Akka streams

Потоки AKKA
Reactive manifesto

Reactive manifesto

- http://www.reactivemanifesto.org/
- Опубликован в 2014 году.
- Указывает на накопившиеся проблемы приложений использующих традиционные подходы в разработке
  - Данные исчисляются петабайтами
  - Один и тот же сервис должен быть доступен не только из заданного приложения но и из браузера и мобильных устройств
  - Пользователи ожидают время реакции – доли секунды.
  - Широкое распостранение многоядерных и много серверных “железных” архитектур

Идеи Reactive manifesto

- Приложения должны быстро отвечать на запросы (responsive)
  - Одна из ключевых характеристик приложения
  - Требует постоянного мониторинга и учета этого требования в момент разработки приложения
- Устойчивость к сбоям (resilient).
  - Репликация – параллельное выполнение входящих запросов независимо друг от друга
  - Использование изолированых контейнеров
  - Восстановление “упавших” контейнеров должно быть поручено другим контейнерам (желательна иерархическая структура)
  - Слабая связность

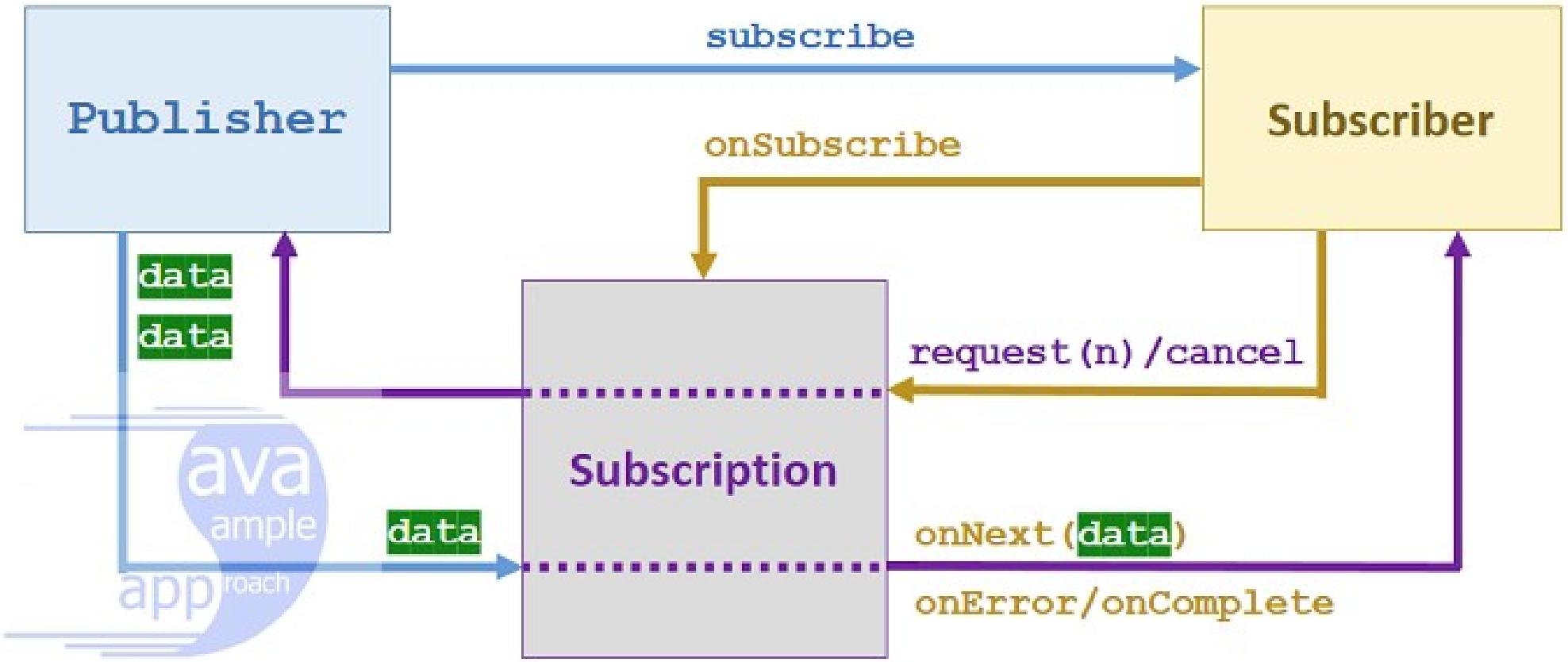
Идеи Reactive manifesto

- Эластичность (Elastic)
  - Приложение должно уметь динамически использовать дополнительные ресурсы в случае увеличения нагрузки
  - Не должно быть “узких мест”
  - Требуется возможность шардирования ключевых компонент и перенастройки среды выполнения на лету
- Взаимодействие через асинхронную передачу сообщений
  - Слабая связность
  - Изоляция
  - Прозрачность местонахождения компонентов
  - Передача сбоев как сообщений
  - Backpressure

Reactive streams

- Инициатива которая предоставила простой API который позволяет организовать поток данных с возможностью back pressure
- Стояла задача минимизации и стандартизации API
- В JDK9 появился java.util.concurrent.Flow – полный аналог

На что это похоже



Reactive streams API

- org.reactivestreams.Publisher
  - void subscribe(Subscriber<? super T> s)

    метод для подписки на события
- org.reactivestreams.Subscriber
  - void onComplete()

    Сигнализирует о успешном финальном состоянии
  - void onError(java.lang.Throwable t)

    Сигнализирует об аварийном финальном состоянии
  - void onNext(T t)

    Вызывается Publisher-ом в ответ на запрос события Subscription.request(long)
  - void onSubscribe(Subscription s)

    Вызывается после Publisher.subscribe(Subscriber).

Reactive streams API

- org.reactivestreams.Subscription
  - void cancel()

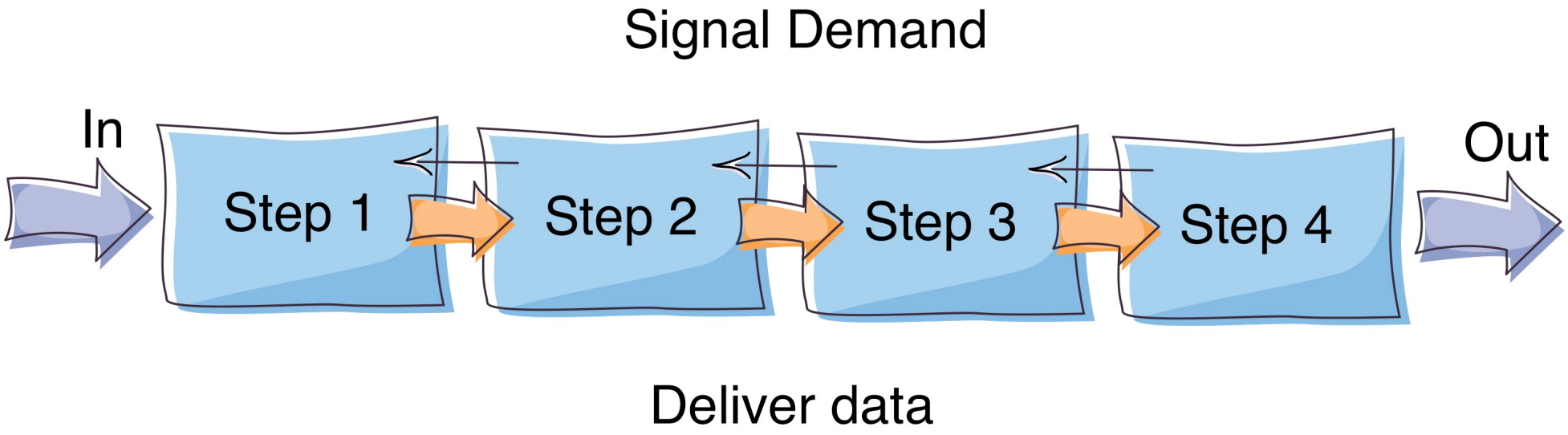
    Запрос к Publisher-у на отмену subscription
  - void request(long n)

    Запрос к Publisher-у на получение следующего сообщения
- org.reactivestreams.Processor

  Является одновременно Publusher и Subscriber

На что похоже Akka streams

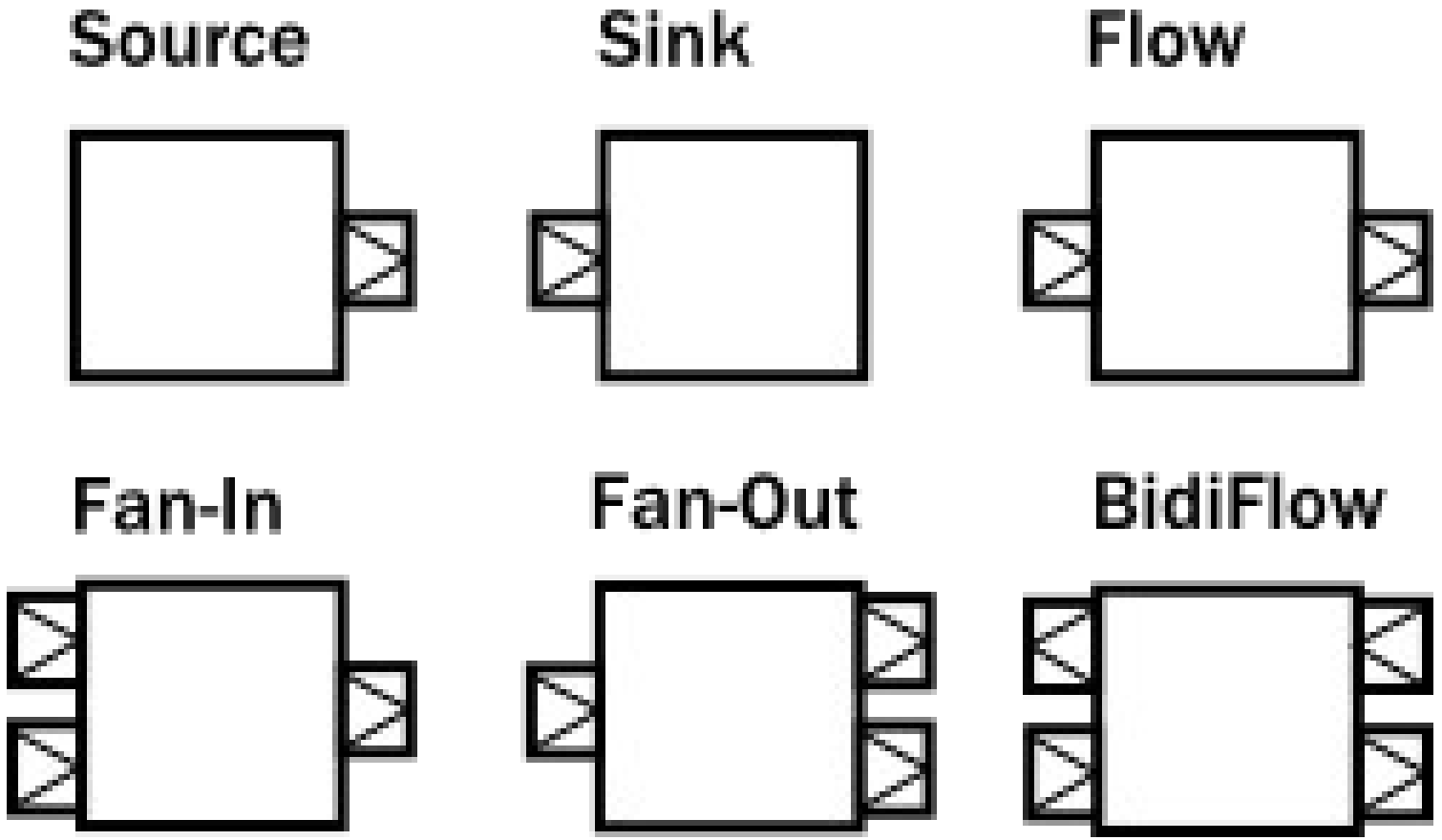
- Akka streams предназначен для запуска потока обработки данных



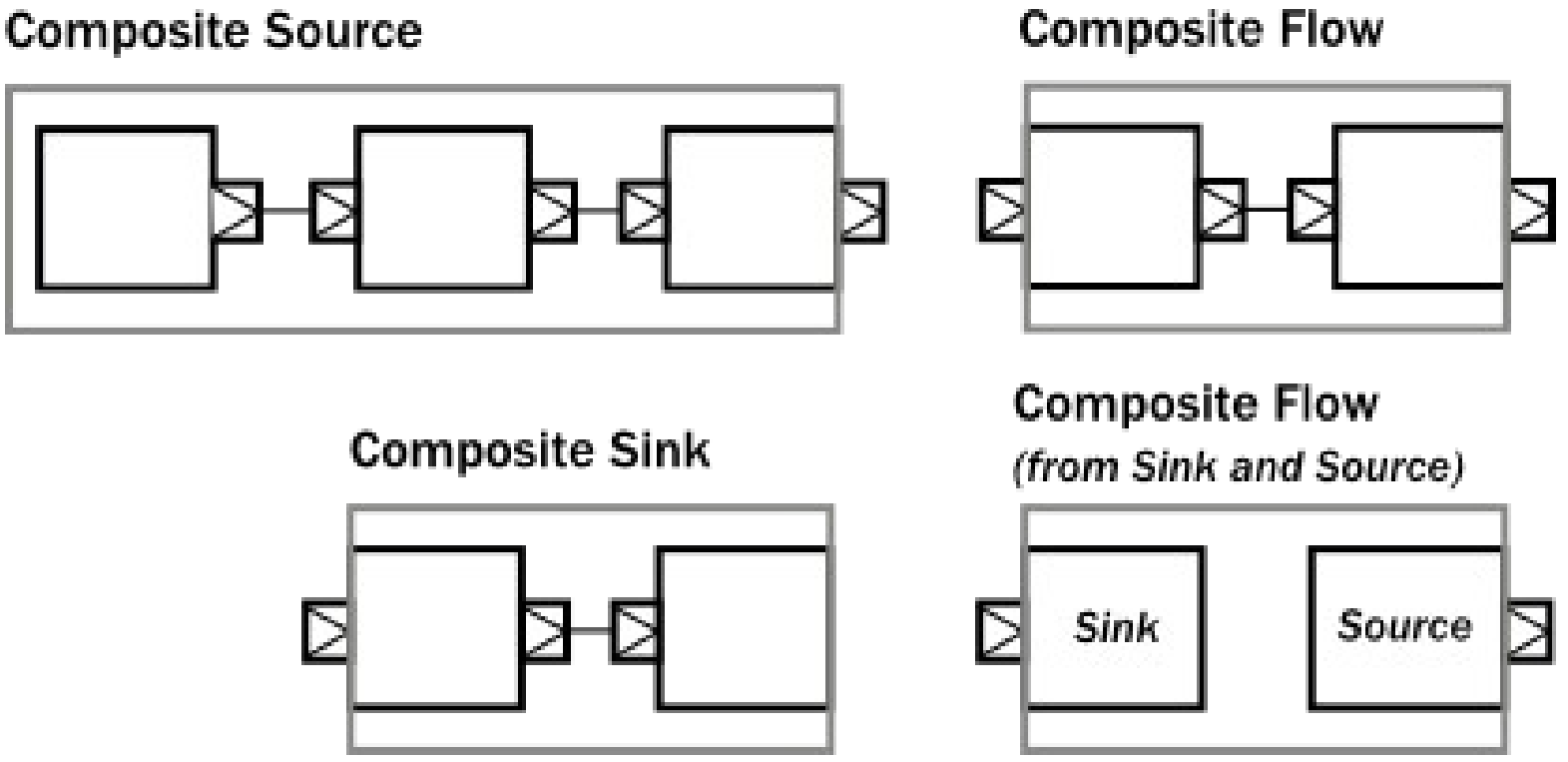
Компоненты Akka streams

- Source
  - Узел потока обработки данных с одним выходом
  - Работает как источник данных потока и генерирует данные
- Sink
  - Узел потока с одним входом
  - Замыкает поток данных и служит его последним узлом – потребляет данные
- Flow
  - Узел потока с одним входом и выходом
  - Предназначен для обработки и преобразования проходящих через него данных

Компоненты



Составные компоненты



Термины

- RunnableGraph – завершенное описание потока данных akka streams
  - состоит из источника (source), этапов обработки и завершающего узла sink
  - автоматически не запускается – требуется материализация
- MaterializedMap – запущенный в окружении AKKA RunnableGraph
- Back pressure – процесс автоматической балансировки нагрузки. Источник данных “подстраивается” под потребителя и не генерирует данных больше чем тот может обработать

Минимальный пример

```
ActorSystem system = ActorSystem.create("simplest-test");
ActorMaterializer materializer = ActorMaterializer.create(system);

Source<Integer, NotUsed> source = Source.from(Arrays.asList(1, 2, 3, 4, 5));
Flow<Integer, Integer, NotUsed> increment = Flow.of(Integer.class).map(x -> x + 1);
Sink<Integer, CompletionStage<Integer>> fold = Sink.fold(0, (agg, next) -> agg + next);

RunnableGraph<CompletionStage<Integer>> runnableGraph =
    source.via(increment).toMat(fold, Keep.right());
CompletionStage<Integer> result = runnableGraph.run(materializer);

result.thenAccept(i -> System.out.println("result=" + i))
        .thenAccept((v) -> system.terminate());
```

Пример композиции

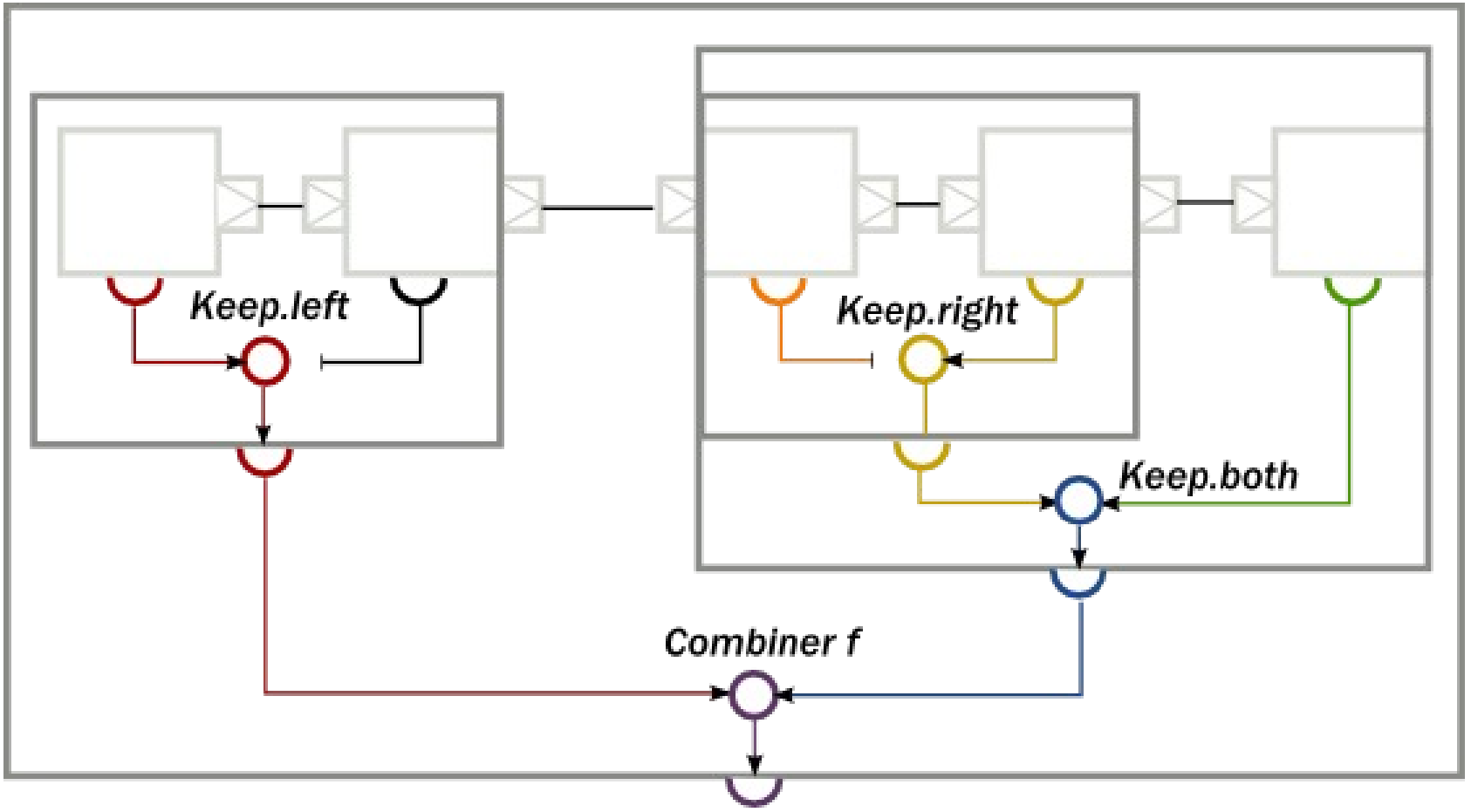
```
Source<Integer, Cancellable> source = Source.tick(
    FiniteDuration.create(0, TimeUnit.SECONDS),
    FiniteDuration.create(100, TimeUnit.MILLISECONDS),
    1);
Flow<Integer, Integer, NotUsed> increment = Flow.of(Integer.class).map(x -> x + 1);
Source<Integer, Cancellable> incremented = source.viaMat(increment, Keep.left());

Flow<Integer, Integer, NotUsed> take10 = Flow.of(Integer.class).take(1000);
Sink<Integer, CompletionStage<Integer>> fold = Sink.fold(0, (agg, next) -> agg +
next);
Sink<Integer, CompletionStage<Integer>> sink = take10.toMat(fold, Keep.right());
```

Materialized value

- После запуска потока мы теряем возможность с ним взаимодействовать
- Для того чтобы сохранить эту возможность – source, sink, flow кроме основного канала взимодействия имеют дополнительный, тип которого задается вторым параметром generic
- В предыдущем примере – у source нет дополнительного materized value а у sink есть. В этом значении мы получаем future с финальным результатом операции fold

Сочетание materialized value



Пример взаимодействия с Materialized value

```
ActorSystem system = ActorSystem.create("simplest-test");
ActorMaterializer materializer = ActorMaterializer.create(system);
Source<Integer, Cancellable> source = Source.tick(
    FiniteDuration.create(0, TimeUnit.SECONDS),
    FiniteDuration.create(100, TimeUnit.MILLISECONDS), 1);
Source<Integer, Cancellable> incremented = source.map(x -> x + 1);
Sink<Integer, CompletionStage<Integer>> fold = Sink.fold(0, (agg, next) -> agg + next);
RunnableGraph<Pair<Cancellable, CompletionStage<Integer>>> graph =
    incremented.toMat(fold, Keep.both());// 0
Pair<Cancellable, CompletionStage<Integer>> run = graph.run(materializer);
Thread.sleep(2000);
run.second().thenAccept(i -> System.out.println("result=" + i));
run.first().cancel();
system.terminate();
```

Обработка ошибок

- В случае возникновения ошибки по умолчаню поток (стрим) останавливается и exception можно получить через materialized value потока
  - run.second()
    .exceptionally(err -> {
      err.printStackTrace();
      return -1l;
    })
    .thenAccept(i -> System.out.println("result=" + i));
- В случае необходимости продолжить обработку для шага обработки или всего графа целиком можно задать функцию Decider которая будет на основании ошибки решать что делать с потоком :
  - Resume, пропустить одно значение в потоке и продолжить дальше
  - Restart, перезапустить поток. Все накопленные значения в шагах агрегатороах очищаются
  - Stop, остановить выполнение потока

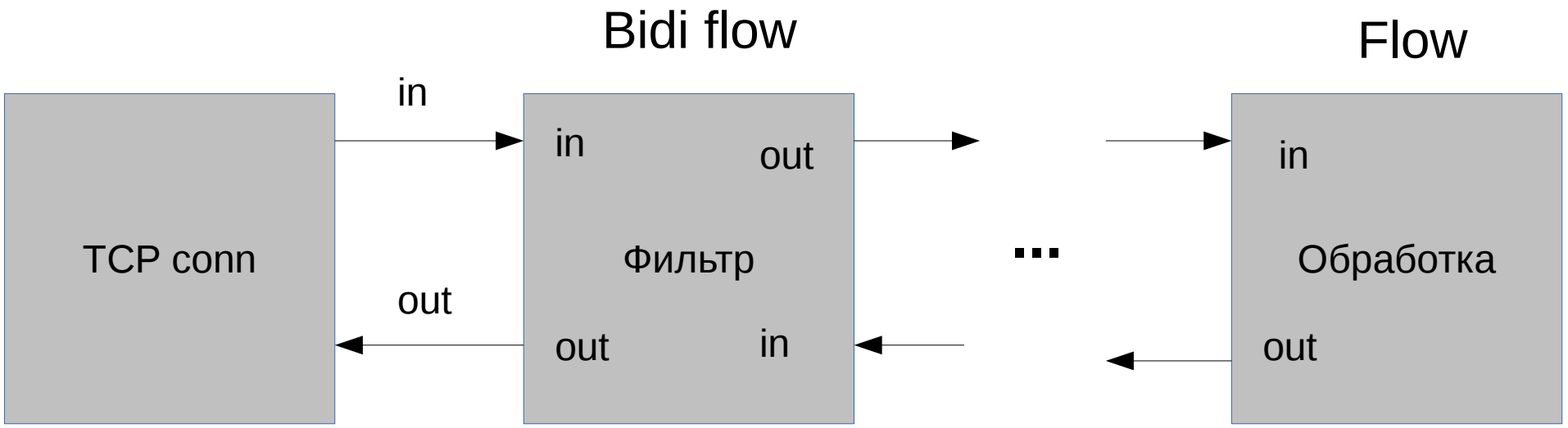
Пример работы с ошибками

```
final Function<Throwable, Supervision.Directive> decider = exc -> {
    if (exc instanceof ArithmeticException)
        return Supervision.resume();
    else
        return Supervision.stop();
};
ActorSystem system = ActorSystem.create("simplest-test");
ActorMaterializer materializer = ActorMaterializer.create(system);
Source<Integer, Cancellable> source = Source.tick(FiniteDuration.create(0,
TimeUnit.SECONDS), FiniteDuration.create(100, TimeUnit.MILLISECONDS), 1);
Source<Pair<Integer, Object>, Cancellable> sourceWithIndex = source.zipWithIndex();
Source<Long, Cancellable> incremented = sourceWithIndex.map(x -> {
        Long index = (Long) x.second();
        //здесь упадет ошибка на 5-м индексе
        Long a = 10 / (index - 5);
        return index + 1;
    }
).withAttributes(ActorAttributes.withSupervisionStrategy(decider));
```

Двунаправленный поток данных

- BidiFlow – является сочетанием двух flow направленных в разные стороны.
- Является типовым патерном в протоколах запрос-ответ.
- Основная функциональность – трансформация запроса проходящего в одну сторону и трансформация ответа который идет обратно.
- С помощью стандартного API мы можем выстраивать цепочки BidiFlow .

BidiFlow


Bidi flow
Flow
in
TCP conn
in
out
Фильтр
out
in
out
...
in
Обработка
out

Пример BidiFlow

```
private static <IN, OUT> BidiFlow<HttpRequest, IN, OUT, HttpResponse, NotUsed>
createTransformer(
        Class<IN> clazzIn,
        Class<OUT> clazzOut
    ) {
        Flow<HttpRequest, IN, NotUsed> inFlow = Flow.of(HttpRequest.class)
                .flatMapConcat(r -> r.entity().getDataBytes())
                .map(entity -> new ObjectMapper().readValue(entity.utf8String(), clazzIn));
        Flow<OUT, HttpResponse, NotUsed> outFlow = Flow.of(clazzOut).map(
                resp -> HttpResponse.create()
                        .withEntity(ByteString.fromString(new ObjectMapper().writeValueAsString(resp)))
        );
        return BidiFlow.fromFlows(inFlow, outFlow);
    }
```

Превращение BidiFlow в обычный Flow :

```
Flow<Ping, Pong, NotUsed> requestProcessor = return createTransformer(Ping.class, Pong.class)
                                                        .join(requestProcessor);
```