Aktorzy

współbieżność sterowana asynchronicznymi wiadomościami

JAG
ETI PG

by Arkadiusz Hiler

O czym?

- Problematyka i wprowadzenie do tematu.
  - przechodzenie między stanami,
  - niemodyfikowalność,
  - krótko o kolekcjach .
- Aktorzy – sterowanie zdarzeniami.
  - Scala,
  - Akka - „integracja” z Javą.
- Podsumowanie.

Skomplikowane przechodzenie między stanami

Java

Mamy klasę z:

```
int wartosc;

public synchronized void setWartosc(int w) { this.wartosc = w; }
public synchronized int getWartosc() { return this.wartosc; }
```

Modyfikujemy:

```
int tmp = x.getWartosc();
//operacje na tmp
x.setWartosc(tmp);
```

Co się stanie gdy będziemy to robić w wielu miejscach na raz? Nieprzewidywalne.

Wszystkie operacje trzeba umieścić w blokach **synchronized**.

Mutexy, semafory, monitory, locki... Podatność na błędy jest ogromna.

Scala
uzupełnienie informacji

Niemodyfikowalność

Scala

Mamy między innymi:
Listy, kolejki, wektory, tablice, sekwencje, stosy, mapy w różnych wydaniach.
Więcej w linkach.

Ogólne kolekcje to tylko traity w:
`scala.collection._`

Implementacje w dwóch wydaniach:
`scala.collection.immutable._`
`scala.collection.mutable._`

Predef zawiera aliasy dla wersji niemodyfikowalnych = są domyślne.
Ukłon w stronę programowania funkcjonalnego

Immutability preferowane WSZĘDZIE we współbieżności. Omija sporo problemów z synchronizacją.

Kolekcje w Scali

Tworzenie klasyczne:

```
new scala.collection.mutable.Queue[Int]
```

Przez companion object (apply):

```
Set(1,2,3) //scala.collection.immutable.Set[Int] = Set(1, 2, 3),typ automatycznie
scala.collection.immutable.Queue[Int](1,2,4) //dla mutable można puste ()
```

Kolekcja dla wszystkiego:

```
scala.collection.mutable.Queue[Any]()
```

Zarządzanie:

```
1 :: List(2,3,4) //List[Int] = List(1, 2, 3, 4)
queue.enqueue(2) //Unit dla mutable, immutable nowa kolejka
queue.dequeue //wartość dla mutable, tuple (wartość, nowa kolejka) dla immutable
//haskellowo, odsyłam do dokumentacji po więcej
```

Iteratory:

```
val collection = List(1,2,3)
val it = collection.toIterator //Iterator[Int] = non-empty iterator
it.hasNext //Boolean = true
it.next //Int = 1
```

Aktorzy
wracamy do tematu

Aktor

Jednostka obliczeniowa, która otrzymuje wiadomości. W reakcji na wiadomość może:

- wysłać skończoną liczbę wiadomości do innych aktorów,
- (możemy odpowiadać aktorowi od którego dostaliśmy adres),
- stworzyć skończoną liczbę nowych aktorów,
- wybrać jak się zachować przy następnej wiadomość jaką dostanie.

Co warto wiedzieć:

- projektowane w latach 70 na systemy z wieloma mikroprocesorami,
- zyskują na popularności dzięki procesorom wielordzeniowym,
- formalizmy - powstało kilka algebr aktorów.

Aktor

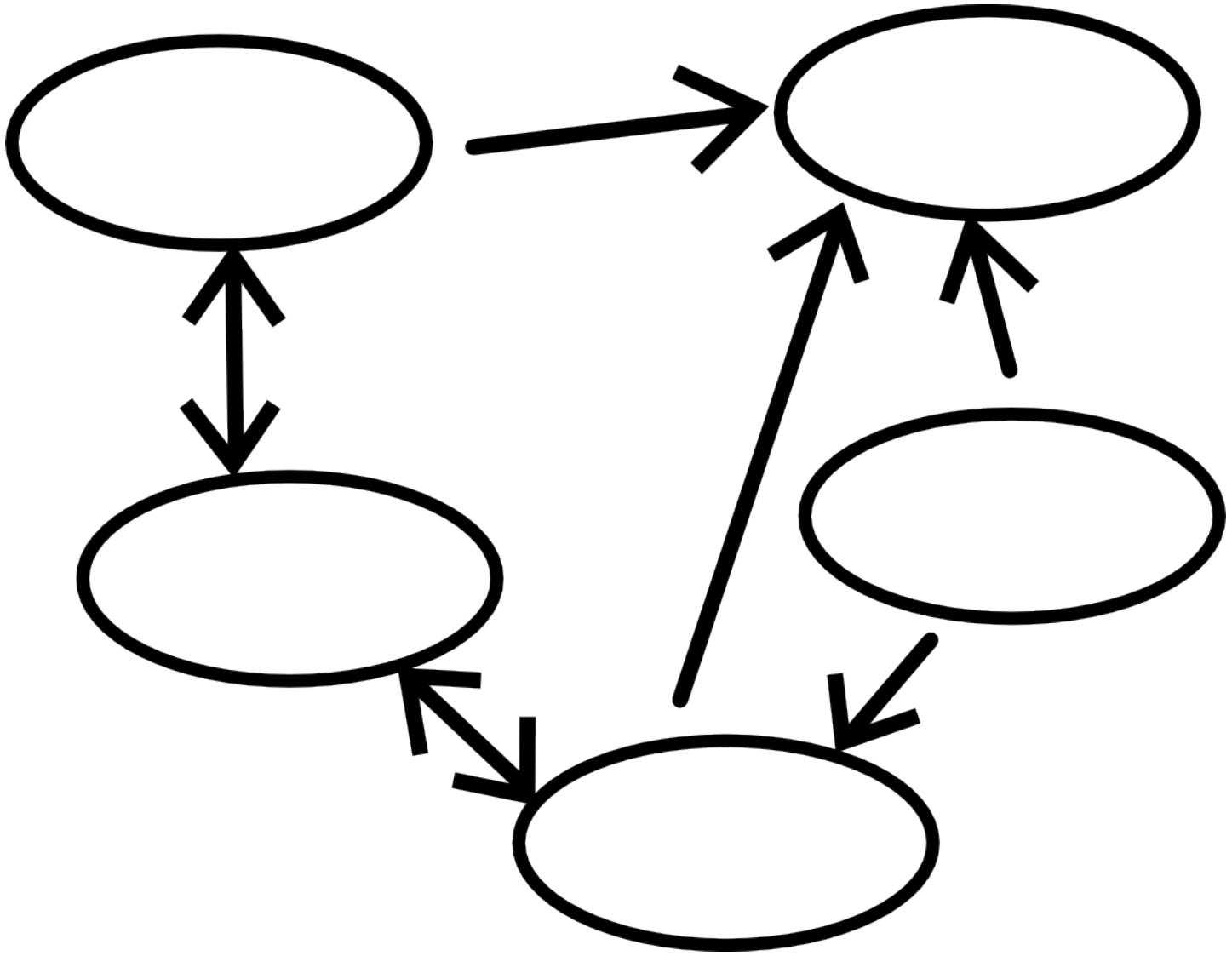
W praktyce (nasz przypadek, programowanie):

- aktora można traktować (najłatwiej) jako pojedynczy wątek,
- kolejka wiadomości na wejściu (przesyłane asynchronicznie),
- za wiadomości służą obiekty niemodyfikowalne,
- wysyłanie nieblokujące, odbieranie blokuje,
- są logicznie aktywni,
- działają niezależnie, „w sieci”, niekoniecznie lokalnie,
- można ustawiać planery w zależności od potrzeb.

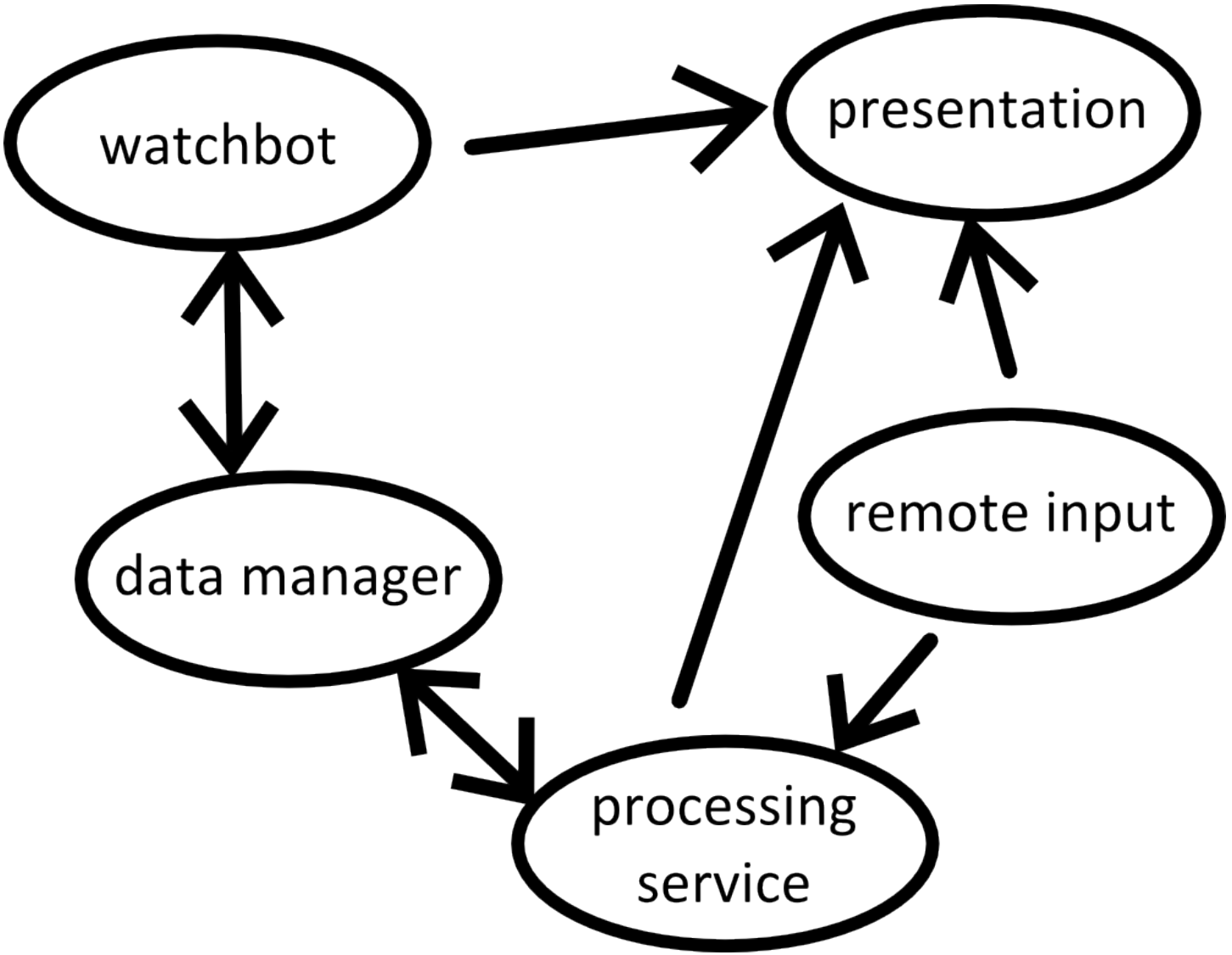
W oparciu o aktorów można rozumieć:

- pocztę - konto jako aktor, adres e-mail adresem aktora,
- web services – analogia oczywista.

Można tak



Przykładowo



Podstawy w Scali

Tworzenie aktora:

```
extends/with scala.actors.Actor //trait, implementacja act()
start() //startuje
exit() //zatrzymuje aktora, używane z wewnątrz, wiadomości po kolejkowane
```

Łatwiej:

```
scala.actors.Actor.actor { kod }
```

Wysyłanie wiadomości:

```
odbiorca ! wiadomosc //proste wysyłanie wiadomości, nie blokuje
odbiorca !? (timeout, wiadomosc) //do timeoutu oczekujemy na odpowiedź
reply(wiadomosc) //odpowiadamy nadawcy
aktor_przekazywacz.send(wiadomosc, odbiorca) //przekazujemy przez
![a](kanał, wiadomość) //case class, dopasowuje msg z !? po timeout
```

Odbieranie:

```
receive { } //wykonuje raz, jeśli trzeba - dać w pętli, zwraca odpowiedź
receiveWithin(timeout) { } //to samo, ogranicza czas (msg TIMEOUT), lepsze
react {} //wątki przyznawane dynamicznie z puli w razie potrzeby, doczytać
```

w bloku dawanym do receive wykonujemy pattern matching, od razu **case**

Przykład

Punkt wejścia – szukanie dzielników:

```
def divisors(n: Int) = {
  (Set[Int]() /: (1 to n)) {
    (set, i) => if (n % i == 0) set + i else set
  }
}
```

Mamy procesor z 6 rdzeniami i HyperThreading. Jak zrobić z tego użytek?

Wersja na przedziałach:

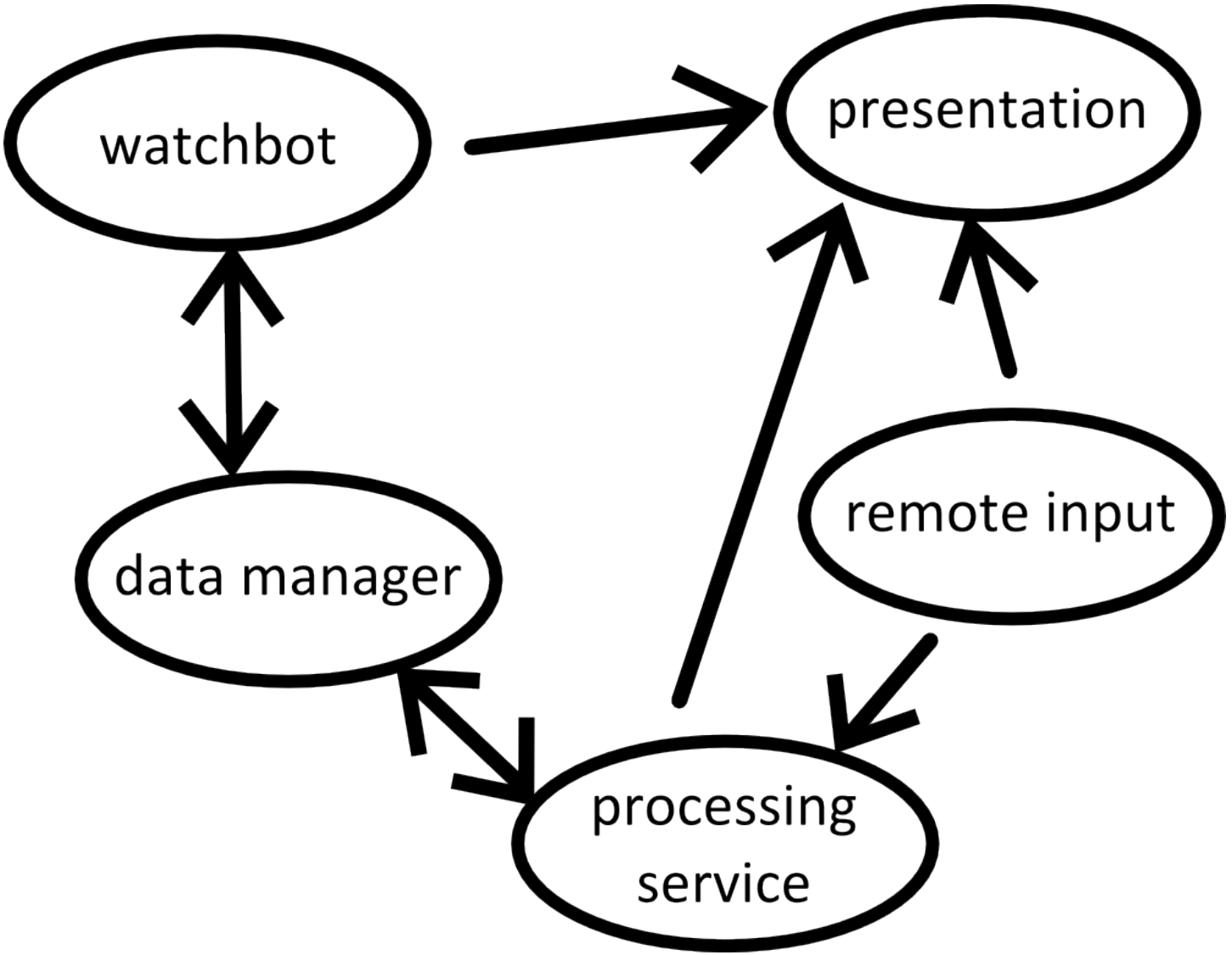
```
def divisorsRange(n: Int, begi: Int, end: Int) = {
  (Set[Int]() /: (begi to end)) {
    (set, i) => if (n % i == 0) set + i else set
  }
}
```

Przykład

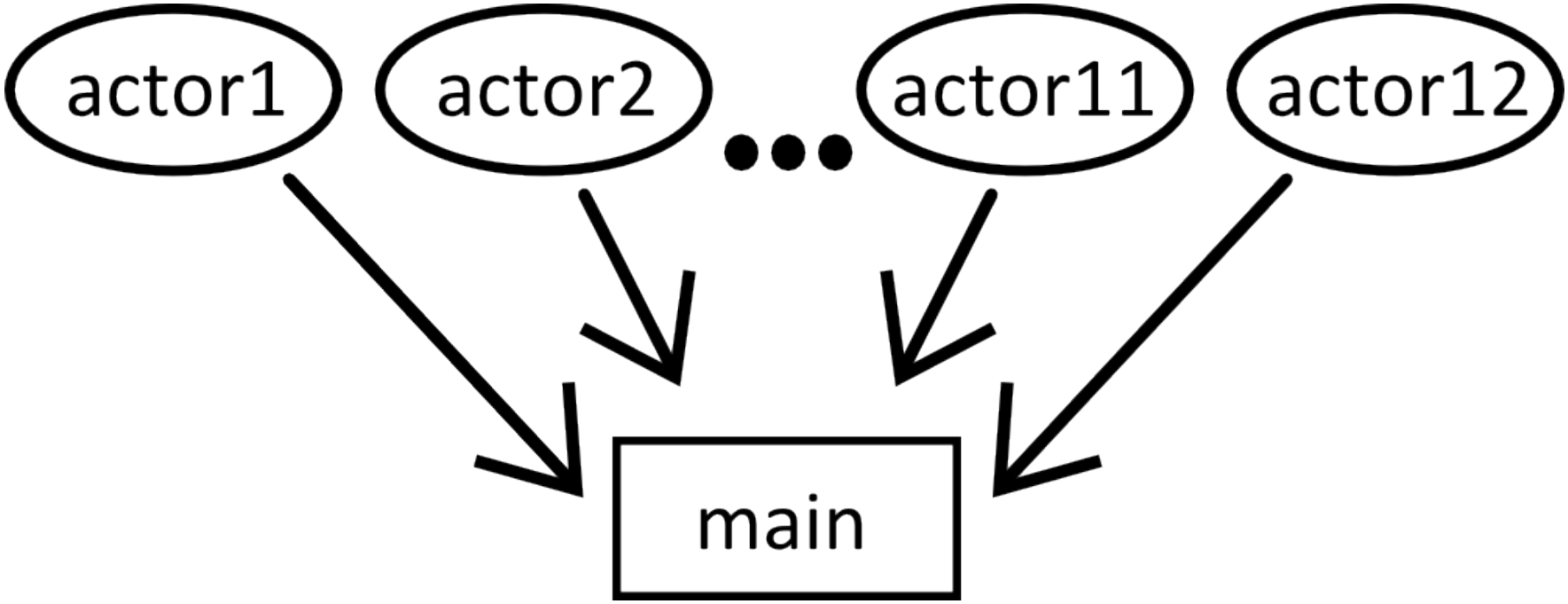
Odpalmy to na kilku ~~wątkach~~ aktorach:

```
def divisorsConcurrent(n:Int) = {
  val partitions = 12; //ilość partycji na jakie dzielimy
  val multi = n.toFloat/partitions //rozmiar przedziału
  val who = self //chcemy otrzymać wyniki

  //odpalamy aktorów szukających dzielników na danych im przedziałach
  //niech odpowiadają nam wyliczonymi przedziałami
  //PS. nie dzielcie tak na przedziały, to zło (tylko dlatego by się zmieścić)
  for (i <- 1 to partitions) {
    actor {
      who ! divisorsRange(n, (multi*(i-1) + 1).floor.toInt, (multi*i).ceil.toInt)
    }
  }

  //odbieramy wszystkie wiadomości od aktorów i sumujemy przedziały
  (Set[Int]() /: (1 to partitions)) {
    (set, i) => receive {
      case subset: scala.collection.immutable.Set[Int] => set ++ subset
    }
  }
}
```

Pamiętacie?


watchbot
presentation
data manager
remote input
processing
service

Prymitywnie



Akka

- API dla Javy i Scali,
- integracja z Spring/Guice,
- "Let it crash" / "Embrace failure",

Przykładowo:

Java

```
//ze strony Akki
public class SampleActor extends UntypedActor {
  public void onReceive(Object message) throws Exception {
    if (message instanceof String) println("actor");
    else throw new IllegalArgumentException("Unknown message: " + message);
  }
}

//zdecydowanie mniej wygodne
import static akka.actor.Actors.*;
ActorRef actor = actorOf(SampleActor.class);
myActor.start();

myActor.sendOneWay("Hello");
Future future = myActor.sendRequestReplyFuture("Hello", getContext(), 1000);
getContext().replyUnsafe(msg + " from " + getContext().getUuid());
myActor.sendOneWay(new Kill());
```

Jak integrować?

- Standardowy model:
  - używamy w Scali naszych Javowych klas,
  - piszemy warstwę abstrakcji w Scali, używamy z Javy.
- Akka:
  - aktorzy Ci sami w dwóch API – wszystko działa.

Java

```
public class AkkaExample extends akka.actor.UntypedActor {
    @Override
    public void onReceive(Object arg) throws Exception {
        if (arg instanceof String) {
            System.out.println("werks! " + (String)arg);
        }
    }
}
```

Scala

```
val aka = akka.actor.Actor.actorOf[AkkaExample]
aka.start
aka ! "yeah!"
```

Po co to?

Aktorzy pomagają zmagać się z :

- problemami z przezroczystością lokacji (aktorzy zdalni),
- kłopotliwą skalowalnością ,
- niespójnością systemów.

Praktyka:

- stare-nowe podejście ,
- elastyczność,
- sporo implementacji (C++/Javy/C#/...),
- od razu w Erlangu, Scali i Io,
- Erlang → używane przez Ericssona w systemach o dużej współbieżności,
- sprawdza się (kilka słów, więcej w linkach),
- wydajność wątków (~) przy mniejszej ilości problemów,
- aktorzy → wieloagentowe systemy .

Pytania?

Źródła

- http://www.scala-lang.org/docu/files/collections-api/
- „Programming Scala” Venkat Subramaniam (Pragmatric Bookshelf)
- http://akka.io/ - aktorzy dla Javy
- http://stackoverflow.com/questions/4493001/good-use-case-for-akka

  - przykłady użycia
- http://osl.cs.uiuc.edu/parley/ - Python
- http://theron.ashtonmason.net/ - C++