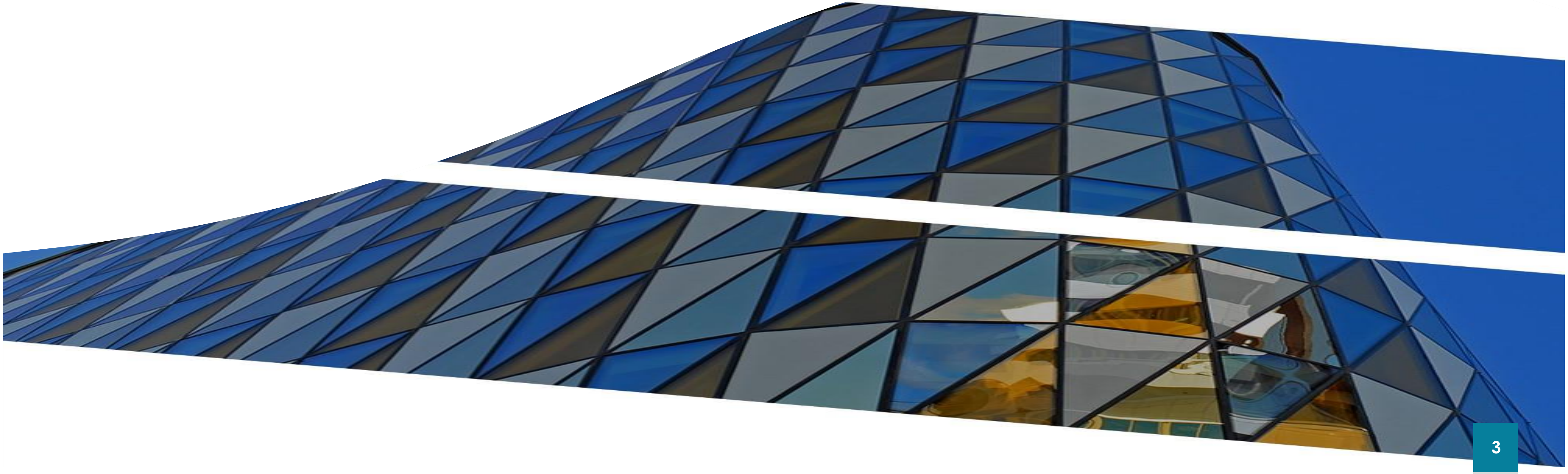
Java Stream API

Where to Find The Code and Materials?

https://github.com/iproduct/course-stream-api-2022

Java 8 Stream API

Practical Exercises – Functional Programming Koans



Agenda for This Session

- Fundamentals
- Functional interfaces
- Method references
- Constructor references

Новости в Java™ 8+

- Ламбда изрази и поточно програмиране – пакети **java.util.function** и **java.util.stream**)
- Референции към методи
- Методи по подразбиране и статични методи в интерфейси – множествено наследяване на поведение в Java 8
- Функционално програмиране в Java 8 с използване на монади (напр. Optional, Stream) – предимства, начин на реализация, основни езикови идиоми, примери

Функционални интерфейси в Java™ 8

- Функционален интерфейс = интерфейс с един абстрактен метод SAM (Single Abstract Method) – @FunctionalInterface
- Примери за функционални интерфейси в Java 8:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
public interface Runnable {
    public void run();
}
public interface Callable<V> {
    V call() throws Exception;
}
```

Ламбда изрази – пакет java.util.function

Примери:

```
(int x, int y) -> x + y

() -> 42

(a, b) -> a * a + b * b;

(String s) -> { System.out.println(s); }

book -> book.getAuthor().fullName()

voter -> voter.getAge() >= legalAgeOfVoting

(person1, person2) -> person1.getAge() - person2.getAge()

(song1, song2) -> song1.getArtist().compareTo(song2.getArtist())
```

Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial)Автор: Graham Allan, Лиценз: GNU GENERAL PUBLIC LICENSE v2

Правила за форматирне на ламбда изрази

- **Ламбда изразите (функциите)** могат да имат произволен брой **параметри**, които се ограждат в скоби, разделят се със запетаи и могат да имат или не деклариран тип (ако нямат - типът им се извежда от **контекста на използване = target typing**). Ако са само с един параметър, то скобите не са задължителни.

- **Тялото на ламбда изразите** се състои от произволен езикови конструкции (statements), разделени с **;** и заградени във фигурни скоби. Ако имаме само една езикова конструкция – израз то използването на фигурни скоби не е необходимо – в този случай стойността на израза автоматично се връща като стойност на функцията.

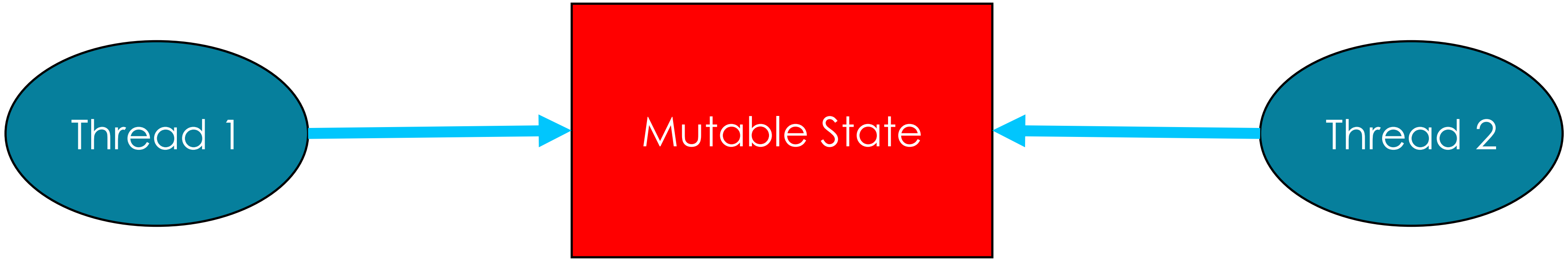
Пакет java.util.function

- **Predicate<T>** – предикат = булев израз представящ свойство на обекта подавано като аргумент
- **Function<A,R>**: функция която приема като аргумент **A** и го трансформира в резултат **R** (метод **apply()**)
- **Supplier<T>** – с помощта на **get()** метод всеки път връща инстанция (обект) – фабрика за обекти
- **Consumer<T>** – приема аргумент (метод **accept()**) и изпълнява действие върху него
- **UnaryOperator<T>** – оператор с един аргумент **T -> T**
- **BinaryOperator<T>** – бинарен оператор **(T, T) -> T**

Data Streams Programming

Problem with OOP: Mutable State

- The object methods are supposed to mutate the object's internal state
- When there is state sharing:



- Bottlenecks (Contention), Deadlocks, Complexity in State Access Management (mutual exclusion between threads)

OOP vs. Functional Composition

- **OOP** – imperative, hard to chieve concurrency, less-reusable abstractions (how many times you have created User class in your career?)
- **FP** – declarative, always safe concurrency (pure functions), coarse grained abstractions, code reuse via functional composition, Composable abstractions: Stream, Optional, etc.

Functional Programming

FP is a type of programming paradigm which has several features:

- Purity:
  - Function reads all inputs from its input arguments.
  - Function exports all outputs to its return values.
  - The function always evaluates the same result value given the same argument value(s).
  - Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.
- Immutability – state of objects cannot be modified after it is created, but wait, how can we program without modifying state ?
- First-Class & High-Order Function
- TCO, Closure, Curry…

First Class Functions

Capability of programming language to:

- pass functions as arguments to other functions
- return functions as the values from other functions
- assign functions to variables
- store functions in data structures

To be concise, function is just like all other values like integer, float, double, etc..

Higher Order Functions

Function that does at least one of the following:

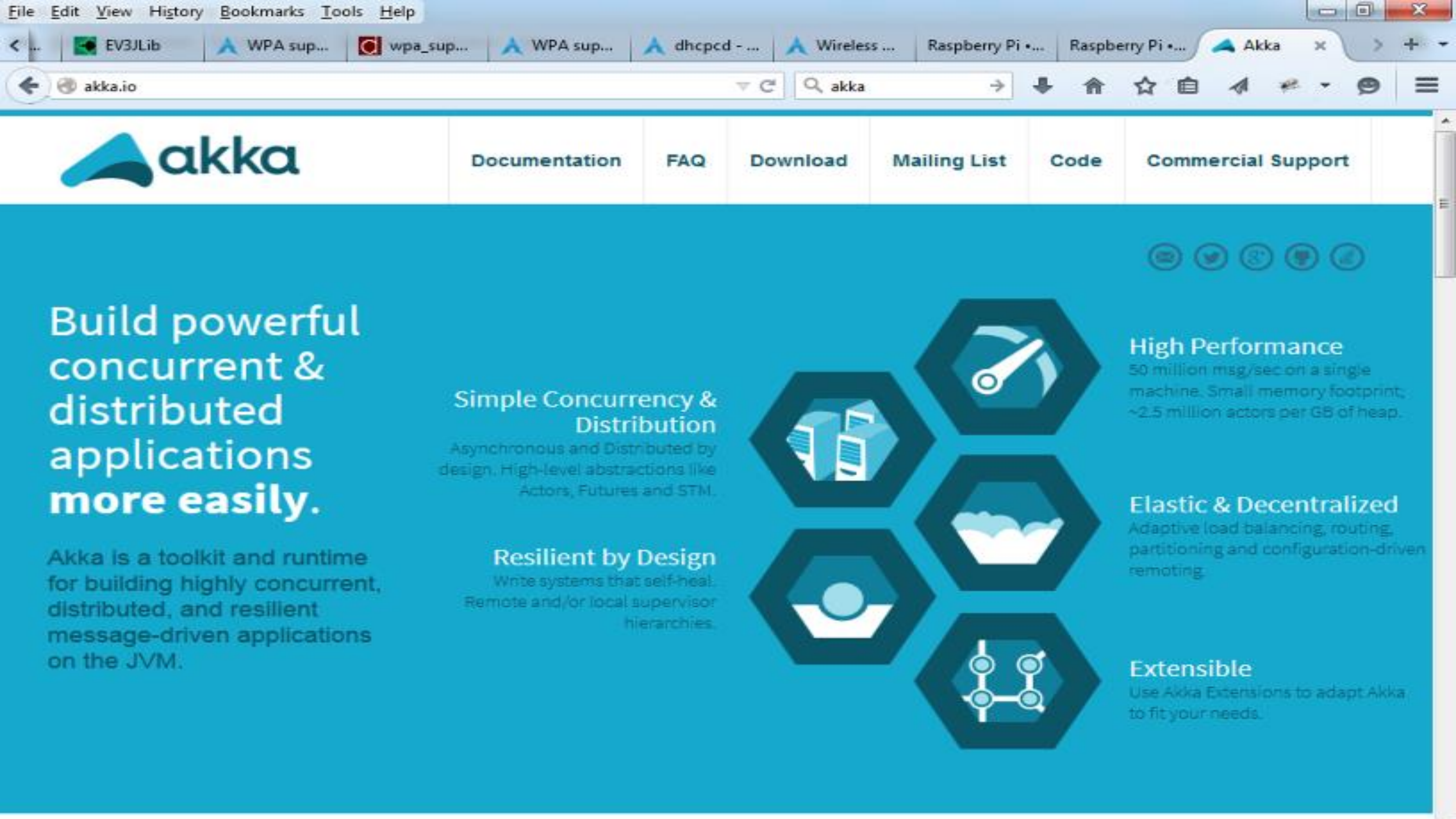
- takes one or more functions as arguments
- returns a function as its result
- Examples:

```
var lines = Files.lines(path).map(line -> line.toUpperCase());

var numbers = IntStream.iterate(1, x -> x + 1).boxed();

var results = zip(numbers, lines, (Integer n, String line) -> n + ": " + line);

results.forEach(System.out::println);
```

What can FP offer to distributed computing ?

- No side-effects and immutable – variables FP facilitates code distribution over several CPU and eases concurrent programming
- Functions are better building components than objects:
  - Functions can be combined, sent remotely
  - Functions can be applied locally on distributed data sets (e.g. parallel stream, using Fork-Join pool underneath).
- In order to do the splitting of the work between multiple threads (forking) the Java Streams use:

  spliterator = split iterator
- The results can be joined after that in a single result (e.g. reduce)
- Example: Map – Reduce big data architecture (Google, Hadoop)



Build powerful concurrent & distributed applications **more easily.**

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.

Simple Concurrency & Distribution

Asynchronous and Distributed by design. High-level abstractions like Actors, Futures and STM.

Resilient by Design

Write systems that self-heal. Remote and/or local supervisor hierarchies.

High Performance

50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.

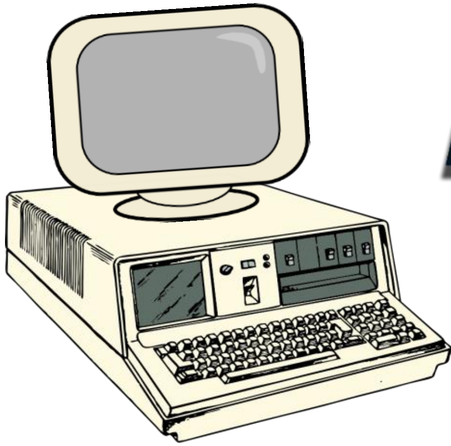
Elastic & Decentralized

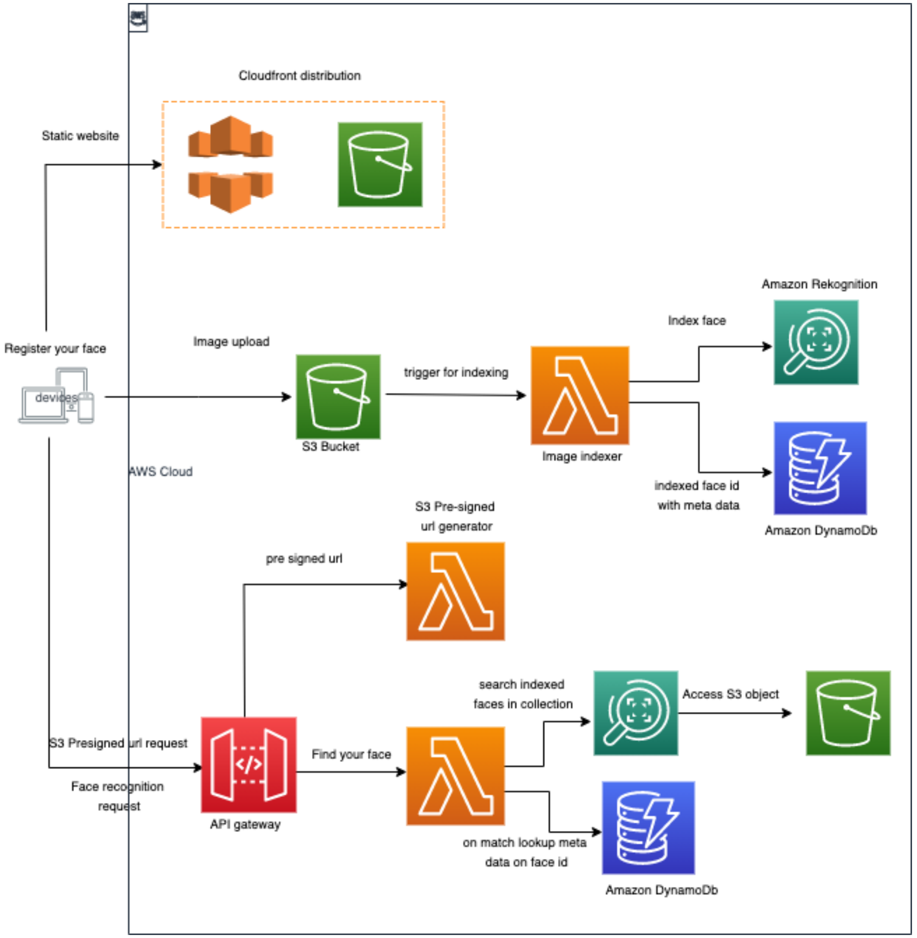
Adaptive load balancing, routing, partitioning and configuration-driven remoting.

Extensible

Use Akka Extensions to adapt Akka to fit your needs.

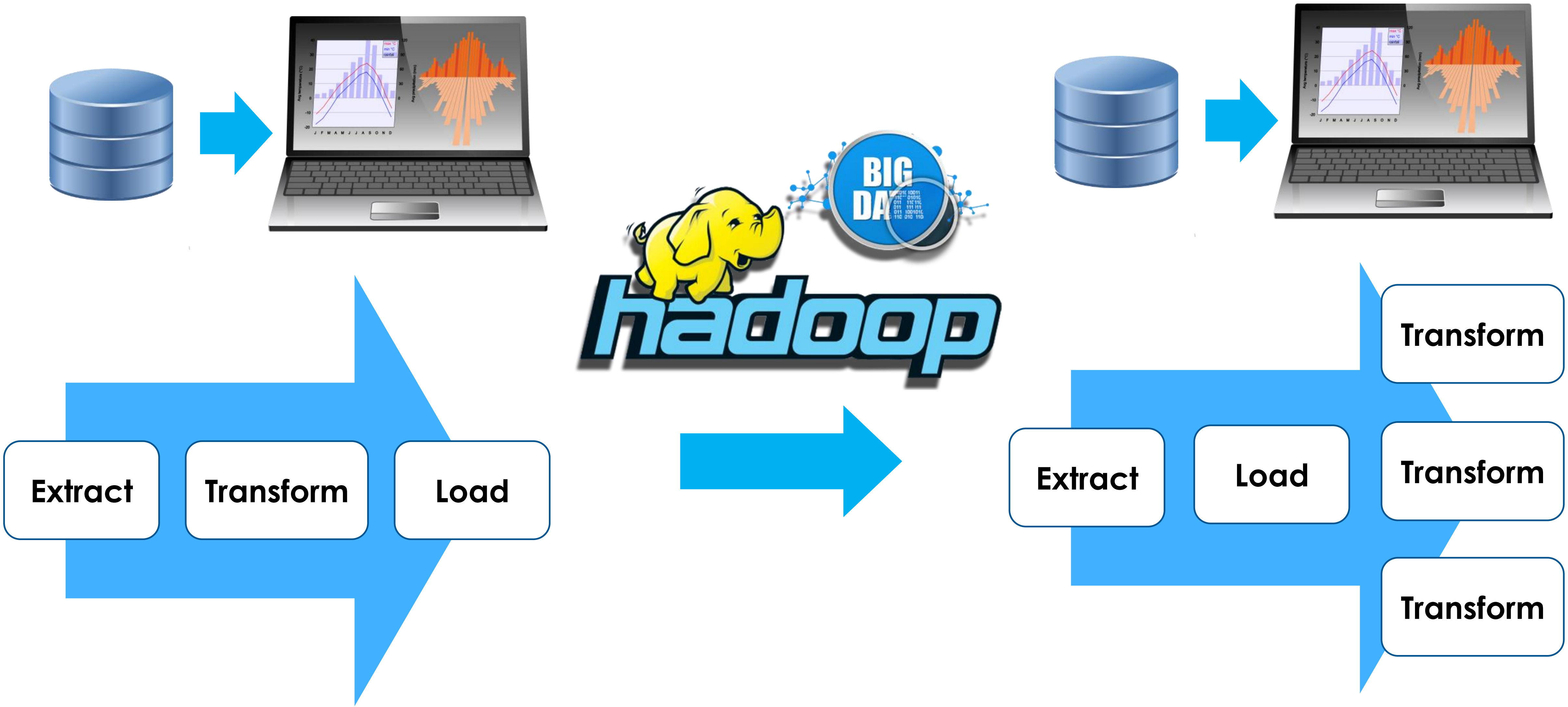
Need for Speed (and Scalability:)





Source: https://aws.amazon.com/blogs/opensource/simplifying-serverless-best-practices-with-aws-lambda-powertools-java/

Batch Processing



Extract

Transform

Load

Extract

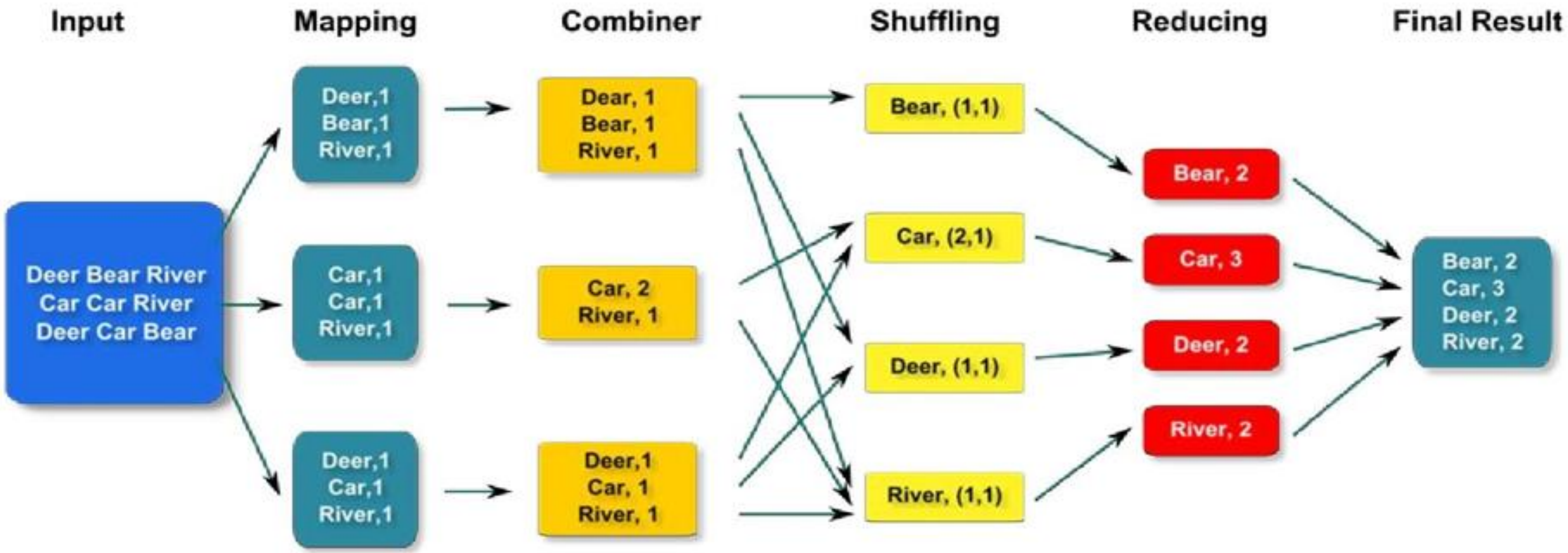
Load

Transform

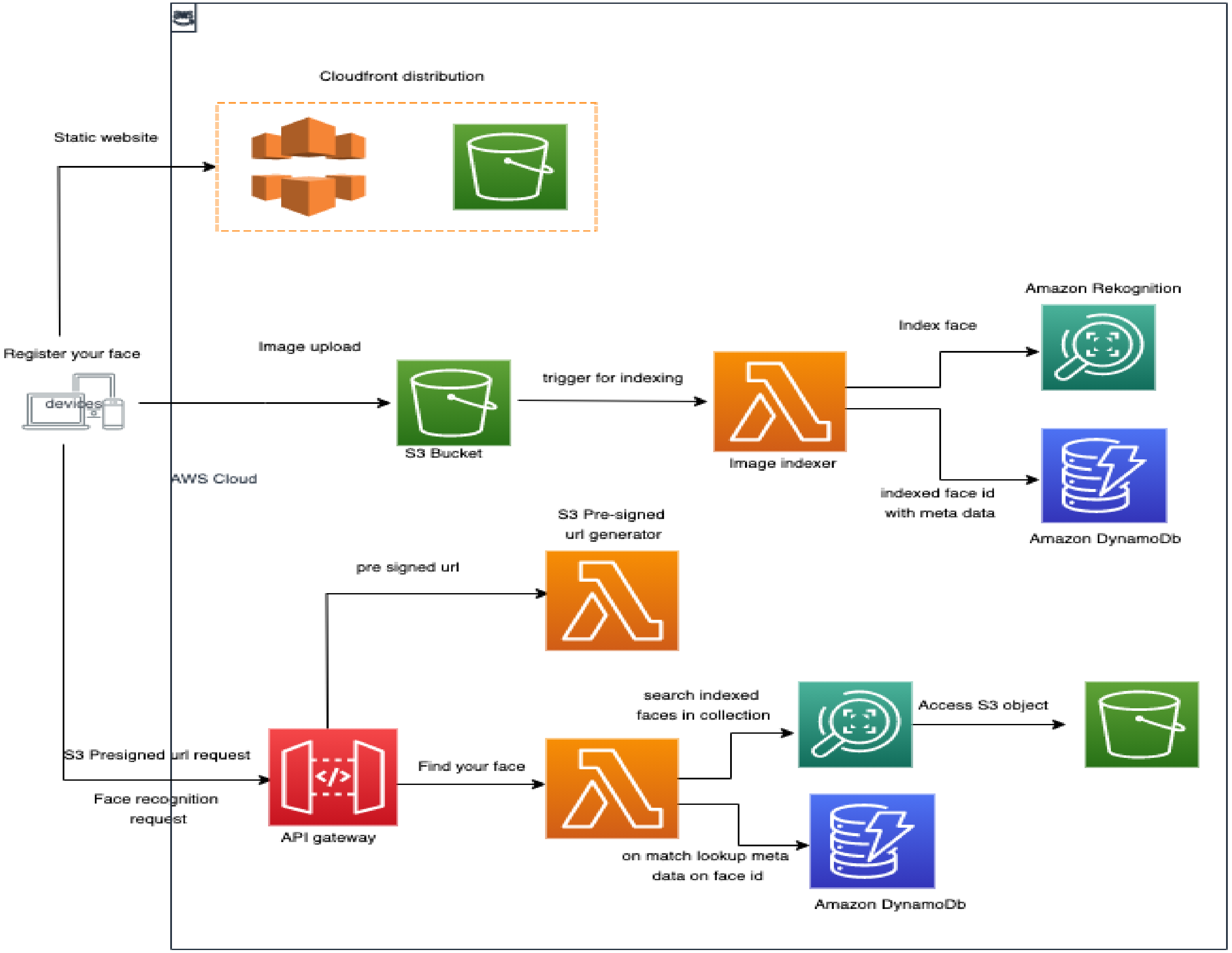
Transform

Transform

Map-Reduce Architecture

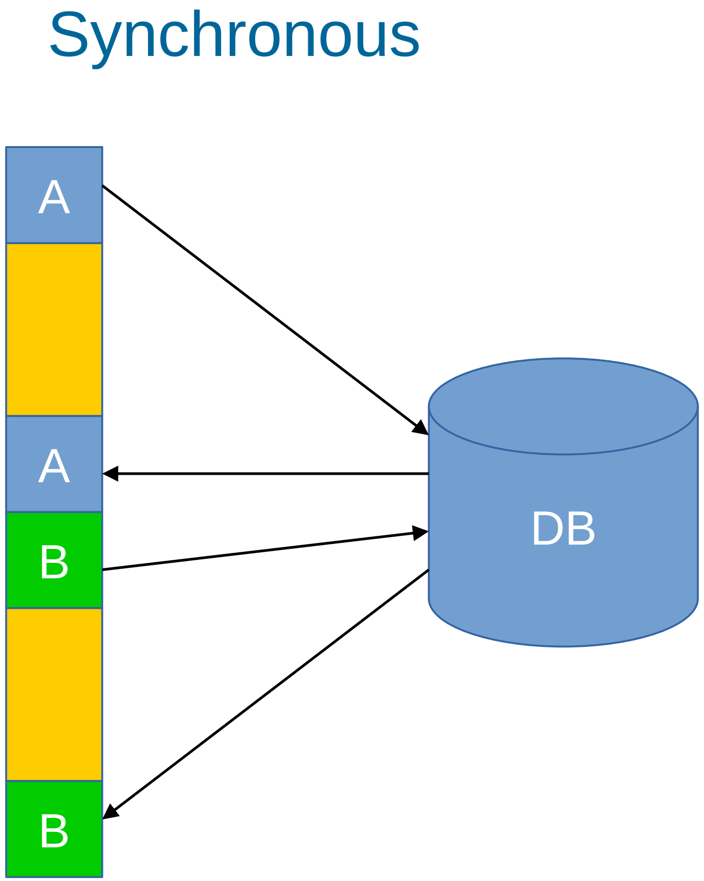


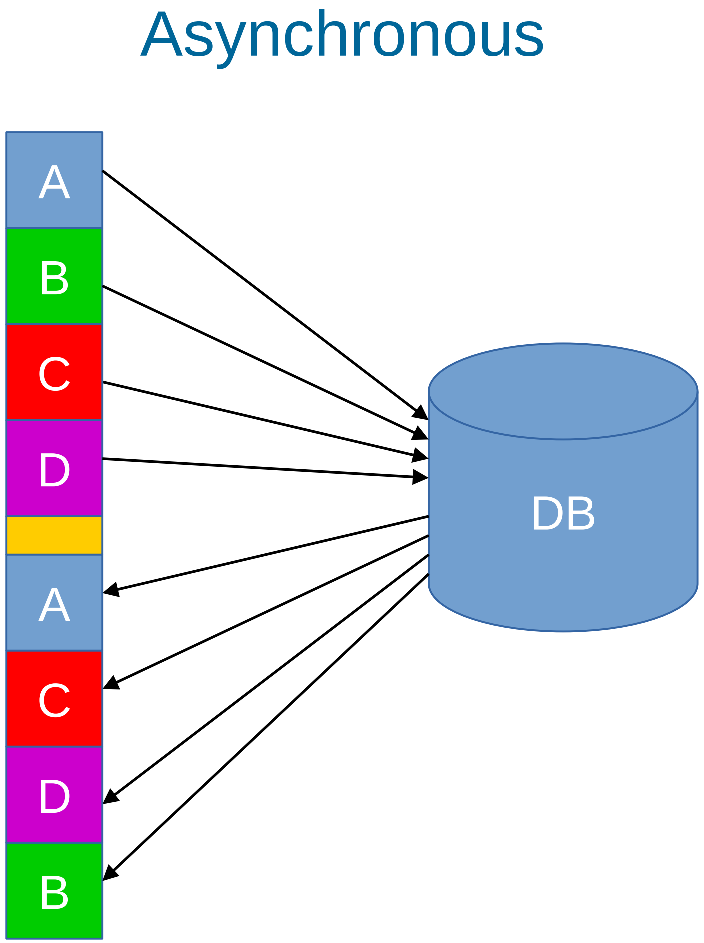
Amazon Lambda



Source: https://aws.amazon.com/blogs/opensource/simplifying-serverless-best-practices-with-aws-lambda-powertools-java/

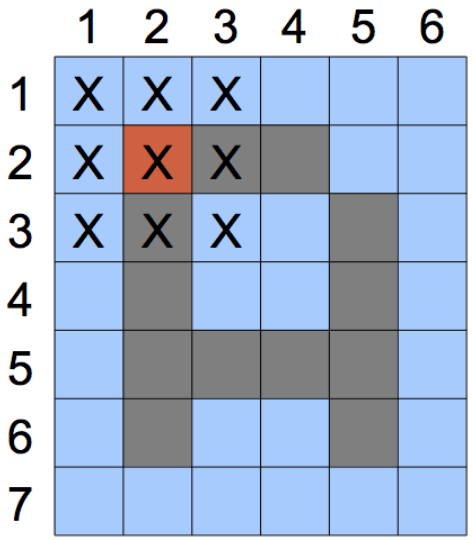
Synchronous vs. Asynchronous IO



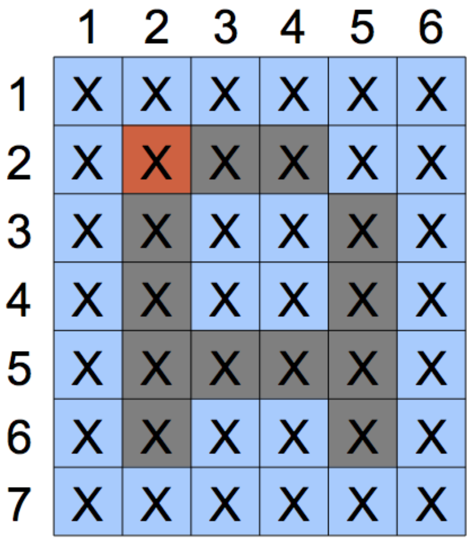


Difference between Local vs. Non-local, and Iterative vs. Non-iterative Processing

https://core.ac.uk/download/pdf/16270417.pdf



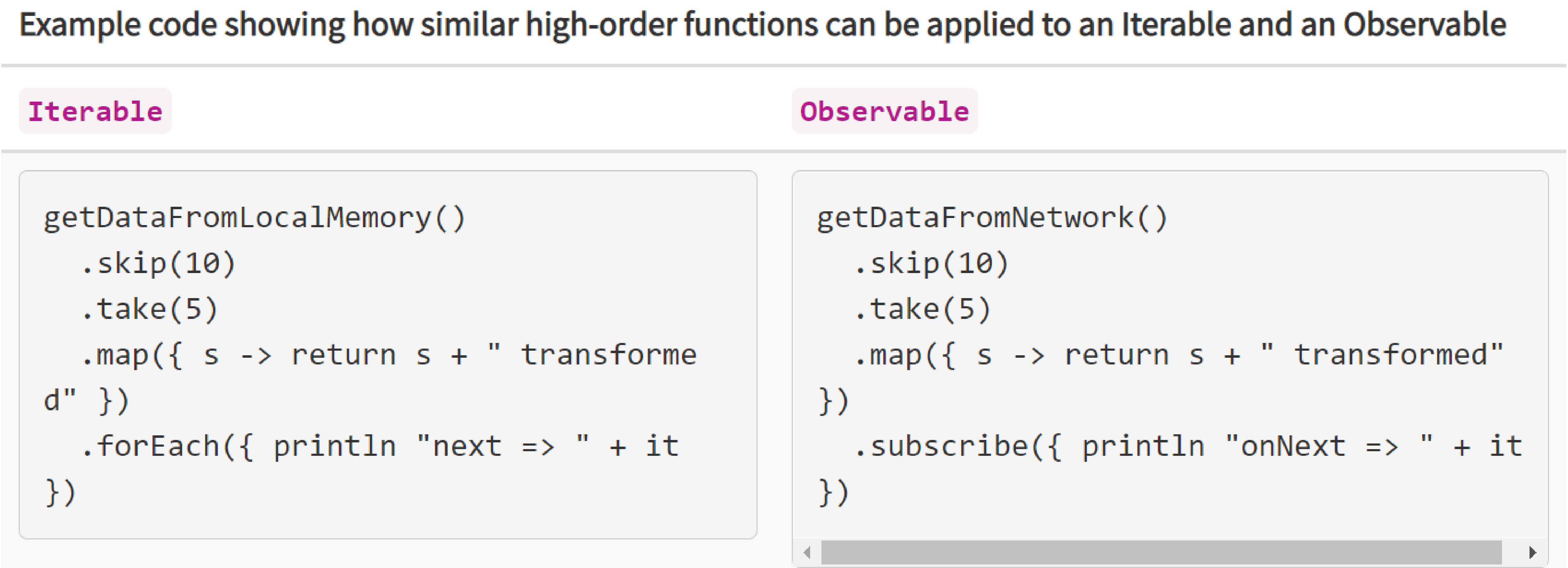
	1	2	3	4	5	6
1	X	X	X			
2	X	X	X			
3	X	X	X			
4						
5						
6						
7						



	1	2	3	4	5	6
1	X	X	X	X	X	X
2	X	X	X	X	X	X
3	X	X	X	X	X	X
4	X	X	X	X	X	X
5	X	X	X	X	X	X
6	X	X	X	X	X	X
7	X	X	X	X	X	X

The distinction between iterative and non-iterative algorithms is simpler: a non-iterative algorithm only processes the image a small, constant number of times to achieve the desired effect, whereas an iterative algorithm requires multiple passes, and often the output image of a previous pass becomes the input of the next pass.

ReactiveX: Observable vs. Iterable



Example code showing how similar high-order functions can be applied to an Iterable and an Observable

```
getDataFromLocalMemory()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transforme
d" })
  .forEach({ println "next => " + it
})
```

```
getDataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed"
})
  .subscribe({ println "onNext => " + it
})
```

You can think of the Observable class as a **"push"** equivalent to Iterable, which is a **"pull."** With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast, with an Observable the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously.

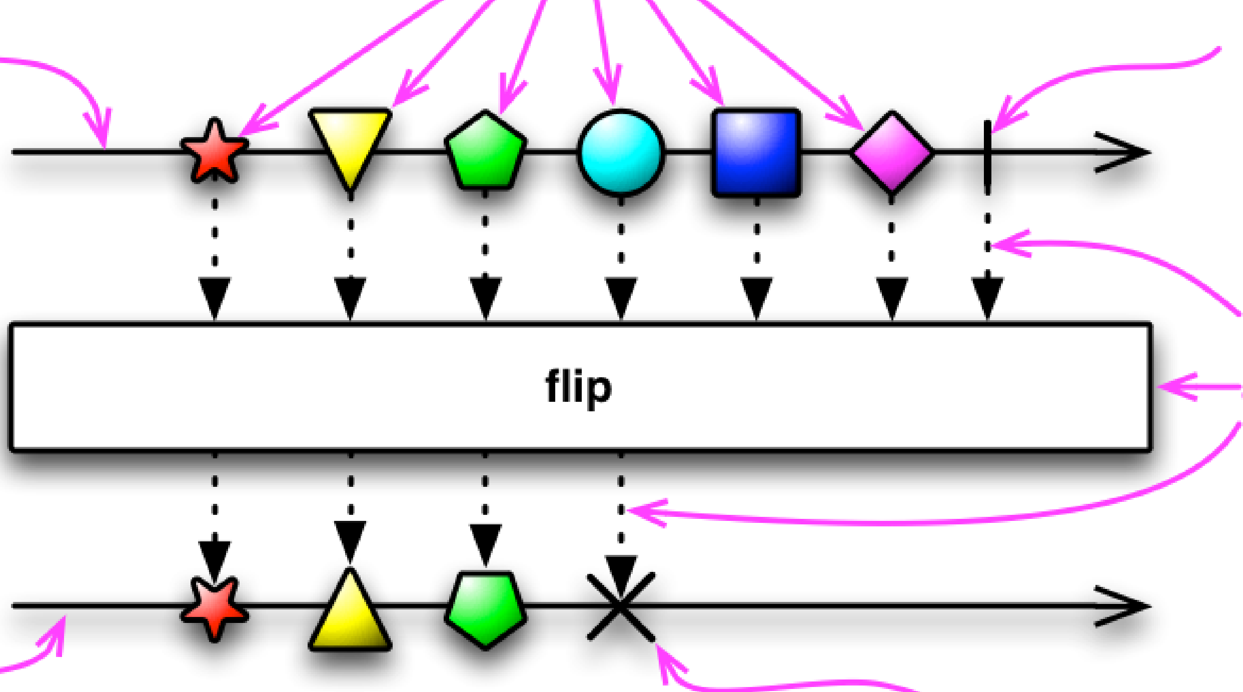
By ReactiveX - Creative Commons Attribution 3.0 License, Source: http://reactivex.io/intro.html

ReactiveX Observable – Marble Diagrams

This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.



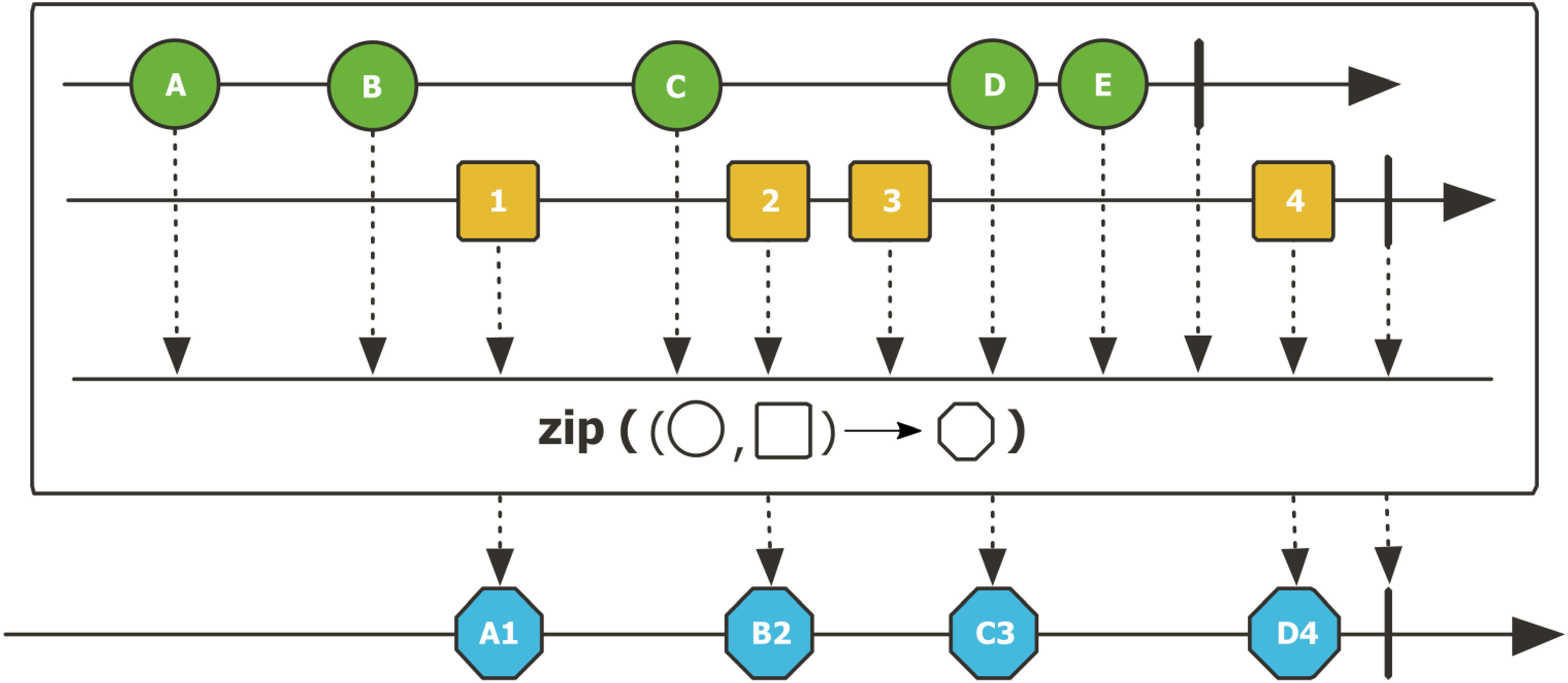
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

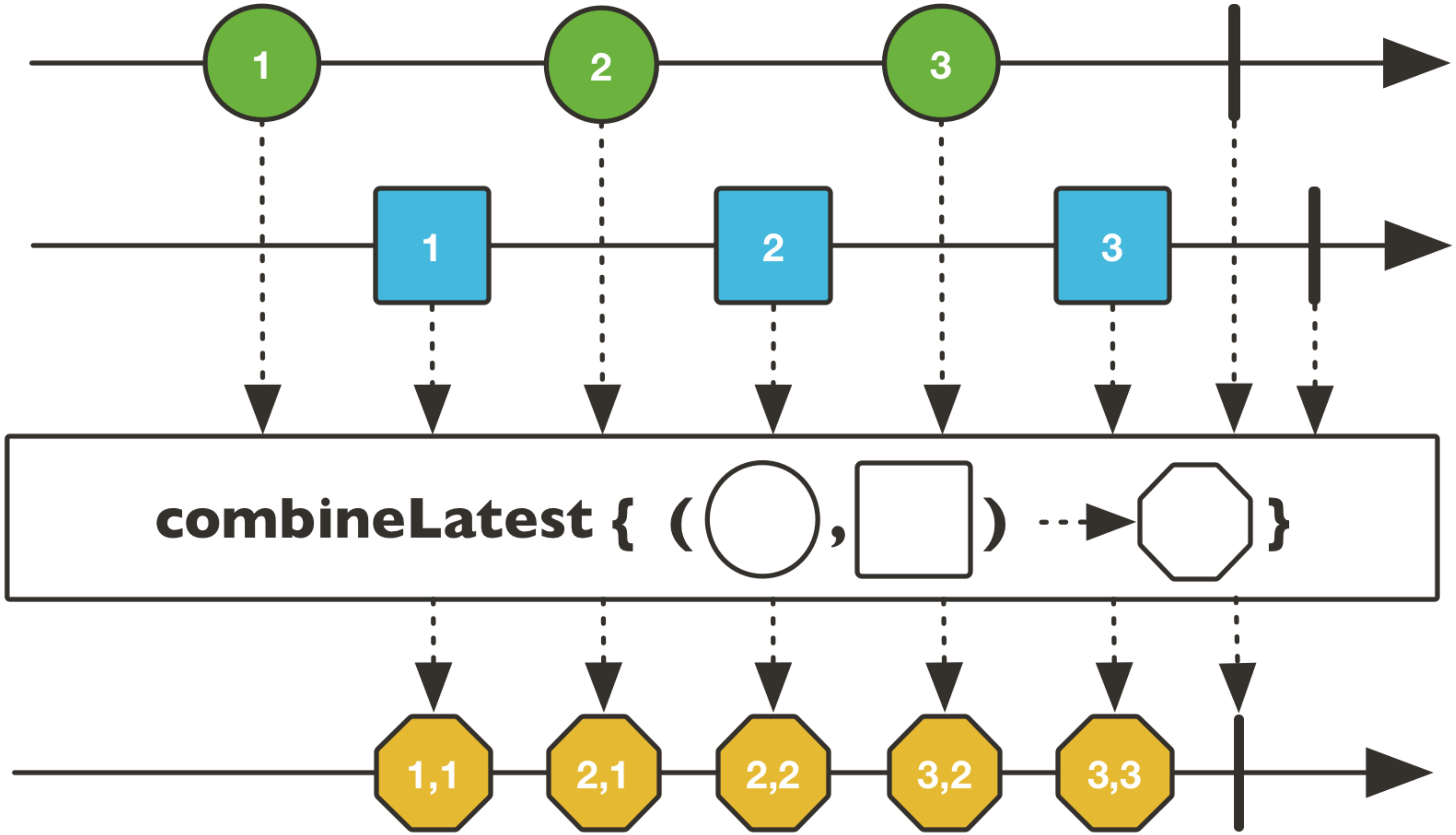
By ReactiveX - Creative Commons Attribution 3.0 License, Source: http://reactivex.io/documentation/observable.html

Example: Zip



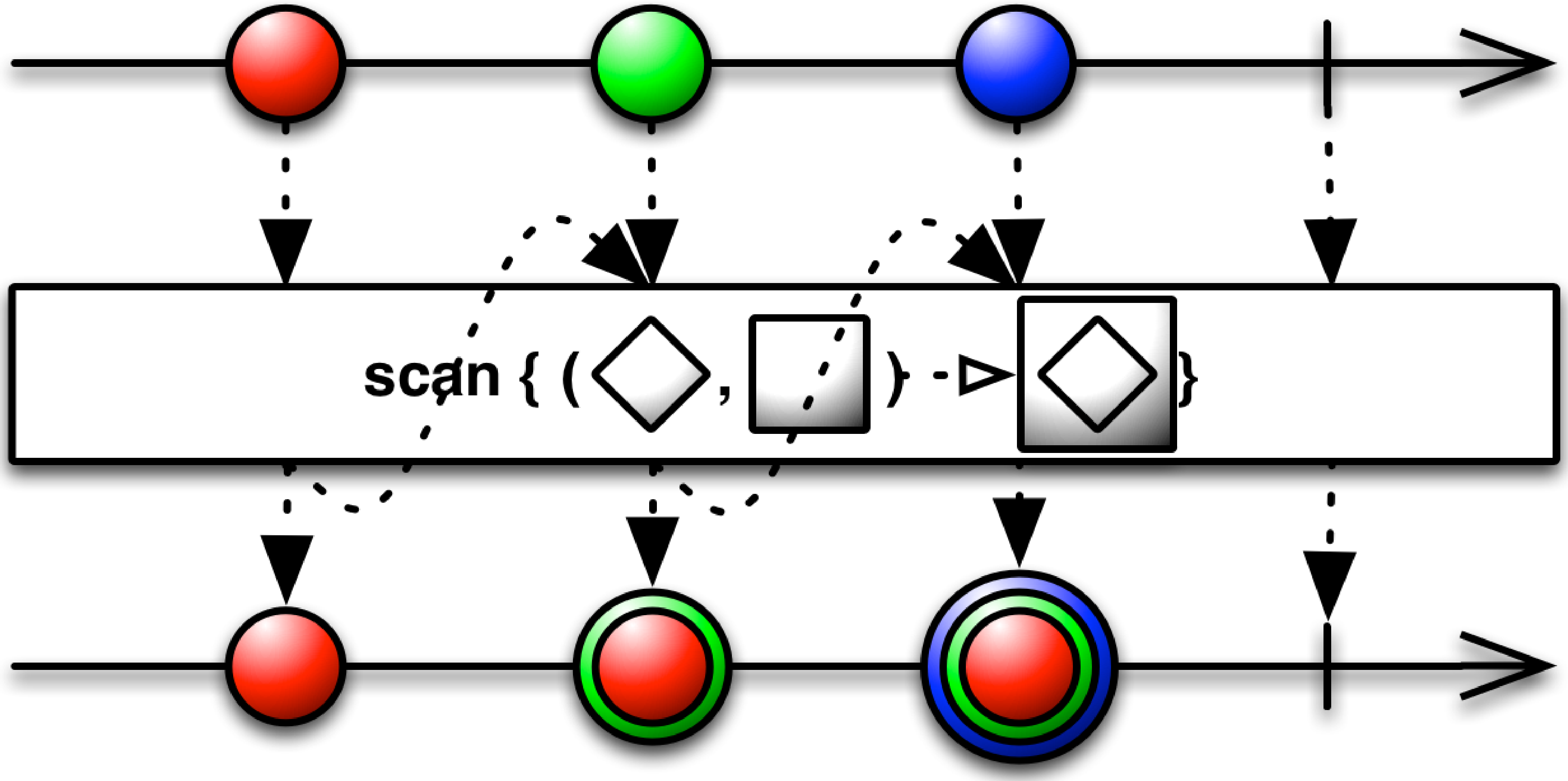
https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html, *Apache Software License 2.0*

Example: CombineLatest



https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html, *Apache Software License 2.0*

Redux == Rx Scan Opearator



Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

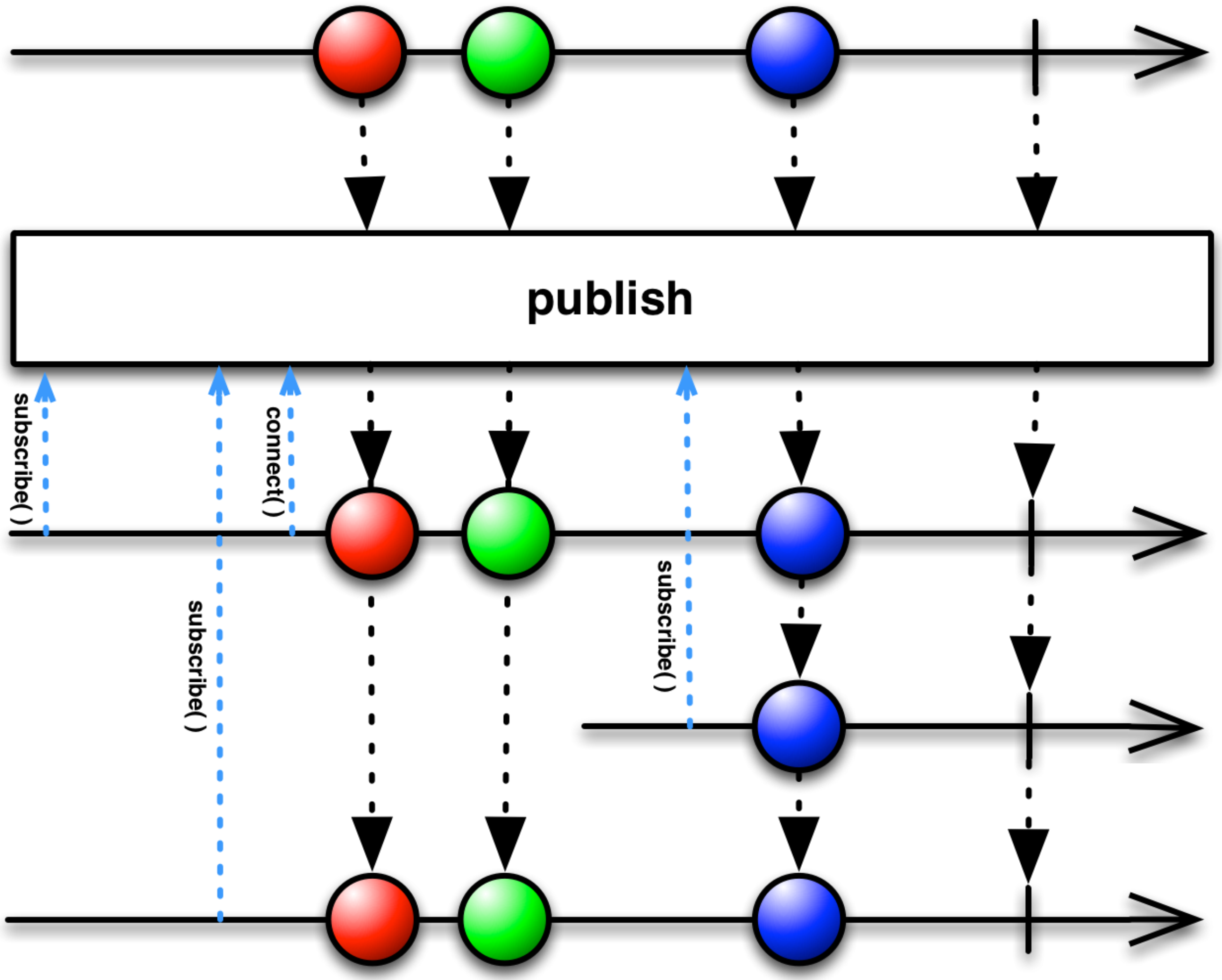
Hot and Cold Event Streams

- PULL-based (Cold Event Streams) – Cold streams (e.g. RxJava Observable / Flowable or Reactor Flow / Mono) are streams that run their sequence when and if they are subscribed to. They present the sequence from the start to each subscriber.

- PUSH-based (Hot Event Streams) – emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not. When subscription is made observer receives current events as they happen.

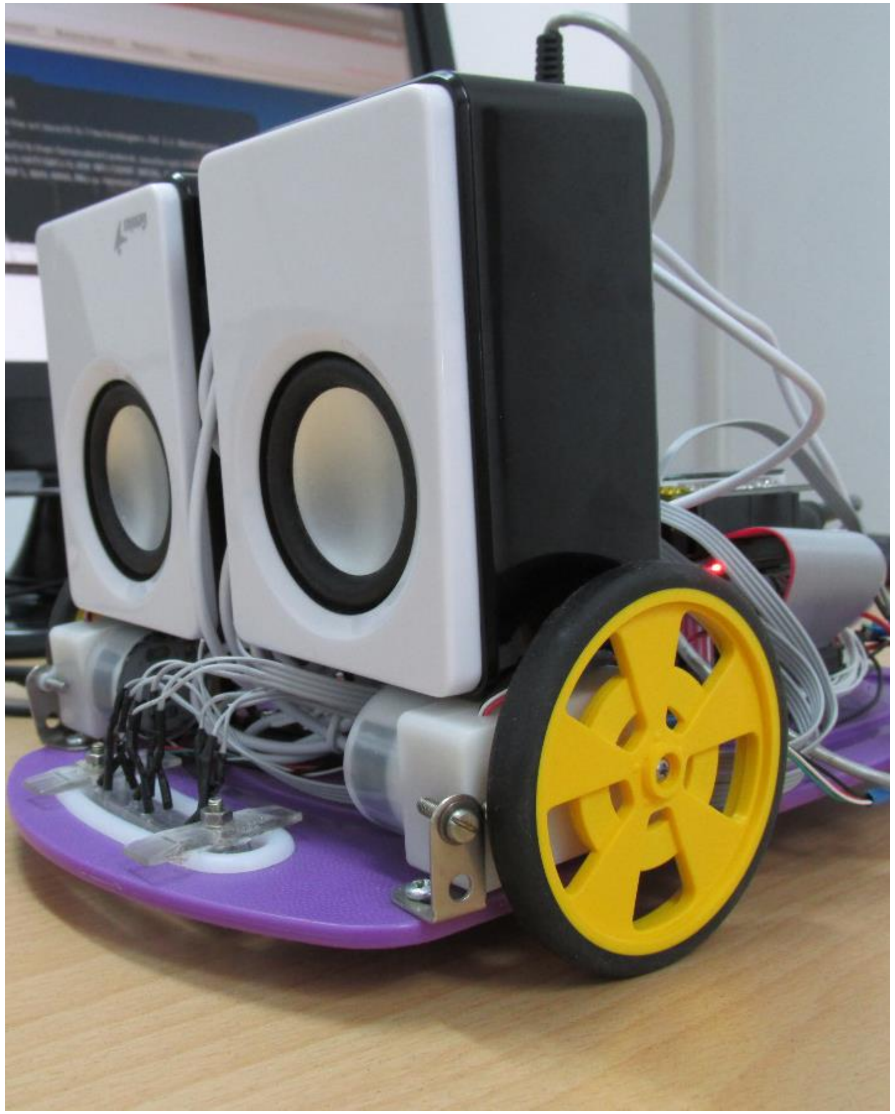
- *Example:* mouse events

Converting Cold to Hot Stream



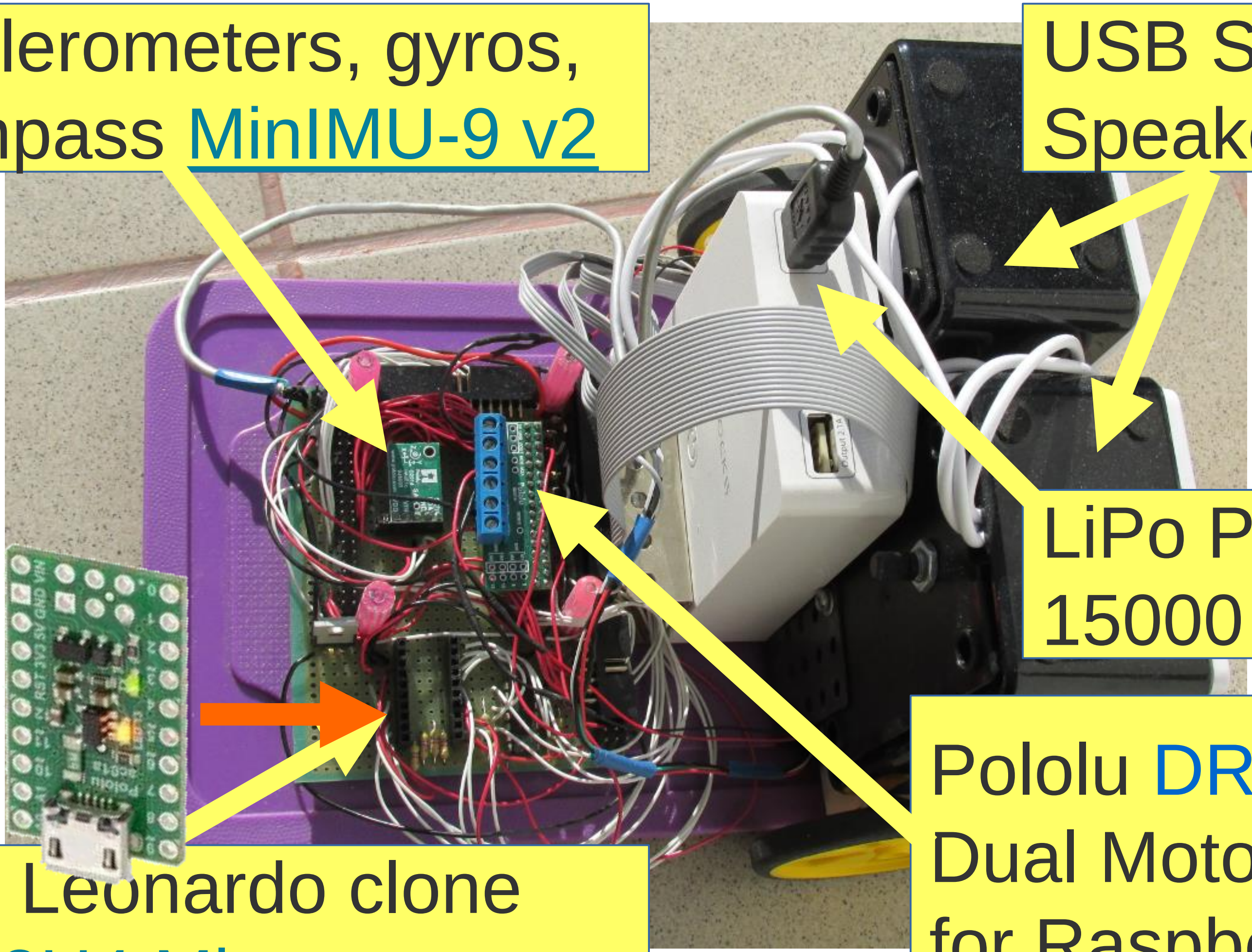
Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

IPTPI: Raspberry Pi + Ardunio Robot



- Raspberry Pi 2 (quad-core ARMv7 @ 900MHz) + Arduino Leonardo clone A-Star 32U4 Micro
- *Optical encoders* (custom), IR optical array, 3D accelerometers, gyros, and compass MinIMU-9 v2
- **IPTPI** is programmed in Python, Java and Go using: Wiring Pi, Numpy, Pandas, Scikit-learn, Pi4J, Reactor, RxJava, GPIO(Go)

IPTPI: Raspberry Pi + Ardunio Robot



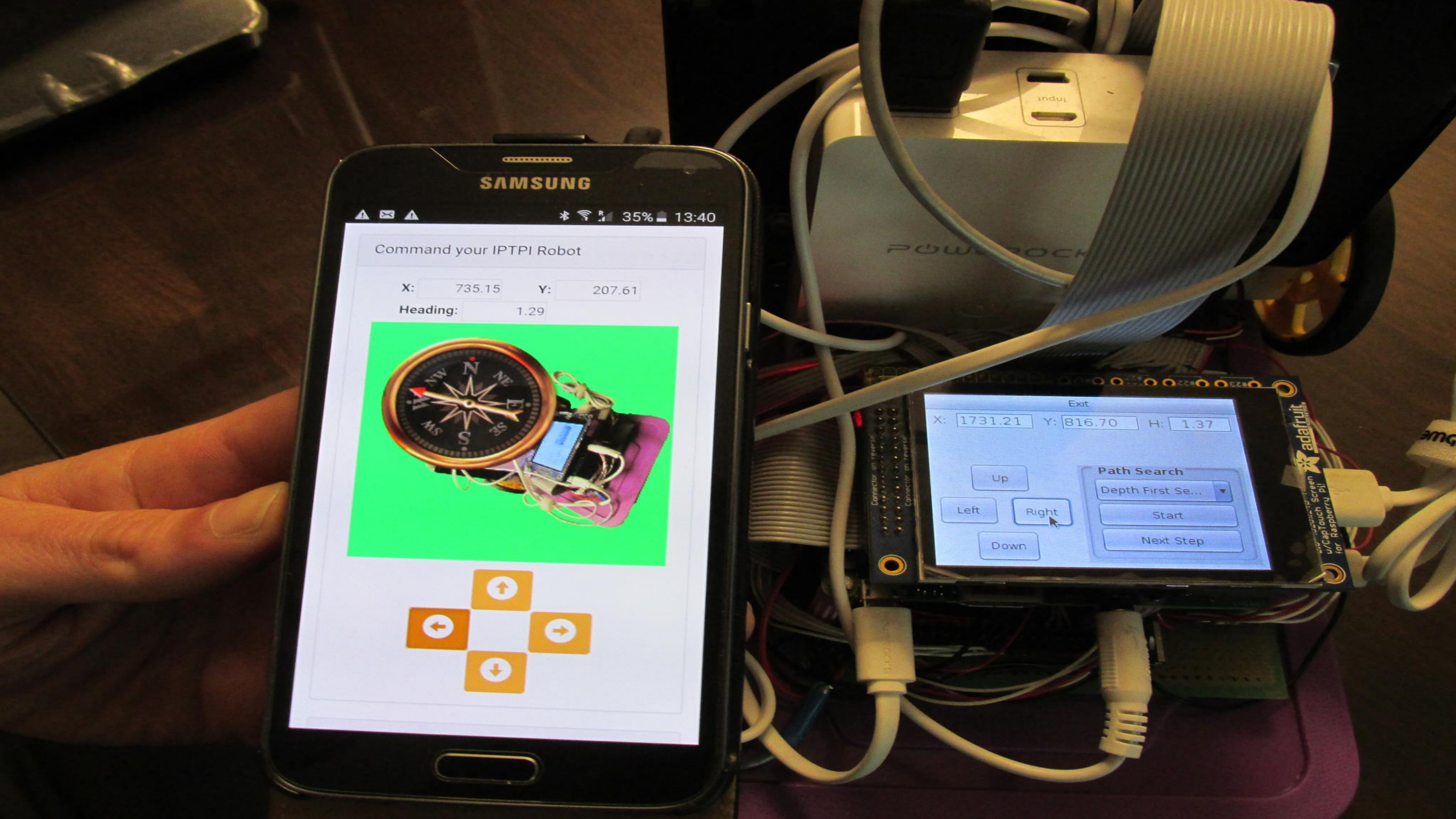
3D accelerometers, gyros, and compass MinIMU-9 v2

USB Stereo Speakers - 5V

LiPo Powebank 15000 mAh

Pololu DRV8835 Dual Motor Driver for Raspberry Pi

Arduino Leonardo clone A-Star 32U4 Micro



Command your IPTPI Robot

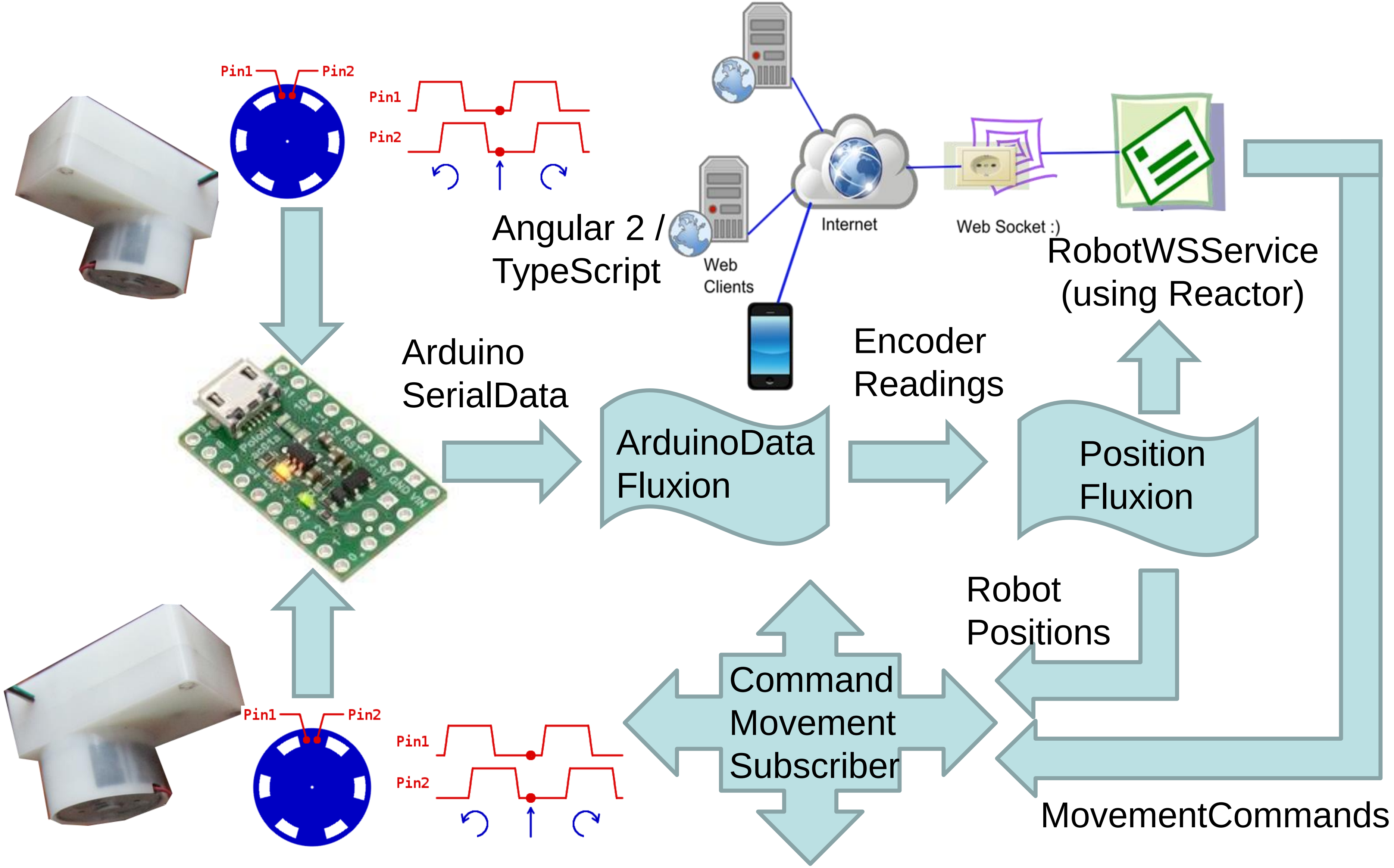
X: 735.15 Y: 207.61

Heading: 1.29

X: 1731.21 Y: 816.70 H: 1.37

Path Search: Depth First Se...

IPTPI Reactive Streams



Angular 2 / TypeScript

Arduino SerialData

ArduinoData Fluxion

Encoder Readings

Position Fluxion

RobotWSService (using Reactor)

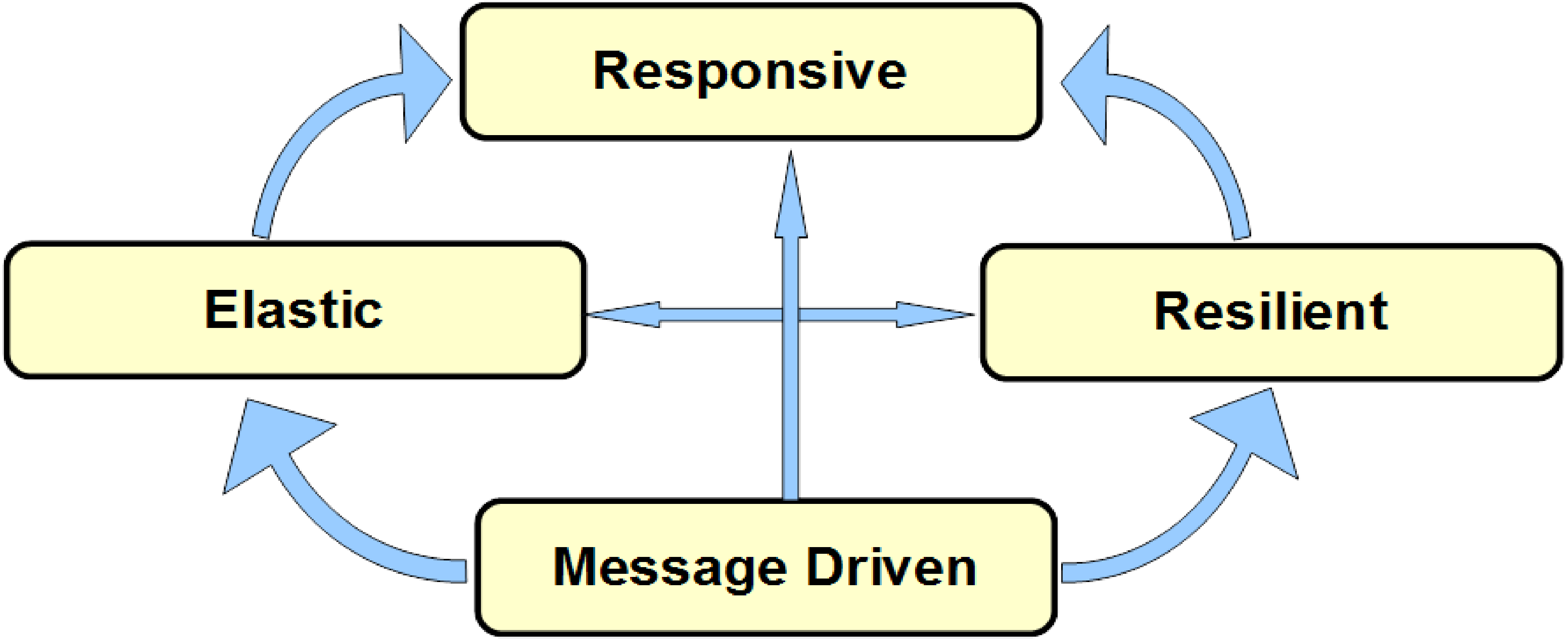
Robot Positions

Command Movement Subscriber

MovementCommands

Reactive Manifesto

http://www.reactivemanifesto.org



Scalable, Massively Concurrent

- **Message Driven** – asynchronous message-passing allows to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages [Reactive Manifesto].
- The main idea is to separate concurrent producer and consumer workers by using **message queues.**
- **Message queues** can be **unbounded** or **bounded** (limited max number of messages)
- **Unbounded** message queues can present memory allocation problem in case the producers outrun the consumers for a long period → **OutOfMemoryError**

Data / Event / Message Streams

“Conceptually, a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.”

Apache Flink: Dataflow Programming Model

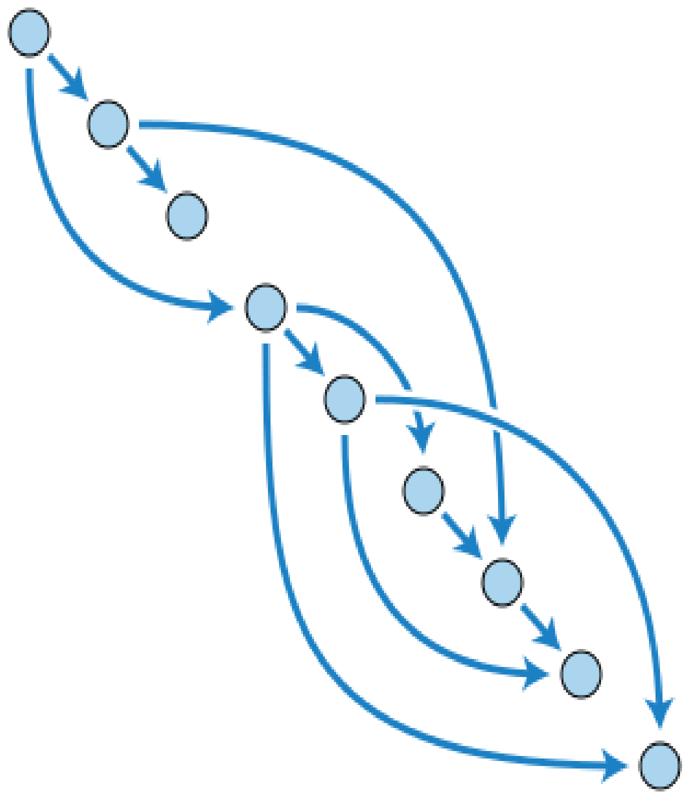
Data Stream Programming

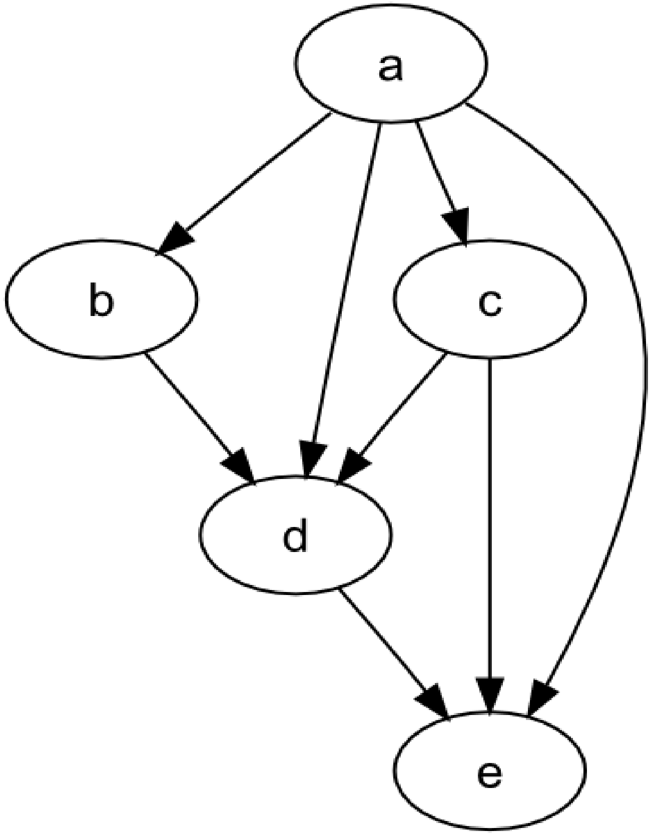
The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

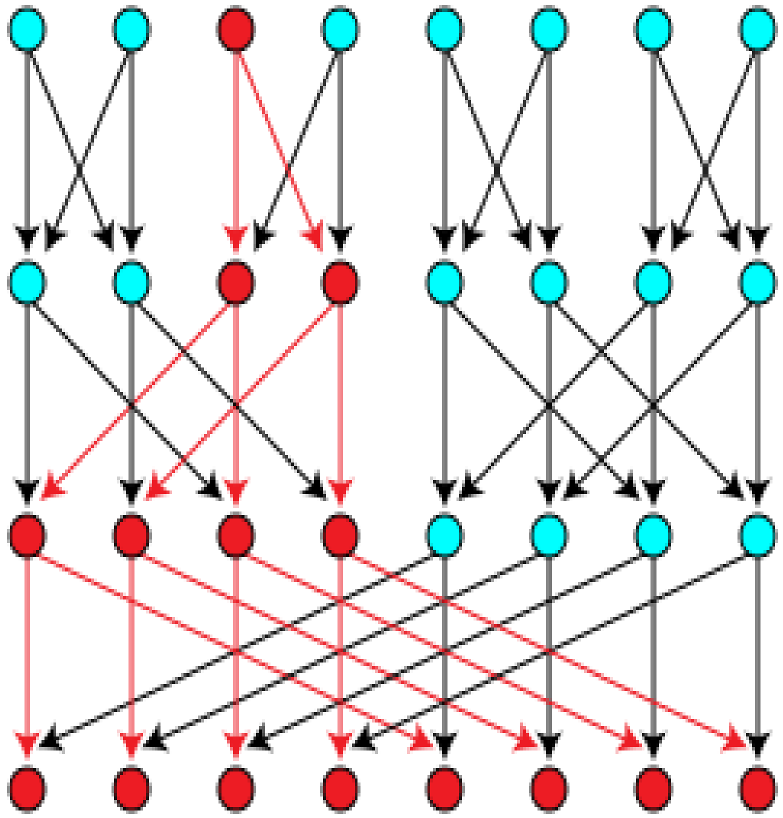
For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark:
New coopetition for squashing the Lambda Architecture?

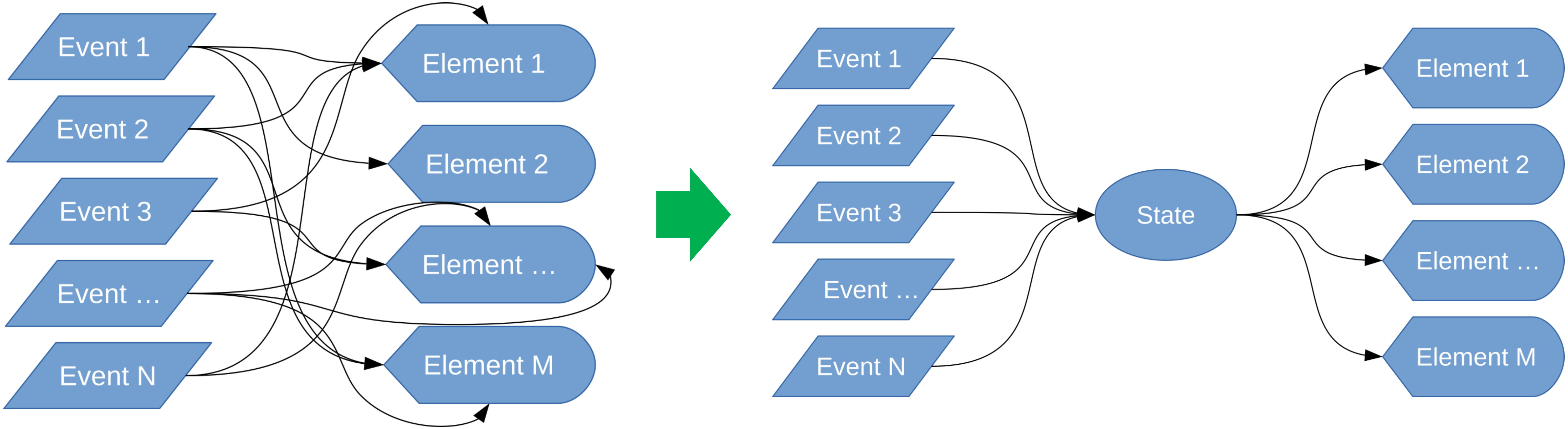
Direct Acyclic Graphs - DAG





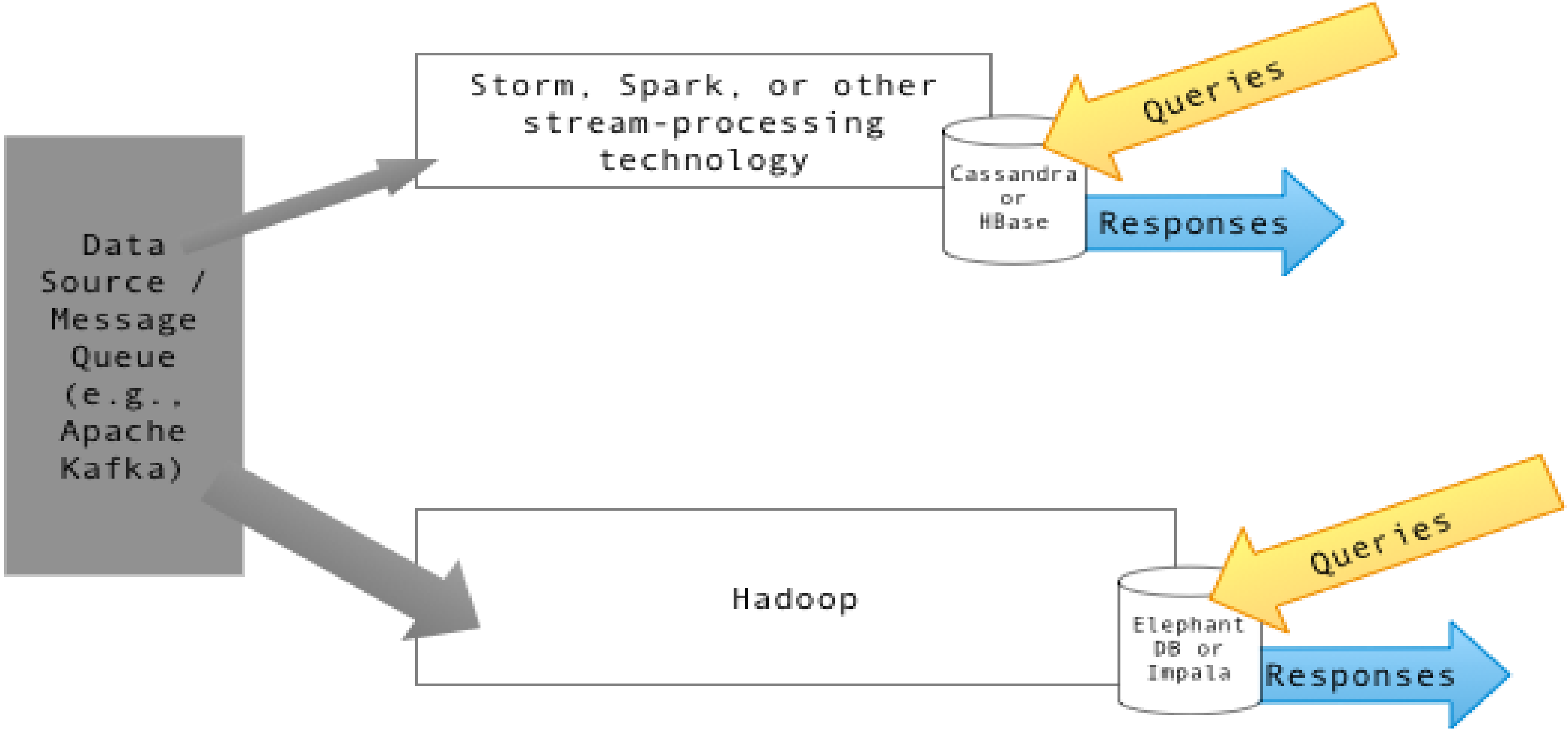


Event Sourcing – Events vs. Sate (Snapshots)



Lambda Architecture - I

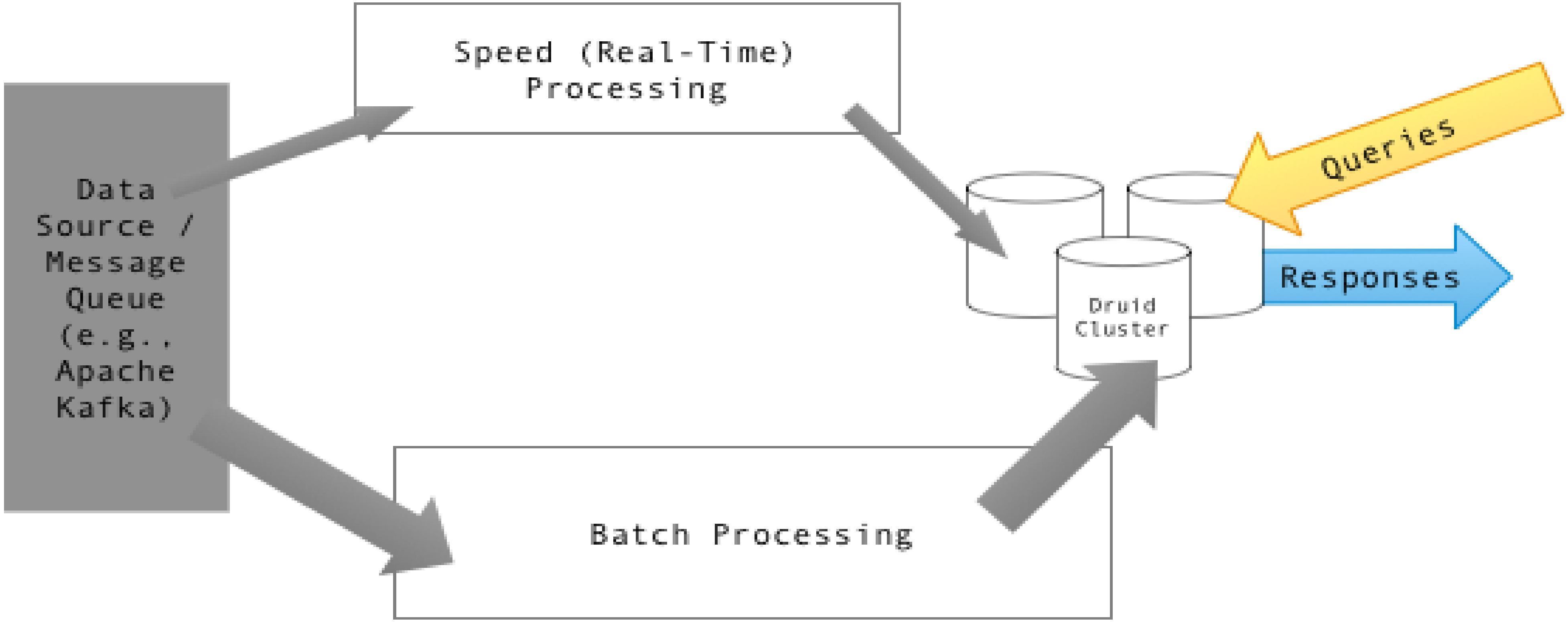
Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)


Storm, Spark, or other stream-processing technology
Cassandra or HBase
Queries
Responses
Data Source / Message Queue (e.g., Apache Kafka)
Hadoop
Elephant DB or Impala
Queries
Responses

https://commons.wikimedia.org/w/index.php?curid=34963987, By Textractor - Own work, CC BY-SA 4

Lambda Architecture - II

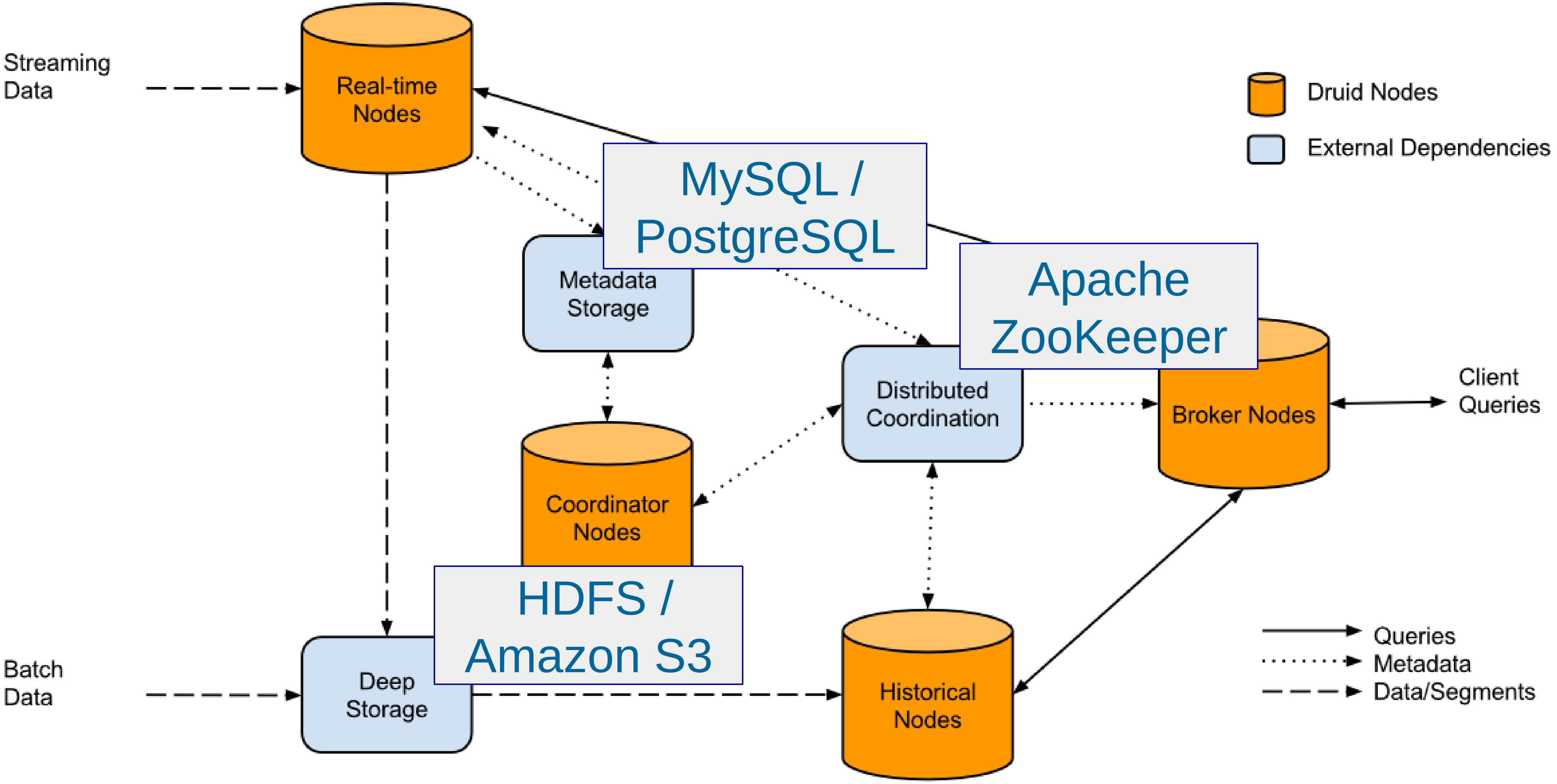
Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



Speed (Real-Time) Processing
Data Source / Message Queue (e.g., Apache Kafka)
Queries
Responses
Druid Cluster
Batch Processing

https://commons.wikimedia.org/w/index.php?curid=34963986, By Textractor - Own work, CC BY-SA 4

Lambda Architecture - Druid Distributed Data Store



https://commons.wikimedia.org/w/index.php?curid=33899448 By Fangjin Yang, GFDL

Kappa Architecture

Query = K (New Data) = K (Live streaming data)

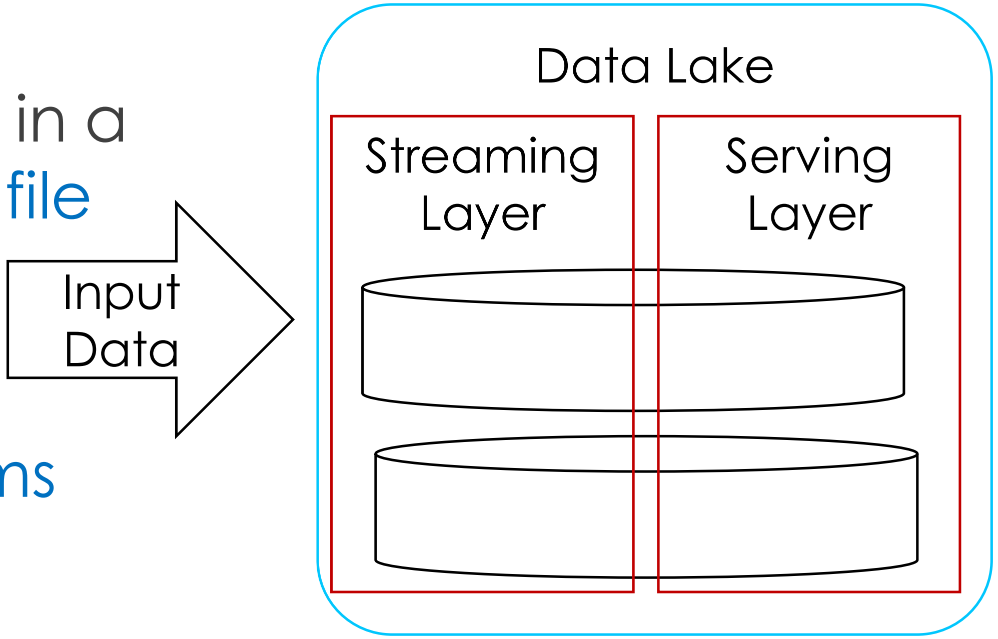
λ vs ϰ

- Proposed by Jay Kreps in 2014
- Real-time processing of distinct events
- Drawbacks of Lambda architecture:
  - It can result in coding overhead due to comprehensive processing
  - Re-processes every batch cycle which may not be always beneficial
  - Lambda architecture modeled data can be difficult to migrate
- Canonical data store in a Kappa Architecture system is an append-only immutable log (like Kafka, Pulsar)

Kappa Architecture II

Query = K (New Data) = K (Live streaming data)

- Multiple data events or queries are logged in a queue to be catered against a distributed file system storage or history.
- The order of the events and queries is not predetermined. Stream processing platforms can interact with database at any time.
- It is resilient and highly available as handling terabytes of storage is required for each node of the system to support replication.
- Machine learning is done on the real time basis



Zeta Architecture

- Main characteristics of Zeta architecture:
  - file system (HDFS, S3, GoogleFS),
  - realtime data storage (HBase, Spanner, BigTable),
  - modular processing model and platform (MapReduce, Spark, Drill, BigQuery),
  - containerization and deployment (cgroups, Docker, Kubernetes),
  - Software solution architecture (serverless computing – e.g. Amazon Lambda)
- Recommender systems and machine learning
- Business applications and dynamic global resource management (Mesos + Myriad, YARN, Diego, Borg).

Distributed Stream Processing – Apache Projects:

- Apache Spark is an open-source cluster-computing framework. Spark Streaming, Spark Mllib

- Apache Storm is a distributed stream processing – streams DAG

- Apache Samza is a distributed real-time stream processing framework.

Distributed Stream Processing – Apache Projects II

- Apache Flink - open source stream processing framework – Java, Scala

- Apache Kafka - open-source stream processing (Kafka Streams), real-time, low-latency, high-throughput, massively scalable pub/sub

- Apache Beam – unified batch and streaming, portable, extensible

What can FP offer to distributed computing ?

- No side-effects and immutable – variables FP facilitates code distribution over several CPU and eases concurrent programming
- Functions are better building components than objects:
  - Functions can be combined, sent remotely
  - Functions can be applied locally on distributed data sets (e.g. parallel stream, using Fork-Join pool underneath).
- In order to do the splitting of the work between multiple threads (forking) the Java Streams use:

  spliterator = split iterator
- The results can be joined after that in a single result (e.g. reduce)
- Example: Map – Reduce big data architecture (Google, Hadoop)

Iterators and Spliterators Example - Zip

```
public static <A, B, C> Stream<C> zip2(Stream<A> streamA, Stream<B> streamB, BiFunction<A, B, C> zipper) {
    Objects.requireNonNull(zipper);
    Spliterator<? extends A> aSpliterator = Objects.requireNonNull(streamA).spliterator();
    Spliterator<? extends B> bSpliterator = Objects.requireNonNull(streamB).spliterator();

    int characteristics = ((aSpliterator.characteristics() & bSpliterator.characteristics()
            & ~(Spliterator.DISTINCT | Spliterator.SORTED))    // Zipping looses DISTINCT and SORTED characteristics
            | (aSpliterator.characteristics() & SIZED | bSpliterator.characteristics() & SIZED));

    long zipSize = (aSpliterator.getExactSizeIfKnown() >= 0) ?
            ((bSpliterator.getExactSizeIfKnown() >= 0) ?
                    Math.min(aSpliterator.estimateSize(), bSpliterator.estimateSize())
                    : aSpliterator.estimateSize())
            :bSpliterator.getExactSizeIfKnown();

    final Iterator<A> iteratorA = Spliterators.iterator(aSpliterator);
    final Iterator<B> iteratorB = Spliterators.iterator(bSpliterator);
    final Iterator<C> iteratorC = new Iterator<C>() {
        @Override
        public boolean hasNext() {
            return iteratorA.hasNext() && iteratorB.hasNext();
        }

        @Override
        public C next() {
            return zipper.apply(iteratorA.next(), iteratorB.next());
        }
    };
    Spliterator<C> split = zipSize > 0 ? Spliterators.spliterator(iteratorC, zipSize, characteristics):
            Spliterators.spliteratorUnknownSize(iteratorC, characteristics);
    return StreamSupport.stream(split, streamA.isParallel() || streamA.isParallel());
}
```

Stream API

Stream Creation

- Empty Stream

```
Stream<String> streamEmpty = Stream.empty();
public Stream<String> streamOf(List<String> list) {
    return list == null | | list.isEmpty() ? Stream.empty() : list.stream();
}
```

- Stream of Collection

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

- Stream of Array

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
String[] arr = {"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Stream Creation II

- Stream.builder()

```
Stream<String> streamBuilder =
  Stream.<String>builder().add("a").add("b").add("c").build();
```

- Stream.generate() – using Supplier<T> (method *generate()*)

```
Stream<String> streamGenerated =Stream.generate(() -> “value").limit(10);
```

- Stream.iterate()

```
Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
```

Primitive Streams: IntStream, LongStream, DoubleStream

- range(), rangeClosed()

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
```

Stream.generate() – using Supplier<T> (method *generate()*)

```
Stream<String> streamGenerated =Stream.generate(() -> “value").limit(10);
```

- Random class methods

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

String and File Streams

- Stream of String

```
IntStream streamOfChars = "abc".chars();
```

- Stream of File lines

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
                        Files.lines(path, Charset.forName("UTF-8"));
```

Referencing a Stream

```
Stream<String> stream =
 Stream.of("a", "b", "c").filter(element -> element.contains("b"));
Optional<String> anyElement = stream.findAny();
```

```
public static void tryStreamTraversal() {
    Stream<String> userNameStream = userNames();
    userNameStream.forEach(System.out::println);
    try {
        userNameStream.forEach(System.out::println);
    } catch(IllegalStateException e) {
        System.out.println("stream has already been operated upon or closed");
    }
}
```

Referencing a Stream

```
List<String> elements =
  Stream.of("a", "b", "c").filter(element -> element.contains("b"))
    .collect(Collectors.toList());
Optional<String> anyElement = elements.stream().findAny();
Optional<String> firstElement = elements.stream().findFirst();
```

Building Stream Pipelines

```
Stream<String> onceModifiedStream =  Stream.of("abcd", "bbcd", "cbcd")
   .skip(1);

Stream<String> stringStream =
  stream.skip(1).map(element -> element.substring(0, 3));

List<String> list = Arrays.asList("abc1", "abc2", "abc3");
long size = list.stream().skip(1)
  .map(element -> element.substring(0, 3)). distinct().count();
```

Execution Order

```
AtomicInteger counter1 = new AtomicInteger();
var result = list.stream()
        .map(element -> {
            counter1.incrementAndGet();
            System.out.println("Calling map #" + counter1);
            return element.substring(0, 5);
        })
        .skip(2)
        .collect(Collectors.toList());
System.out.println(result);
```

```
Calling map #1

Calling map #2

Calling map #3

[hello]
```

```
AtomicInteger counter2 = new AtomicInteger();
var result2 = list.stream()
        .skip(2)
        .map(element -> {
            counter2.incrementAndGet();
            System.out.println("Calling map #" + counter2);
            return element.substring(0, 5);
        }).collect(Collectors.toList());
System.out.println(result2);
```

```
Calling map #1

[hello]
```

Stream Reducers

- Standatrd reducers: count(), max(), min(), sum()
- Custom reducers:
  - identity – the initial value for an accumulator, or a default value if a stream is empty and there is nothing to accumulate
  - accumulator –function that specifies element aggregation logic:

    (ACC, VAL) -> NEW_ACC

  - combiner – a function which aggregates the results of the accumulator. We only call combiner in a parallel mode to reduce the results of accumulators from different threads (must be compatible with the accumulator function)

Reducer Demo

```
OptionalInt reduced = IntStream.rangeClosed(1, 10).reduce((a, b) -> a + b);
System.out.println("Reduced: " + reduced);

int reducedWithInitVal = IntStream.rangeClosed(1, 10).reduce(1, (a, b) -> a * b);
System.out.println("Reduced with initial value: " + reducedWithInitVal);

int reducedParallel = IntStream.rangeClosed(1, 10000).boxed().parallel()
        .reduce(0,
                (a, b) -> a + b,
                (a, b) -> {
                    System.out.printf("combiner called for %s and %s%n", a, b);
                    return a + b;
                });
System.out.println("Reduced with accumulator and combiner: " + reducedParallel);
```

Reducer Demo

```
var path = Paths.get("src/course/stream/demos/StreamApiDemo04.java");
    String fResult = Files.lines(path)
        .reduce(new Tuple2<>("", 1), (acc, line) -> // accumulator
            new Tuple2<>(acc.getV1() + acc.getV2() + ": " + line + "\n", acc.getV2() + 1),
        (acc1, acc2) -> // combimner
            new Tuple2<>(acc1.getV1() + "\n" + acc2.getV2(), 0)).getV1();
    System.out.println(fResult);
```

Поточно програмиране (1)

Примери:

```
books.stream().map(book ->
    book.getTitle()).collect(Collectors.toList());
books.stream()
      .filter(w -> w.getDomain() == PROGRAMMING)
      .mapToDouble(w -> w.getPrice()) .sum();
document.getPages().stream()
     .map(doc -> Documents.characterCount(doc))
     .collect(Collectors.toList());
document.getPages().stream()
     .map(p -> pagePrinter.printPage(p))
     .forEach(s -> output.append(s));
```

Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial) Автор: Graham Allan, Лиценз: GNU GENERAL PUBLIC LICENSE v2

Поточно програмиране (2)

Примери:

```
document.getPages().stream()
        .map(page -> page.getContent())
        .map(content -> translator.translate(content))
        .map(translated -> new Page(translated))
        .collect(Collectors.collectingAndThen(
                Collectors.toList(),
                pages -> new
        Document(translator.translate(document.getTitle()), pages)));
```

- Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial)
- Автор: Graham Allan, Лиценз: GNU GENERAL PUBLIC LICENSE v2

Референции към методи

- Статични методи на клас – Class::staticMethod
- Методи на конкретни обектни инстанции – object::instanceMethod
- Методи на инстанции реферирани чрез класа – Class::instanceMethod
- Конструктори на обекти от даден клас – Class::new

```
Comparator<Person> namecomp = Comparator.comparing(Person::getName);

Arrays.stream(pageNumbers).map(doc::getPageContent).forEach(Printers::print);

pages.stream().map(Page::getContent).forEach(Printers::print);
```

- Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial)
- Автор: Graham Allan, Лиценз: GNU GENERAL PUBLIC LICENSE v2

Статични и Default методи в интерфейси

- Методите с реализация по подразбиране в интерфейс са известни още като virtual extension methods или defender methods, защото дават възможност интерфейсите да бъдат разширявани, без това да води до невъзможност за компилация на вече съществуващи реализации на тези интерфейси (което би се получило ако старите реализации не имплементират новите абстрактни методи).
- Статичните методи дават възможност за добавяне на помощни (utility) методи – например factory методи директно в интерфейсите които ги ползват, вместо в отделни помощни класове (напр. Arrays, Collections).

- Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial)
- Автор: Graham Allan,

Пример за default и static методи в интерфейс

- @FunctionalInterface

```
interface Event {
    Date getDate();
    default String getDateFormatted() {
            return String.format("%1$td.%1$tm.%1$tY", getDate());
    }
    public static <T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> getKey) {
        return (c1, c2) -> getKey.apply(c1).compareTo(getKey.apply(c2));
    }
}
Event current = () -> new Date();
System.out.println(current.getDateFormatted());
```

Функционално програмиране и монади

- Понятие за **монада** във функционалното програмиране (теория на категориите) – **Монадата** е множество от три елемента:

– Параметризиран тип **M<T>**

– "**unit**" функция: **T -> M<T>**

– "**bind**" операция: **bind(M<T>, f:T -> M<U>) -> M<U>**

- В Java 8 пример за монада е класът **java.util.Optional<T>**

Параметризиран тип: **Optional<T>**

– "**unit**" функции: **Optional<T> of(T value) , Optional<T> ofNullable(T value)**

– "**bind**" операция: **Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**

Exercise: Java 8 Stream API Koans

Available @GitHub: https://github.com/iproduct/course-java-web-development/tree/master/lambda-tutorial-master

1. Read carefully the JavaDoc for the unit tests stating the problem to solve: Exercise_1_Test.java, Exercise_2_Test.java, Exercise_3_Test.java, Exercise_4_Test.java and Exercise_5_Test.java
2. Fill the code in place of comments like: // [your code here]
3. Run the unit tests to check if your proposed solution is correct. If not return to step 1.

Литература и интернет ресурси

- Oracle tutorial – lambda expressions - http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Java SE 8: Lambda Quick Start - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- OpenJDK Lambda Tutorial - https://github.com/AdoptOpenJDK/lambda-tutorial

Thank’s for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD