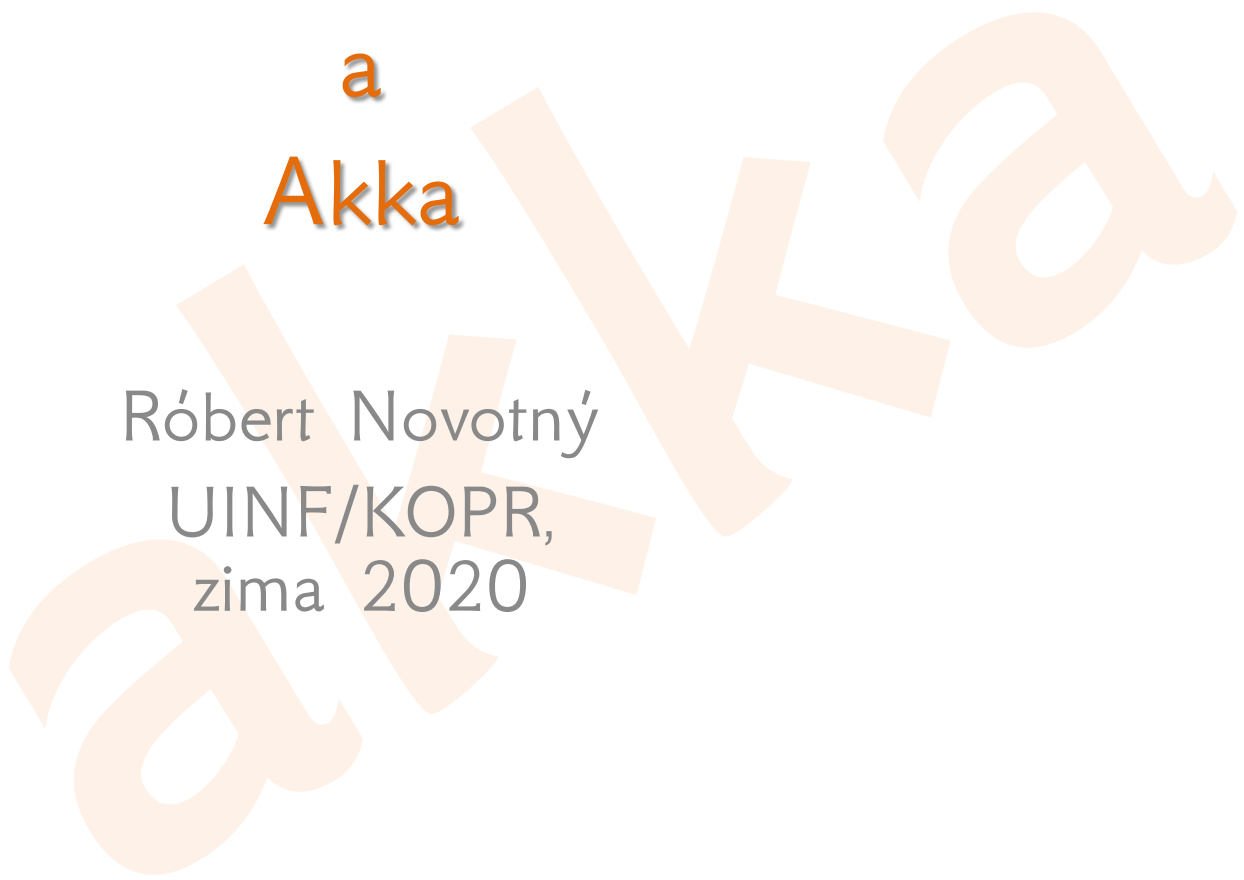
Aktory
a
Akka

Róbert Novotný
UINF/KOPR,
zima 2020



Tony dát a hordy klientov

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

Škálovateľnosť má hranice

Škálovateľnosť

- vertikálne:
  - zvyšujeme frekvenciu,
  - dodávame RAM…
- horizontálne:
  - pridávame servery
  - pridávame procesory
  - pridávame jadrá

Trampoty vertikály

- frekvencia CPU je na maxime
- vlákna neškálujú
  - 10000 vlákien?
- vlákna vyžadujú ninje
  - zámky, monitory, samé deadlocky, race conditions
  - nereprodukovateľné problémy
  - veľký kognitívny load
  - stále slabá podpora nástrojov

Základné riešenie ťažkého kalibru

- zrušme prístup k zdieľaným dátam a bude dobre!
  - kde nie sú zdieľané dáta, nie sú deadlocky, prepisy, ...
- žiadne zdieľané inštančné premenné...
- ako potom programovať?

Potreba iných modelov!

Trampoty horizontály

- komunikácia po ceste sa kazí
  - káble sa sekajú, datacentrá umierajú, servery horia, aplikácie lagujú
- konzistencia dát
  - synchronizácia dát medzi uzlami
- transportné protokoly
  - štýl správ a forma ich obsahu

Potreba iných modelov!

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

- správy sú odosielané **asynchrónne**
- nečaká sa na prijatie
  - **fire-and-forget**
- nezáleží na poradí prijatých správ
  - inšpirácia z packet-oriented systems (UDP?)
- v praxi: podporuje sa aj synchronicita

Správa

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

Aktorový model je
distribuovaný i paralelný!

Ďalšie črty aktorov

- hierarchia aktorov
  - vhodné pri rozdeľovaní práce na podúlohy
- jednoznačná identifikácia
  - pomocou URI (hostname, port, cesta)
- a iné
  - voliteľná synchronicita posielania správ
  - podpora transakcií (model STM)

Implementácie aktorového modelu

- Erlang
  - funkcionálny jazyk
  - namiesto zdieľaného stavu odovzdávanie správ
- Akka
  - open source framework pre aktorov
  - implementácie: Java / Scala

Use-Case

- Erlang:
  - telefónna ústredňa: Ericsson (198x/199x)
  - infraštruktúra mobilných sietí (aj dnes)
  - WhatsApp
- Akka:
  - množstvo výkonných služieb na pozadí

Jestvujú výkonné Akka systémy

- SOAP/REST/HTTP endpointy
- prijímateľ údajov zo siete
- perzistencia dát
- paralelné a distribuované výpočty
- dávkové spracovanie a BigData
- microservices,
- transakčné operácie,
- mediácie a integrácie

Aktory v Akke

- Java + Typed Actors + Object-Oriented
  - ukážeme si tento model
    - vieme Javu
    - vieme OOP
    - nevieme aktory
- existuje 3 x 2 rozličných spôsobov implementácie

Akka Typed

- moderný spôsob písania aktorov
  - od Akka 2.6 (nov. 2019)
- aktory sú podobní konečnostavovým automatom
  - reagujú na správy
  - menia svoje správanie (behavior)

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))
```

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);
    }
```

oddedený konštruktor

```
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!

Thread-safety aktorových tried

- aktory v Akke fungujú nad Java Virtual Machine
- interne využívajú vlákna
  - tie nás v Akke nezaujímajú
  - je to iný svet
- výnimočne musíme prekročiť hnusný svet vlákien a krásny svet aktorov

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**

Buffrovanie správ: áno alebo nie?

- jeden názor: správy sa nebufferujú: všetko sa deje naraz
- kompromisný: „správy sa bufferujú v éteri"
- prakticky [Akka]: správy sa radia do buffera / frontu

Mailboxy a buffrovanie

- aktor má mailbox v ktorom sa kopia správy
  - každý aktor má vlastný
  - nedajú sa zdieľať
- správy defaultne radené podľa času odoslania
- možno využiť prioritný mailbox:
  - radenie podľa priority
- mailbox je front!
  - aktor vidí **len** jeho začiatok

Dôležité rysy aktora

- aktor spracováva v danom čase **len jednu správu**
  - ostatné sa kopia vo fronte
  - dlhotrvajúce spracovanie správy znamená dlhšie čakanie!
- správy z aktora A do aktora B **garantujú poradie** doručenia
  - ak nemáme špeciálny typ mailboxu
  - nespoliehajme sa na poradie doručenia správ z rozličných aktorov!

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

Analýza v Akke

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

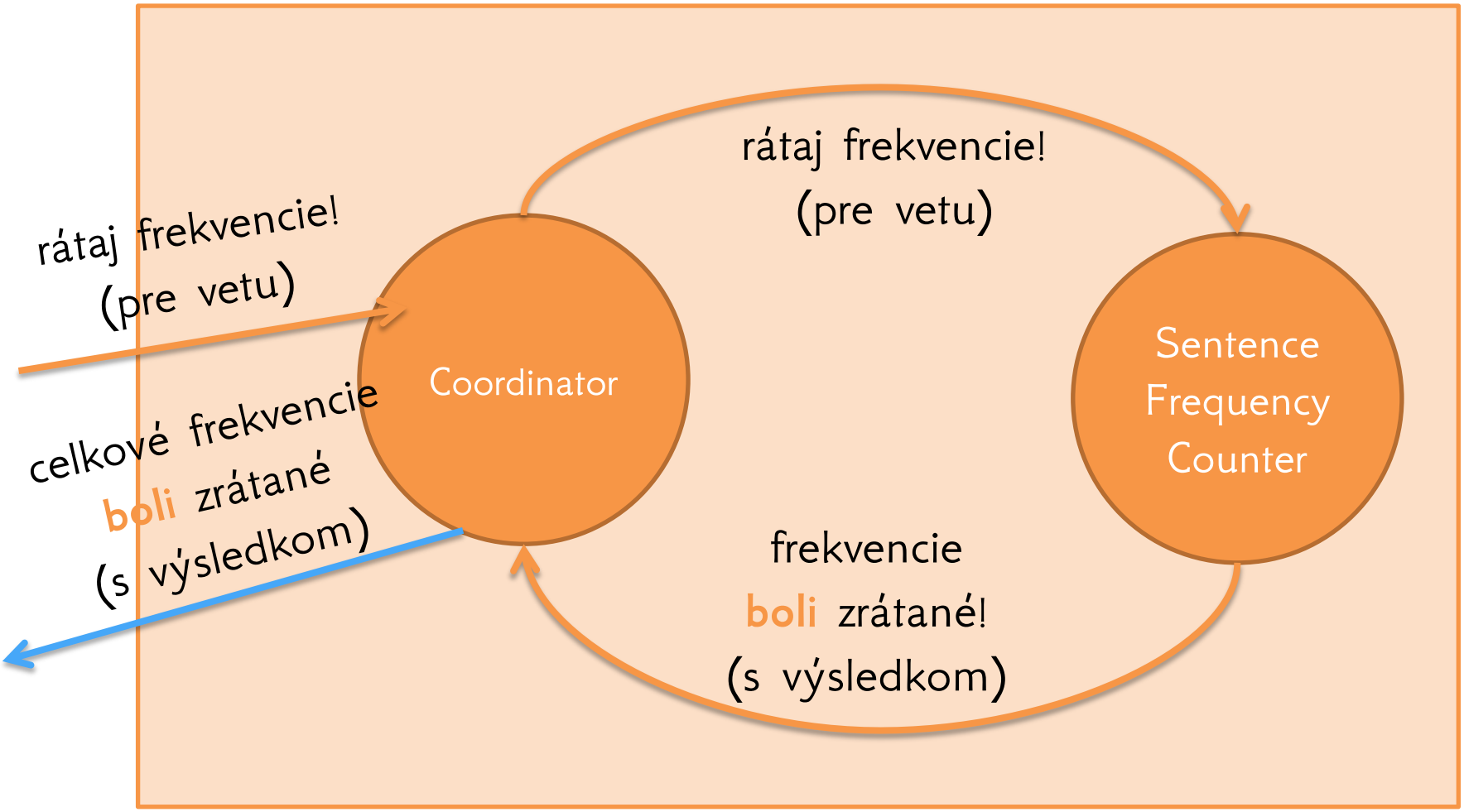
spočítaj frekvencie!

...

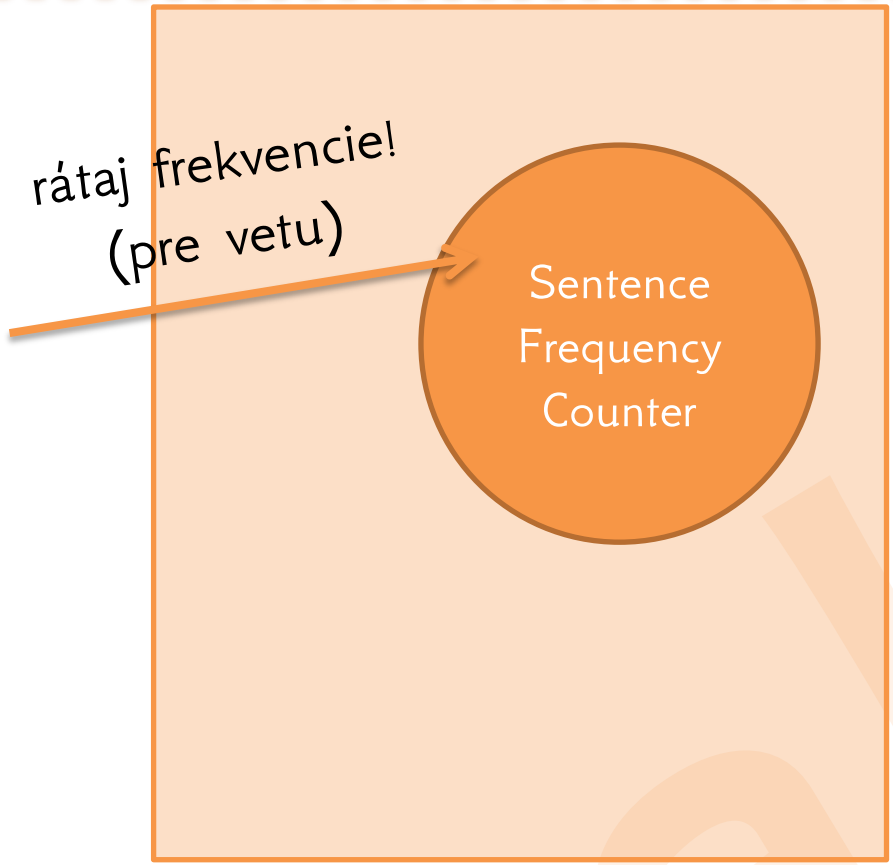
frekvencie boli spočítané

Príkazy a udalosti

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


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

Verzia 1: jeden aktor



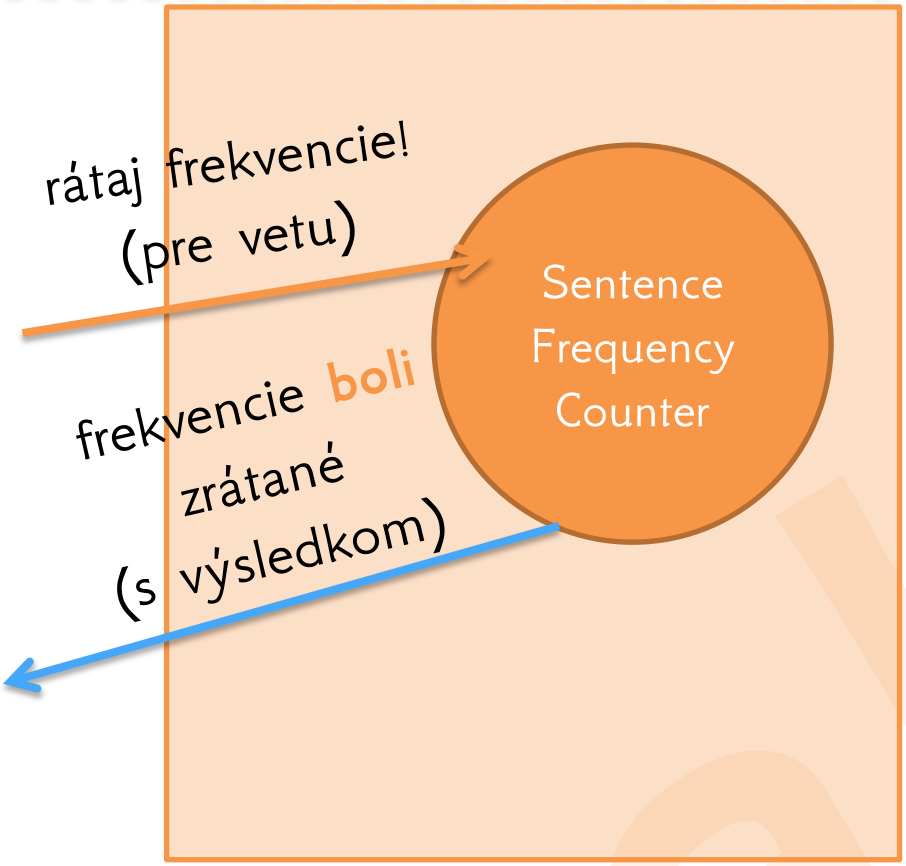
- 1 trieda správy
  - command
- konvencia: správy ako statické vnútorné triedy
- ešte neagregujeme celkové frekvencie
- výstup posielame na stdout

```
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 this;
}
```

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

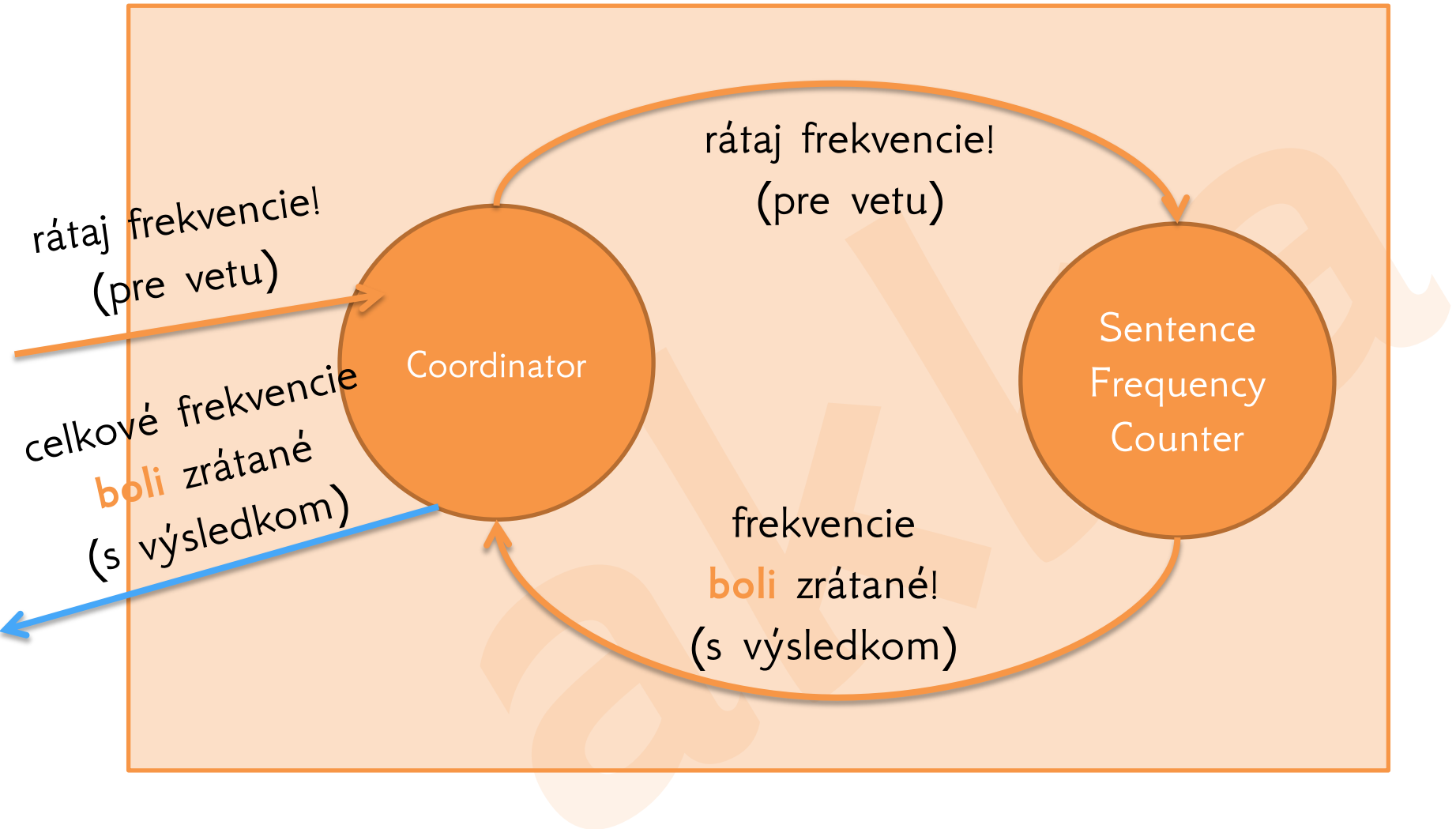
```
CompletionStage<WordFreqCalculated> result = AskPattern
    .ask(system,
        replyTo -> new GetFreq("Life is Life", replyTo),
        Duration.ofSeconds(5),
        system.scheduler());
```

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

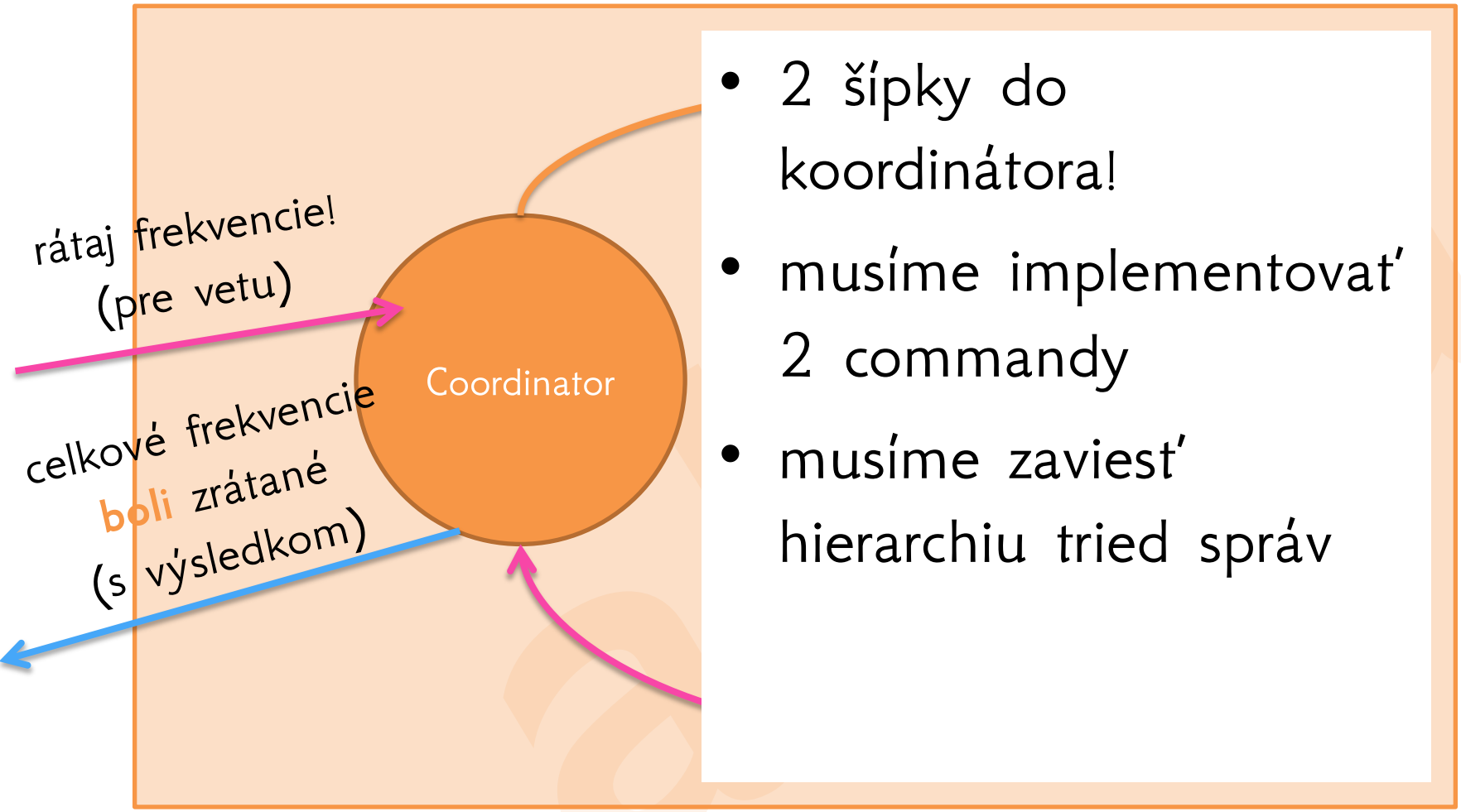
1. vytvoríme sadu správ pre každého aktora
2. naučíme koordinátora vytvárať workerov
3. adresujeme problém s udalosťou, ktorá má byť príkazom pre aktora
4. naučíme koordinátora posielať správy workerovi
5. naučíme koordinátora reagovať na udalosti od workera

Komunikácia medzi aktormi

Krok 1

- 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

- interface Coordinator.Command
  - class Coordinator.CalcWordFreq
  - class Coordinator.AggregateWordFreq

```
class Coordinator extends AbstractBehavior<Coordinator.Command>
```

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

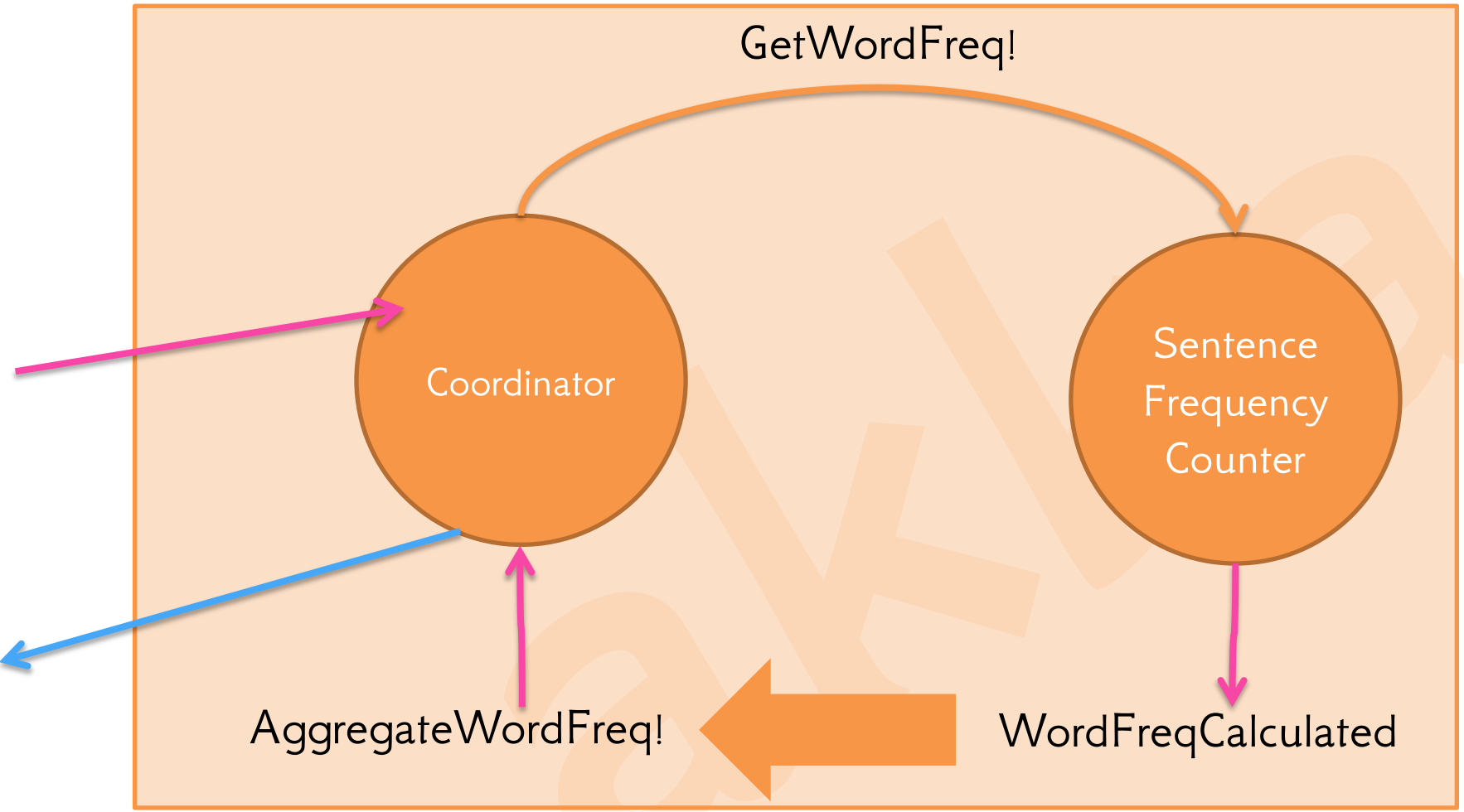
Komunikácia medzi aktormi: spawn

- 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

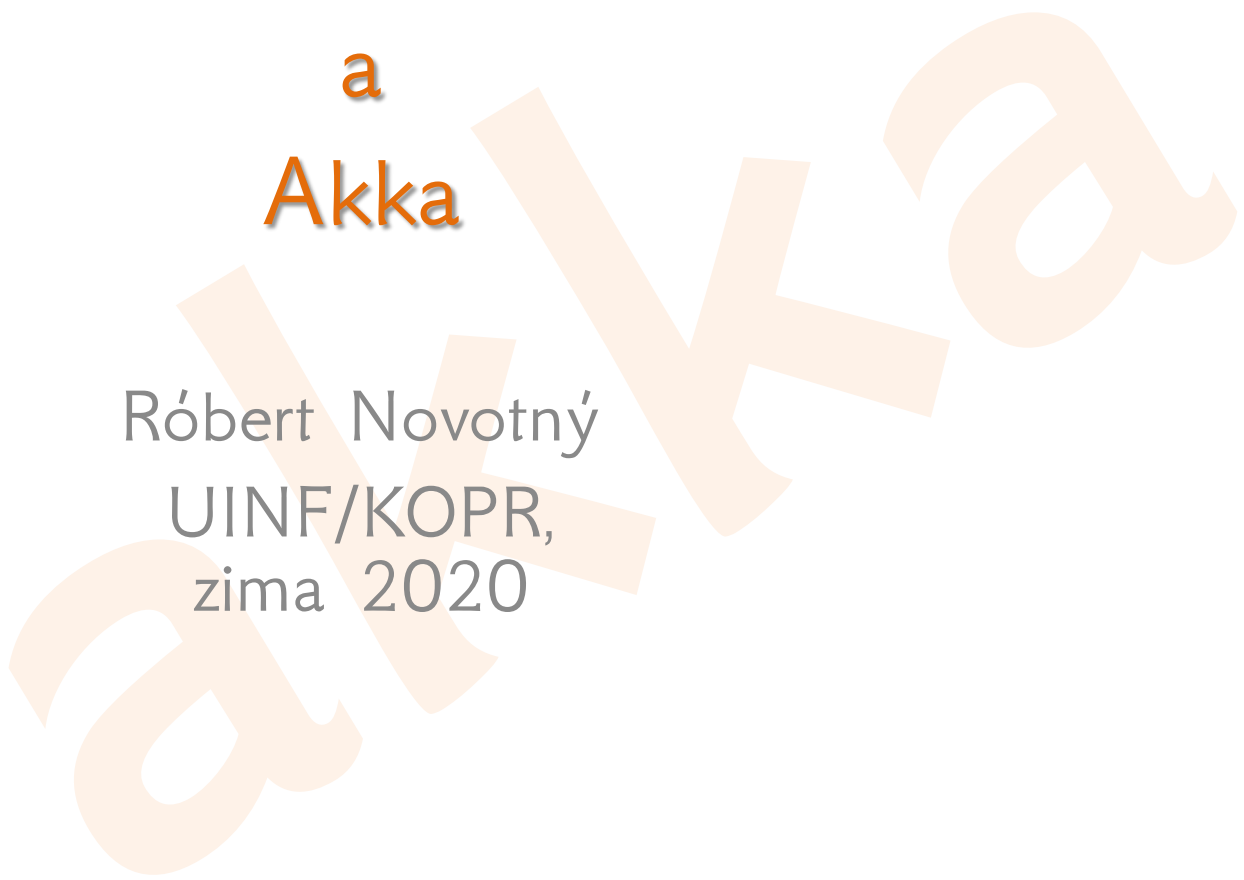
Do budúcna

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

Otázky?

Aktory
a
Akka

Róbert Novotný
UINF/KOPR,
zima 2020



Súčasný stav