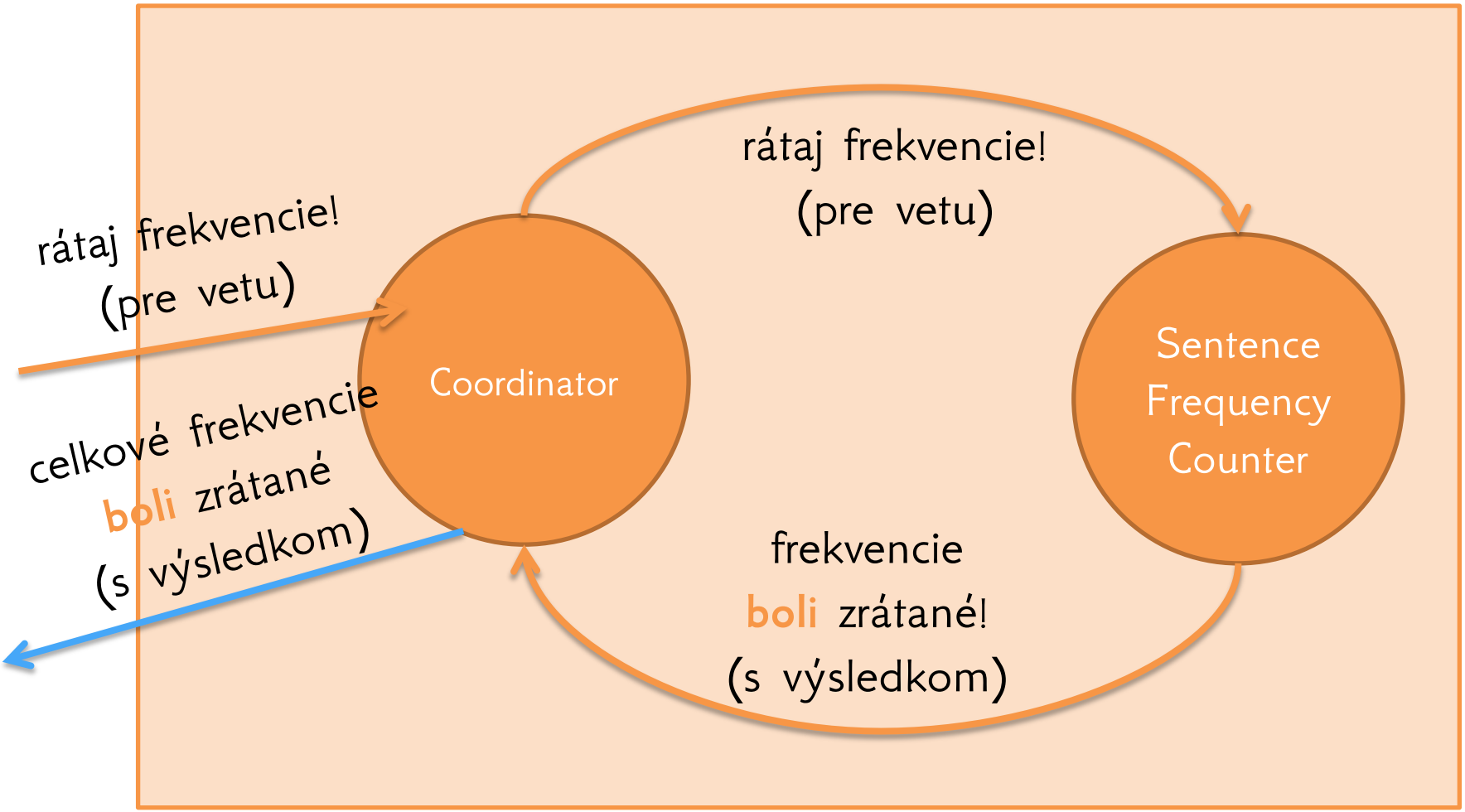
Analýza v Akke

- potrebujeme navrhnúť správy tečúce medzi aktormi
- správa je identifikovaná svojim dátovým typom

spočítaj frekvencie!

...

frekvencie boli spočítané


rátaj frekvencie!
(pre vetu)
Coordinator
celkové frekvencie
boli zrátané
(s výsledkom)
rátaj frekvencie!
(pre vetu)
Sentence
Frequency
Counter
frekvencie
boli zrátané!
(s výsledkom)

Akka Typed / Java / Object-Oriented

- dedí od triedy **AbstractBehavior**
- prekrýva metódu **createReceive**
- definuje **pravidlá** pre správy:

„ak príde správa typu X, vykonaj toto"

Akktor a Java: pravidlá pre správy

- dátový typ mapovaný na funkciu spracovávajúcu objekt správy

```
.onMessage(String.class, m -> System.out.println(m))
```

Akktor a Java: pravidlá pre správy

- pravidlá iniciované v metóde **createReceive()**
- budované cez **newReceiveBuilder()**

```
.onMessage(String.class, m -> System.out.println(m))
```

dátový typ prichádzajúcej správy

```
import akka.actor.typed.*;
public class ExampleActor extends AbstractActor<String> {
    public Receive<String> createReceive() {
        return newReceiveBuilder()
                .onMessage(String.class, s -> sayHello())
                .build();
    }
}
```

Akktory a ich vytváranie

- na vytváranie aktora sa nepoužíva **new ...()!**
- musíme deklarovať iniciálne správanie (behavior)
- konvencia:
  - 1 ks statická metóda na vytvorenie inštancie
  - 1 ks privátny oddedený konštruktor

```
import akka.actor.typed.*;
public class HelloActor extends AbstractBehavior<String> {

    private HelloActor(ActorContext<String> context) {
        super(context);
    }
```

```
import akka.actor.typed.*;
public class HelloActor extends AbstractBehavior<String> {

    private HelloActor(ActorContext<String> context) {
        super(context);
    }

    public static Behavior<String> create() {
        return Behaviors.setup(HelloActor::new);
    }

...
```

factory metóda

iniciálny behavior vytorí aktora konštruktorom

Hello World akktor

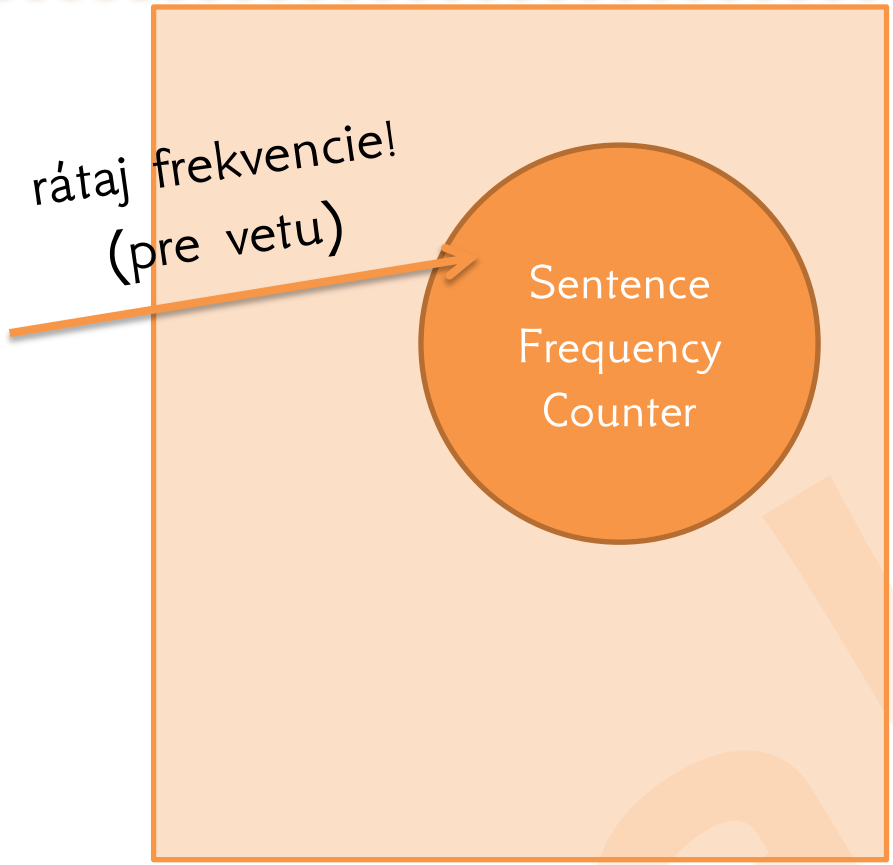
```
public static void main(String[] args) {
    ActorSystem<String> system
            = ActorSystem.create(HelloActor.create(), "system");

    for (int i = 0; i < 5; i++) {
        system.tell("Hi at " + new Date());
    }
}
```

factory metóda

povedz správu!

Verzia 1: jeden aktor



- 1 trieda správy
  - command
- konvencia: správy ako statické vnútorné triedy
- ešte neagregujeme celkové frekvencie
- výstup posielame na stdout

Thread-safety aktorových tried

- aktor spracuje správu pred spracovaním ďalšej správy
  - v súlade s "happens-before" zásadou z JVM
  - inštančné premenné nemusia byť thread-safe
- implementácia cez ideu **mailboxu**

Mailboxy a buffrovanie

- aktor má **mailbox**, v ktorom sa kopia správy
  - každý aktor má vlastný
  - je to **front**
  - radia sa podľa času odoslania
  - nedajú sa zdieľať
- možno využiť prioritný mailbox:
  - radenie podľa priority

Spracovanie správ

- aktor spracováva v danom čase **len jednu správu**
- ostatné sa kopia vo fronte
- dlhotrvajúce spracovanie správy znamená dlhšie čakanie!

message loop

Poradie doručenia

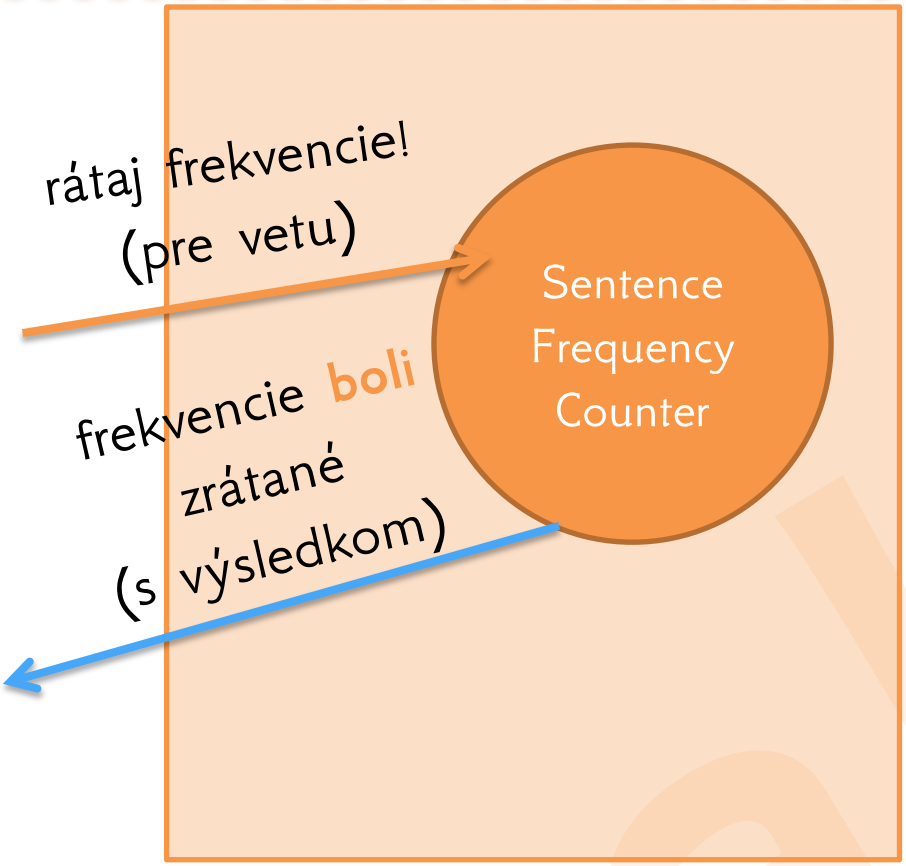
- správy z aktora A do aktora B **garantujú poradie** doručenia
  - ak nemáme špeciálny typ mailboxu
  - nespoliehajme sa na poradie doručenia správ z rozličných aktorov!

```
Receive<GetWordFreq> createReceive() {
    return newReceiveBuilder()
            .onMessage(GetWordFreq.class, this::getWordFreq)
            .build();
}

Behavior<GetWordFreq> getWordFreq(GetWordFreq command) {
    // 1. výpočet frekvencií
    // 2. výpis frekvencií na štandardný výstup
    return Behaviors.same();
}
```

spracovali sme správu,
správanie sa nemení

Verzia 2: jeden aktor, výsledky do sveta



- 2 triedy správ
  - command
  - event
- musíme zaviesť **adresáta** pre odpoveď
- ale prekračujeme hranice medzi svetmi

Interakcia požiadavka/odpoveď

- request/response
  - obvykle sa deje asynchrónne
- musíme zaviesť triedu pre odpovede
- musíme vedieť, ktorému aktorovi odpovedať

do commandu vložíme odkaz na adresáta odpovede

Event: udalosť s odpoveďou

```
public static class WordFreqCalculated {
    private final Map<String, Long> frequencies;

    public WordFreqCalculated(Map<String, Long> frequencies) {
        this.frequencies = frequencies;
    }
}
```

Interakcia požiadavka/odpoveď

```
public static class GetFreq {
    private final String sentence;
    private final ActorRef<Frequencies> replyTo;

    public GetFreq(String sentence,
                   ActorRef<WordFreqCalculated> replyTo)
    {
        this.sentence = sentence;
        this.replyTo = replyTo;
    }
}
```

dátový typ správy pre udalosť

odkaz na adresáta odpovede

```
Behavior<GetWordFreq> getWordFreq(GetWordFreq command) {
    // 1. výpočet frekvencií
    // 2. odpoveď pôvodcovi príkazu: posielame event

    command.getReplyTo().tell(new WordFreqCalculated(freqs))

    return this;
}
```

spracovali sme správu,
správanie sa nemení

Ask Pattern: synchrónna komunikácia

1. ak posielame požiadavku/odpoveď v štýle RPC
   - pošleme požiadavku
   - blokujeme, kým nedôjde odpoveď
   - HTTP štýl!

Ask Pattern: synchrónna komunikácia

2. ak komunikujeme medzi svetmi OOP a aktormi
   - ak aktor odpovedá „niekomu", kto nie je z tohto sveta aktorov
   - prekračujeme hranice medzi svetmi

Ask: pýtaj sa a čakaj na výsledok

```
CompletionStage<WordFreqCalculated> result = AskPattern
    .ask(system,
        replyTo -> new GetFreq("Life is Life", replyTo),
        Duration.ofSeconds(5),
        system.scheduler());
```

aktor, ktorého sa pýtame

lehota na vyčkávanie

implicitnýsystémový plánovač úloh

Ask: pýtaj sa a čakaj na výsledok

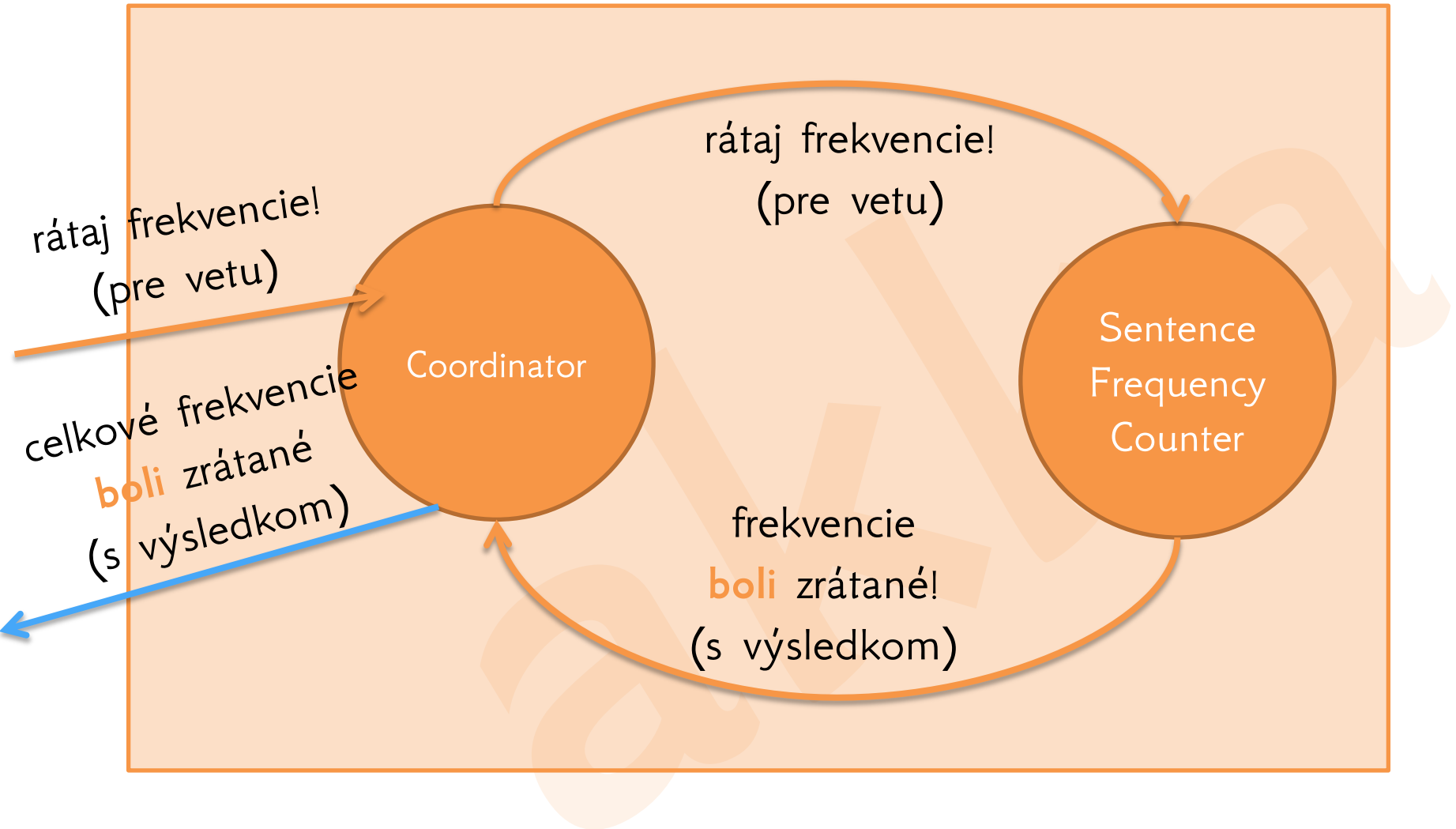
```
CompletionStage<WordFreqCalculated> result = AskPattern
    .ask(system,
        replyTo -> new GetFreq("Life is Life", replyTo),
        Duration.ofSeconds(5),
        system.scheduler());
```

message factory: funkcia, ktorá poskytne fiktívneho aktora pre príjem odpovede

Ask Pattern: synchrónna komunikácia

- v bežnej komunikácii je **ask** zbytočný
  - čaká
  - blokuje
  - mrhá prostriedkami
- prípady:
  - unit testy / demá
  - prekračovanie hraníc sveta
  - špecifické situácie, keď vieme, čo robíme

Verzia 3: dva aktory



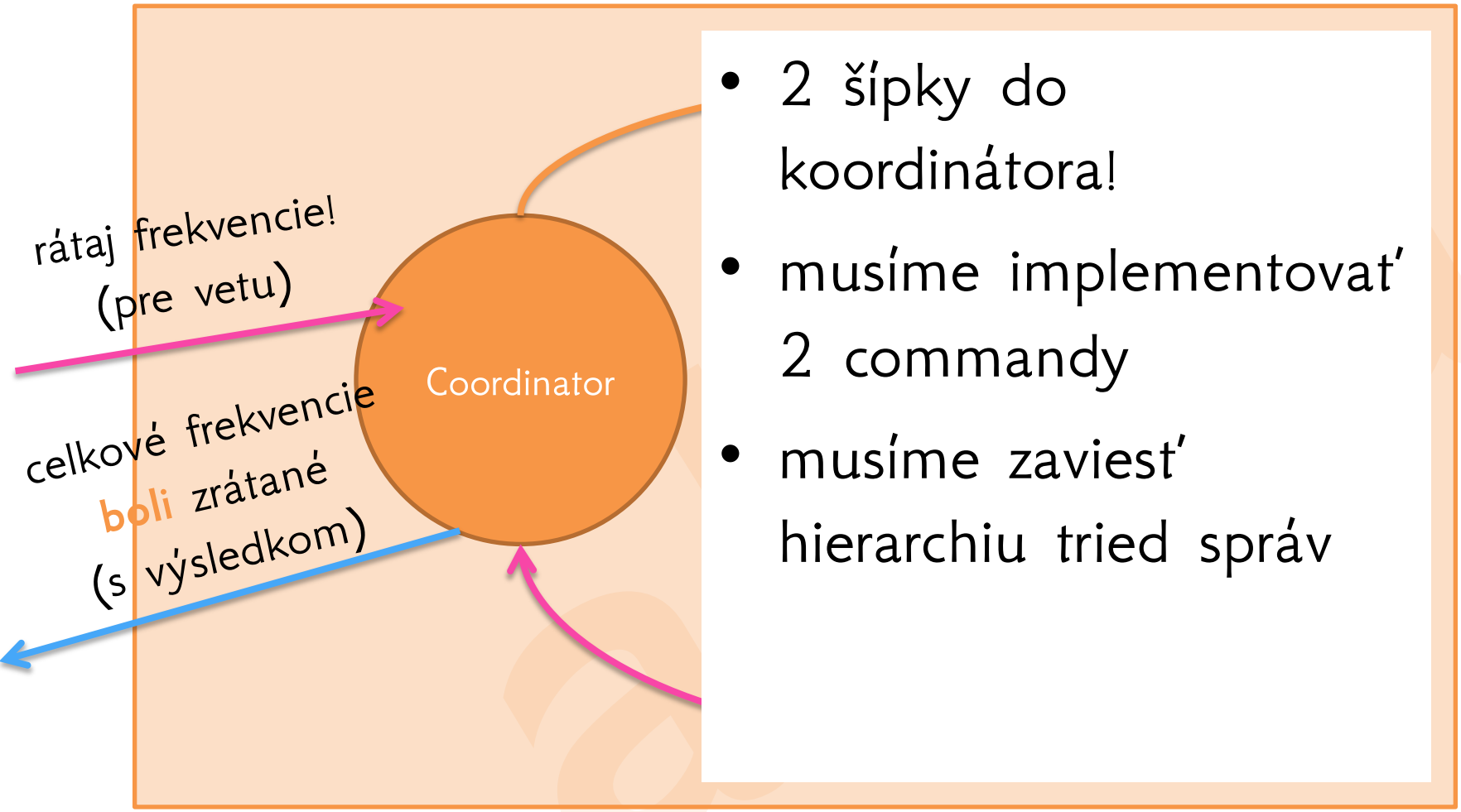
Problémy na riešenie

1. vytvoríme sadu správ pre každého aktora
2. naučíme koordinátora vytvárať workerov
3. adresujeme problém s udalosťou, ktorá má byť príkazom pre aktora
4. naučíme koordinátora posielať správy workerovi
5. naučíme koordinátora reagovať na udalosti od workera

Komunikácia medzi aktormi

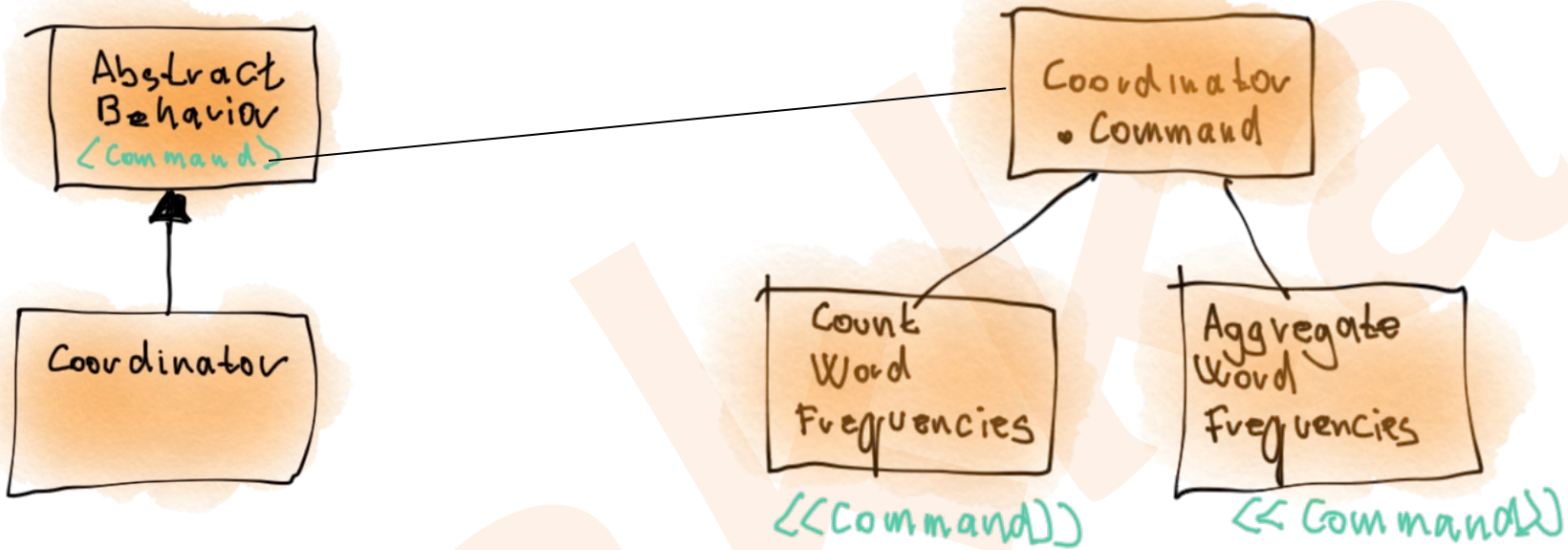
- každý aktor nech má vlastnú sadu správ
  - statické vnútorné triedy pre správy
  - best practice!
- bad practice: globálne triedy správ

Verzia 3: dva aktory



- 2 šípky do koordinátora!
- musíme implementovať 2 commandy
- musíme zaviesť hierarchiu tried správ

Hierarchia správ



Aktor tvorí aktora: spawn

Krok 2

- aktor vie vytvoriť konečný počet iných aktorov
- aktor si nesie referenciu na iného aktora

```
private Coordinator(ActorContext<Coordinator.Command> ctx) {
    super(ctx);
    this.worker = context.spawn(
        SentenceFrequencyCounter.create(),
        "frequency-counter"
    );
}
```

vytvárame aktora

ľubovoľné logické meno

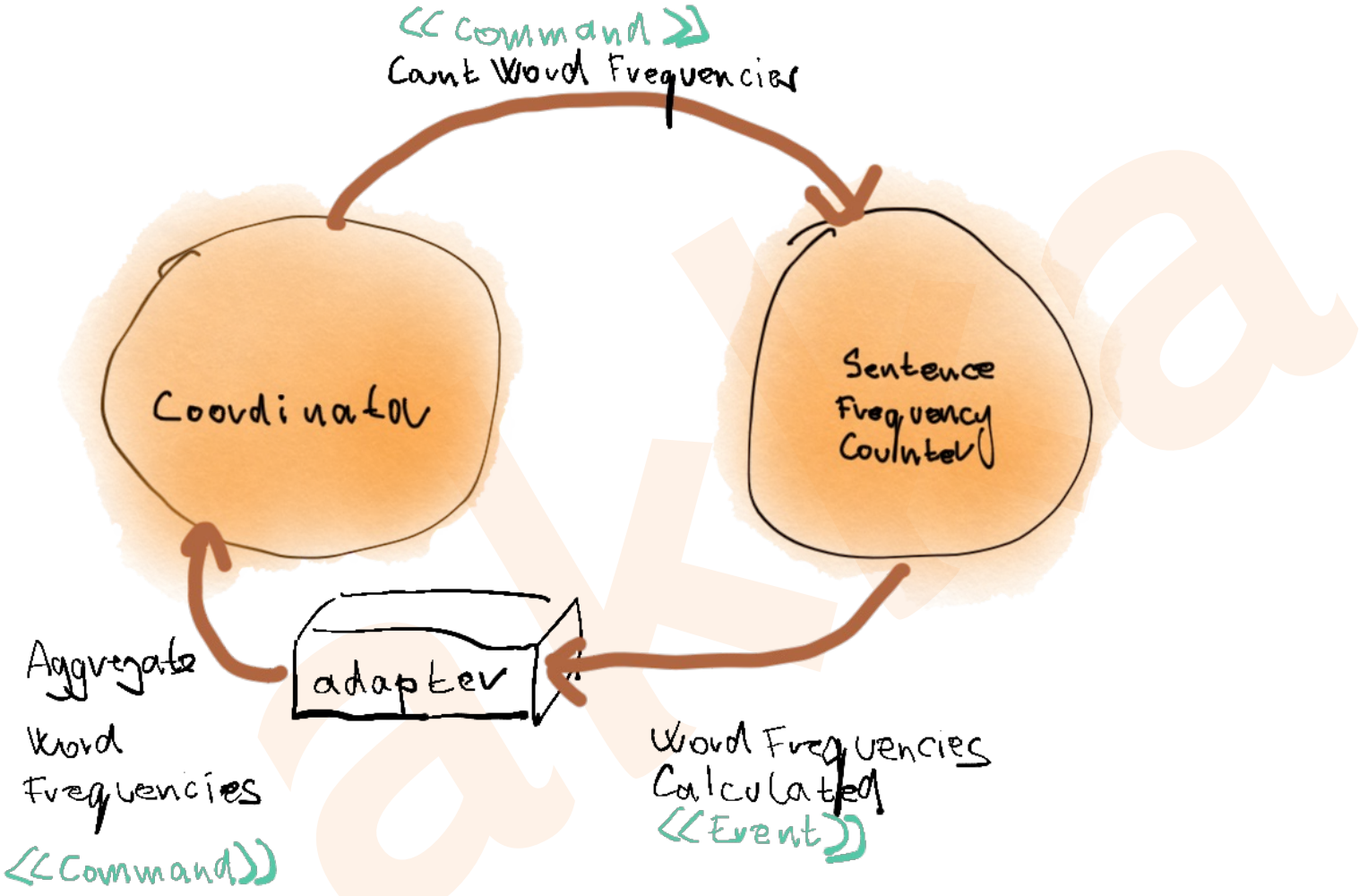
Komunikácia medzi aktormi: spawn

- aktor si nesie referenciu na iného aktora

```
private ActorRef<SentenceFrequencyCounter.GetWordFreq> worker;
```

Udalosti jedného=príkazy druhému

Krok 3



Udalosti jedného sú príkazy druhému

- koordinátor posiela reťazce do počítadla
- adresátom odpovede je **koordinátor**

- **v počítadle** nastane udalosť **WordFreqCalculated**
- **koordinátor** očakáva príkaz **AggregateWordFreq**

message adapter:
adaptuje jednú správu na iný

Koordinátor a referencia na aktora

```
ActorRef<WordFreqCalculated> messageAdapter;

private Coordinator(ActorContext<Command> ctx) {
    super(ctx);
    //...
    messageAdapter =
        ctx.messageAdapter(
          WordFreqCalculated.class,
          freq -> new AggregateWordFreq(freq.getFrequencies())
        );
}
```

prevádzame event na command

Zasielanie správ z koordinátora

Krok 4

```
SentenceFrequencyCounter.GetWordFreq command
    = new SentenceFrequencyCounter GetWordFreq(
          "Life is Life,
          messageAdapter
      );

this.worker.tell(command);
```

adaptér sa postará o prevod odpovede (udalosti) na príkaz

Reakcia na nový príkaz

Krok 5

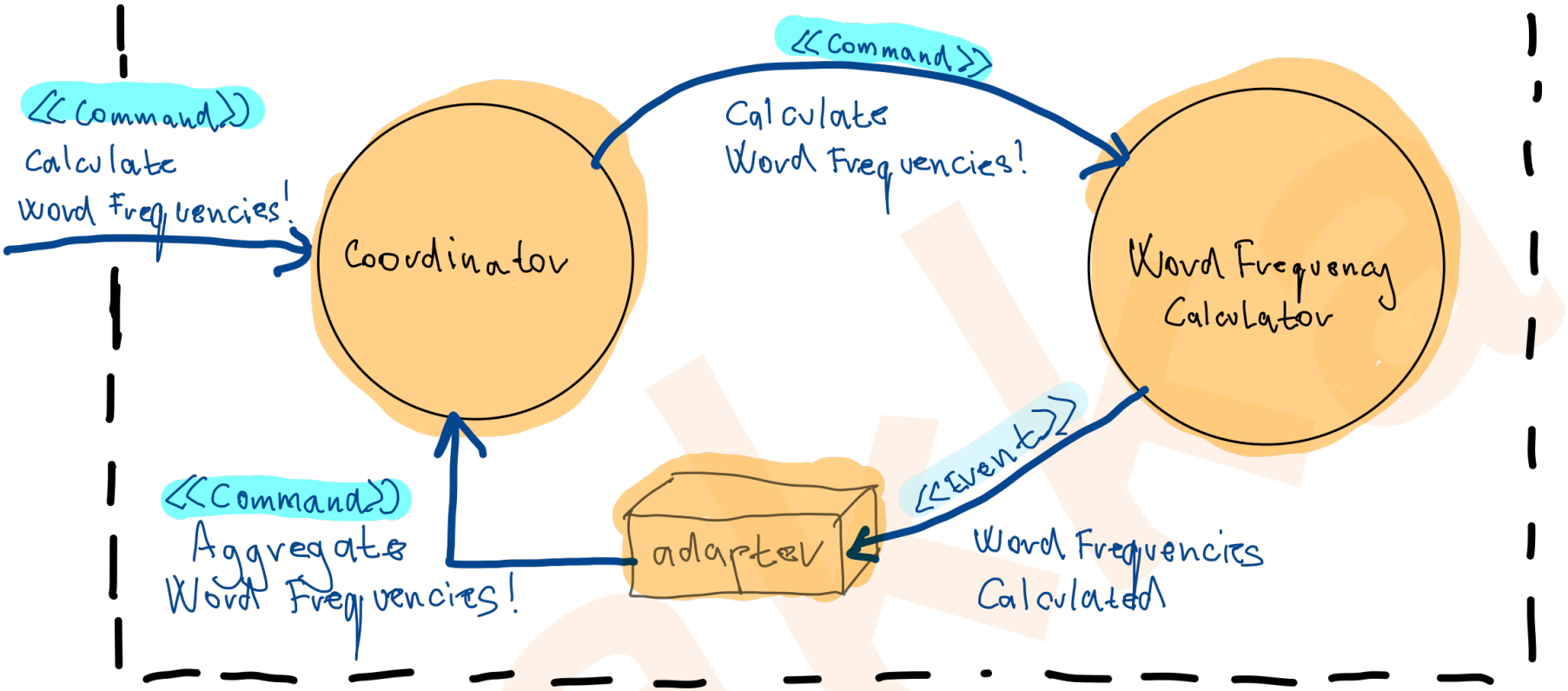
```
public Receive<Coordinator.Command> createReceive() {
    return newReceiveBuilder()
            .onMessage(AggregateWordFreq.class,
                    this::aggregateWordFreq
            )
            ...
```

bežná obslužná metóda príkazu

Do budúcna

- Ako riešiť škálovanie?
  - Máme 1 koordinátora a len jedno počítadlo.
- Kedy systém skončí?
  - Systém neustále beží a vyčkáva na ďalšie správy

Celý stav


«Command»
Calculate
Word Frequencies!
Coordinator
«Command»
Calculate
Word Frequencies!
Word Frequency
Calculator
«Command»
Aggregate
Word Frequencies!
adapter
«Event»
Word Frequencies
Calculated

Škálovanie aktorov

```
context.spawn(createPool(), "workers");
```

createPool()

```
return Routers.pool(3, WordFrequencyCounter.create())
```

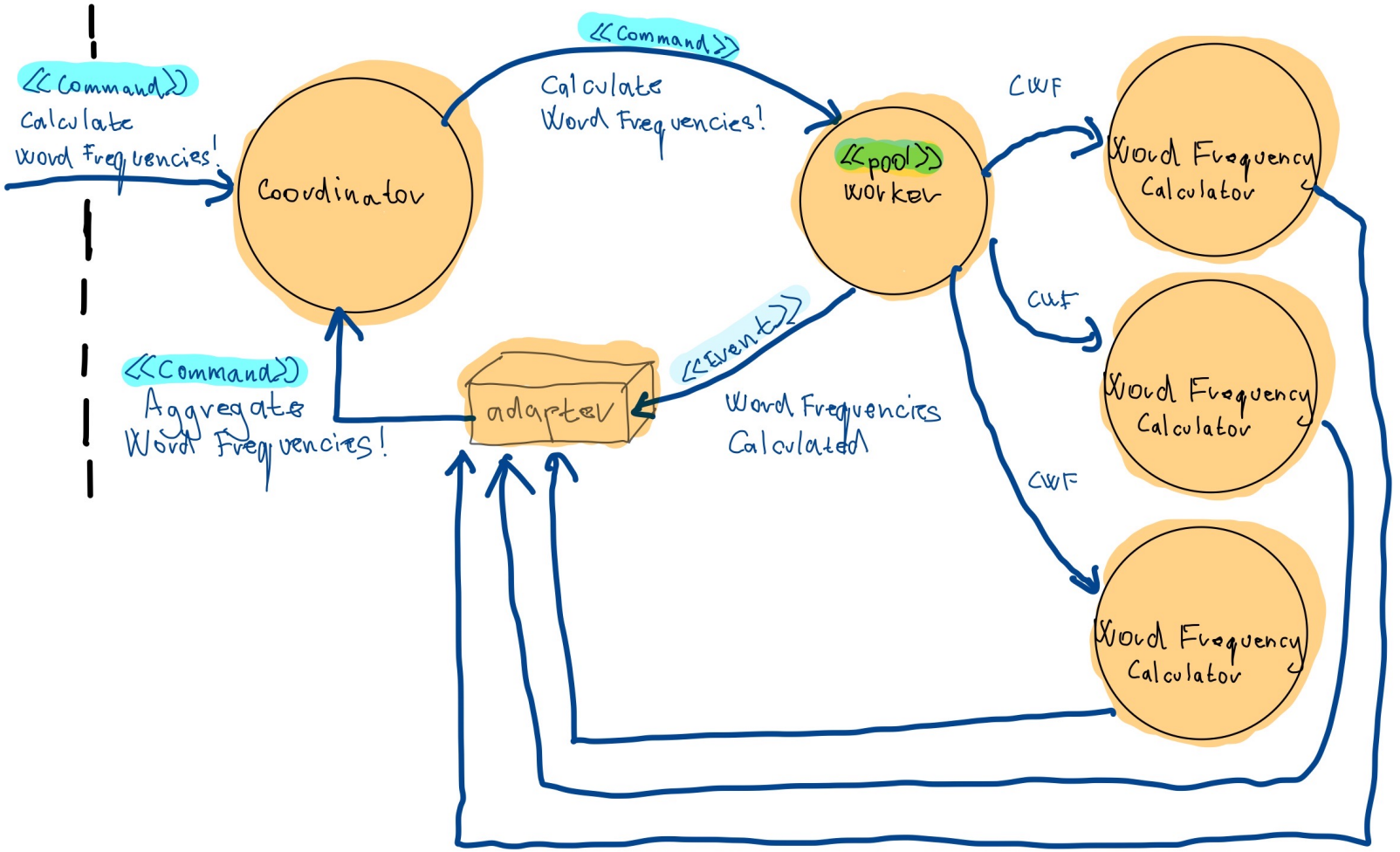
Round Robin Pools

- pool:
  - fixný zoznam aktorov
- router smeruje správy do aktorov vypočítankou

3 aktori s routerom typu round-robin

```
return Routers.pool(3, WordFrequencyCounter.create())
```

Stav s poolom



Round Robin Pool

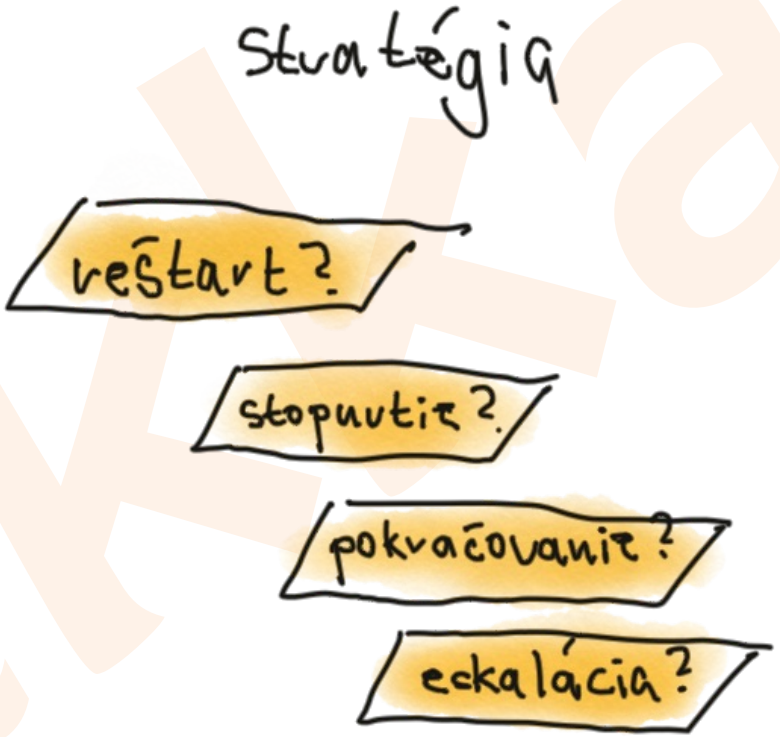
- pool sa tvári navonok ako jeden aktor
- správy idú do routera
- odpovede idú do cieľa v **evente** napriamo

Failover

- aktor môže zlyhať výnimkou
- štandardné správanie: **aktor sa stopne**

Supervision

- aktora obalíme supervíziou
- správanie, ktoré
  - odchytí výnimku
  - zvolí stratégiu



Supervízia

```
Behaviors
    .supervise(WordFrequencyCounter.create())
    .onFailure( IllegalStateException.class,
                SupervisorStrategy.restart())
```

supervízia počítadla
+
pravidlá

Signály

- aktor vie prijímať špeciálne správy
- súvisia so životným cyklom
- typicky:
  - PostStop: po zastavení inštancie aktora
  - PreRestart: pred reštartom aktora
  - Terminated: sledovaný aktor sa zastavil

Signály

```
return  newReceiveBuilder()
    .onSignal(PreRestart.class,  this::onPreRestart)
    .onSignal(PostStop.class,  this::onPostStop)
    .build();
```

Signály: reštart

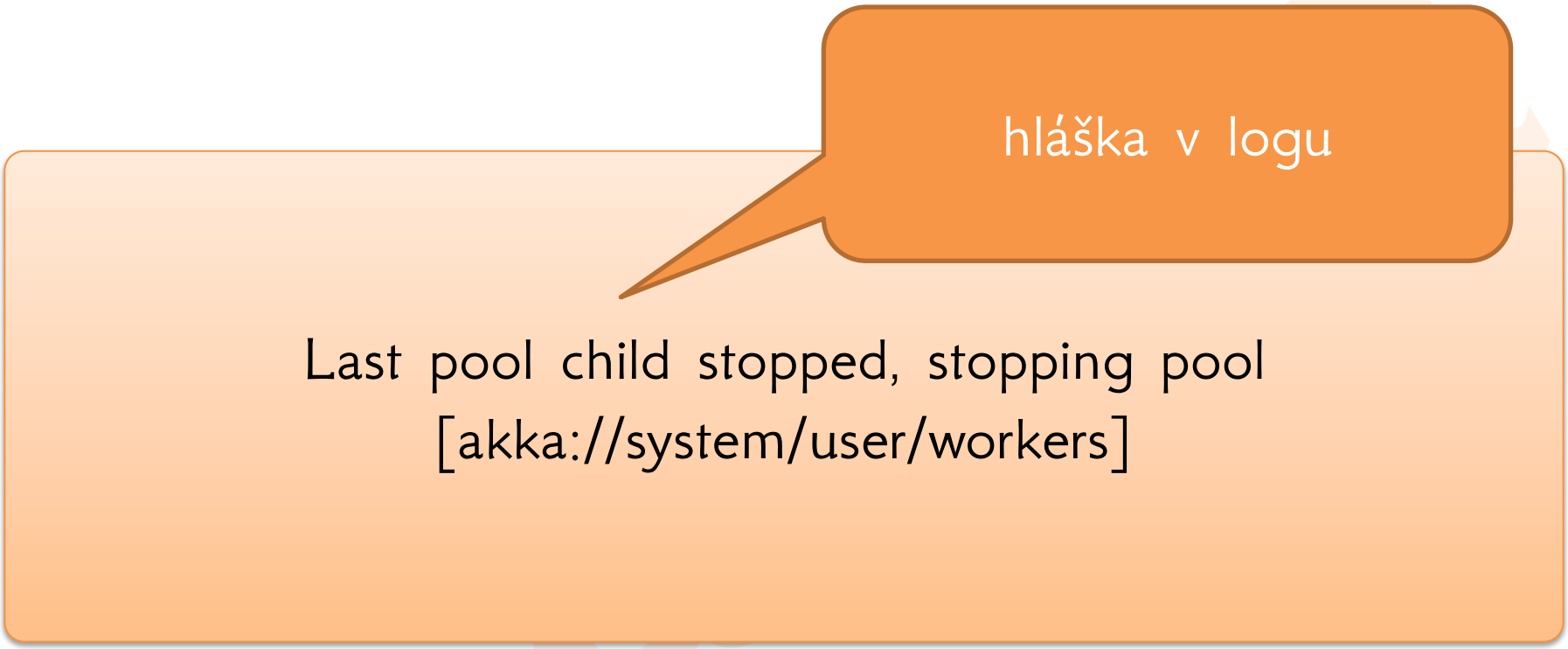
- reštart:
  - odstráni aktora
  - vytvorí novú inštanciu
- pozor: reštart nie je stop->štart!
  - signál PostStop sa pri reštarte nezasiela

Signály: stopnutie

- stopnutie:
  - zastaví aktora
  - ak je v poole, notifikuje router!
- pošle signál PostStop

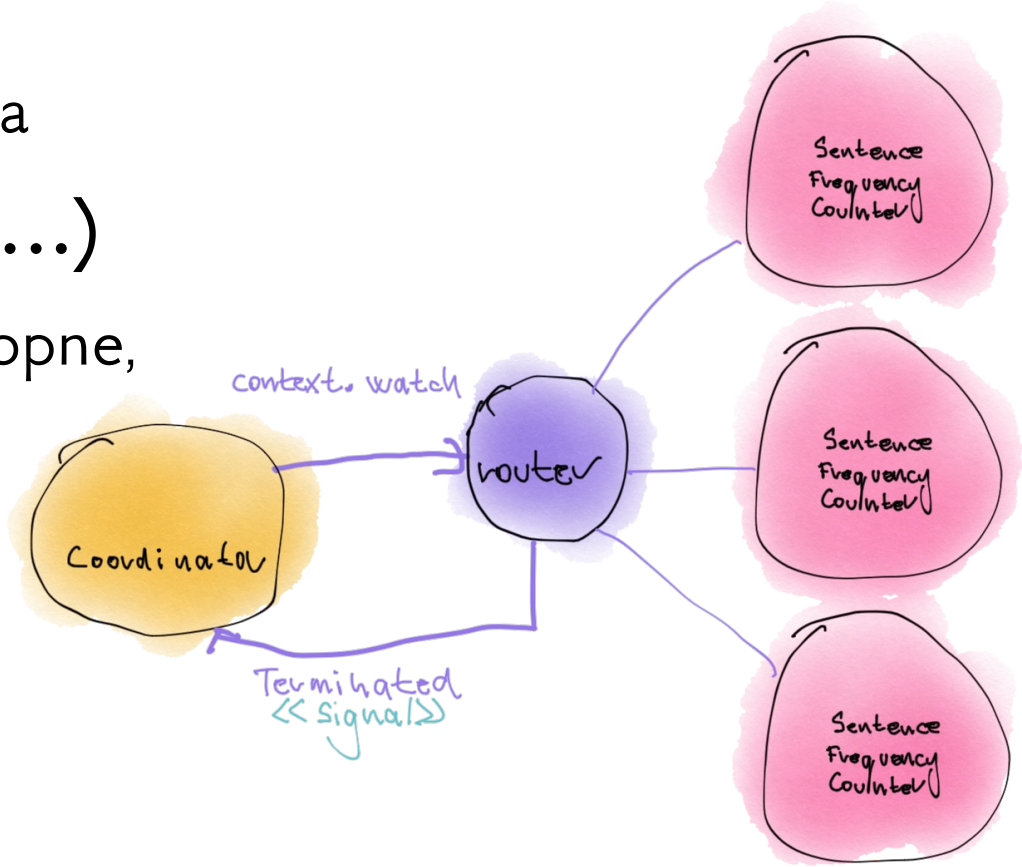
Router a stopnutie

- ak sa stopnú všetky aktory za routerom, router sa stopne tiež



Death Watch

- aktor vie zistiť, že sa podriadený aktor zastavil
- context.watch()
  - začne sledovať aktora
- onSignal(Terminated…)
  - ak sa podriadený stopne, dorazí signál



Death Watch

- typické príklady:
  - aktor sa zastaví, ak sa zastaví router
  - aktor sa zastaví, ak sa zastaví jeho spawnutý worker
  - sledovanie aktorov po sieti

Kedy sa oplatí Death Watch?

- ošetrí prípady:
  - korektného ukončenia ak sa aktor zastaví sám cez Behavior.stopped()
  - ak je aktor zabitý Akkou
  - ak sa na aktora nemožno pripojiť cez sieť

Graceful Shutdown: korektné vypnutie

1. ukončíme workerov
2. ukončíme router
3. ukončíme koordinátora
4. ukončíme systém

Graceful Shutdown: korektné vypnutie

- naši aktori musia dostať správu do protokolu
- obohatíme **Command**-y aktorov
  - Coordinator: Eof
  - WordFrequenciesCalculator: Shutdown

Graceful Shutdown: korektné vypnutie

- router má možnosť vyslať správu všetkým aktorom
- broadcast

príkaz Shutdown sa pošle všetkým aktorom v poole

```
return Routers.pool(...)
    .withBroadcastPredicate(
        WordFrequencyCounter.Shutdown.class::isInstance
    );
```

Graceful Shutdown: korektné vypnutie

1. ukončíme workerov
   - pošleme im Shutdown z protokolu

Coordinator#onEof

```
private Behavior<Command> eof(Eof eof) {
    this.workers.tell(new WordFrequencyCounter.Shutdown());
    return Behaviors.same();
}
```

Graceful Shutdown: korektné vypnutie

1. ukončíme workerov
   - pošleme im Shutdown z protokolu
2. ukončíme router
   - ukončí sa sám s posledným stopnutým workerom
3. ukončíme koordinátora
   - cez death-watch sledujeme router
   - po signáli Terminated zastavíme koordinátora
4. ukončíme systém
   - stopnutý koordinátor vedie k stopnutiu systému

Graceful Shutdown: 2. ukončíme router

Router sa ukončí sám so stopnutím posledného workera

Last pool child stopped, stopping pool [akka://system/user/workers]

Graceful Shutdown: 3. ukončíme koordinátora

- cez death-watch sledujeme router
- po signáli Terminated zastavíme koordinátora

```
context.watch(this.workers)
```

```
onSignal(Terminated.class, ...)
```

Graceful Shutdown: 4. ukončíme systém

- aktorový systém má pod sebou koordinátora
- so stopnutím koordinátora sa stopne systém

autohláška z logov

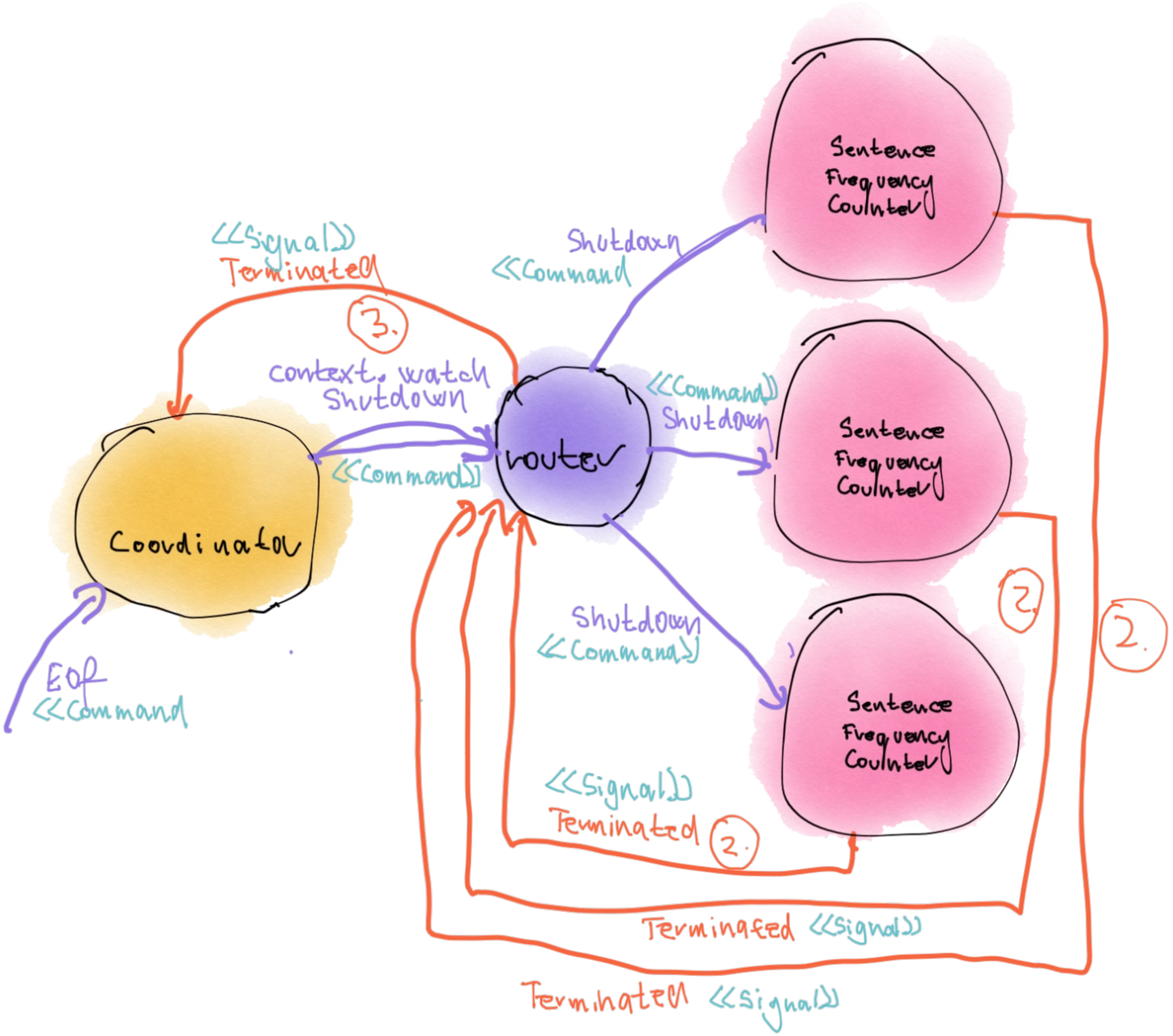
received AutoReceiveMessage
Envelope(Terminated(Actor[akka://system/user/workers)

Použitie

- do systému pošleme vety a na záver Eof

explicitný shutdown

```
for (String sentence : sentences) {
    system.tell(new Coordinator.CountWordFrequencies(sentence));
}
system.tell(Coordinator.Command.EOF);
```


Sentence Frequency Counter
Sentence Frequency Counter
Sentence Frequency Counter
<<Signal>> Terminated
3.
Shutdown <<Command
context.watch Shutdown
<<Command>> Shutdown
<<Command>>
router
Coordinator
EOF <<Command
Shutdown <<Command>>
2.
2.
<<Signal>> Terminated
2.
Terminated <<Signal>>
Terminated <<Signal>>

Otázky?