JavaLand: Introduction to Apache Kafka & Kafka Streams

Trayan Iliev, IPT – IT Education Evolved, http://iproduct.org/

Disclaimer

All information presented in this document and all supplementary materials and programming code represent only my personal opinion and current understanding and has not received any endorsement or approval by IPT - Intellectual Products and Technologies or any third party. It should not be taken as any kind of advice, and should not be used for making any kind of decisions with potential commercial impact.

The information and code presented may be incorrect or incomplete. It is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. In no event shall the author or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the information, materials or code presented or the use or other dealings with this information or programming code.

About Me

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies

– Oracle® certified programmer 15+ Y

– 12+ years IT trainer

– End-to-end reactive fullstack apps with Go, Python, Java, ES6/7, TypeScript, Angular, React and Vue.js

– Machine Learning (ML) and Deep Learning (DL) with Python, Neuro-Symbolic learning, big data/event processing in realtime (Spark, Kafka, etc.), IoT, web and mobile technologies.

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Lecturer @ Sofia University – courses: Fullstack React, Angular & TypeScript, Spring 5 Reactive, Distributed Machine Learning, Practical Robotics & IoT, Kotlin, Golang

– Robotics / smart-things/ IoT enthusiast, RoboLearn hackathons organizer

Where to Find The Code and Presentation?

https://github.com/iproduct/kafka-streams-javaland

Machine Learning + Big Data in Real Time + Cloud Technologies

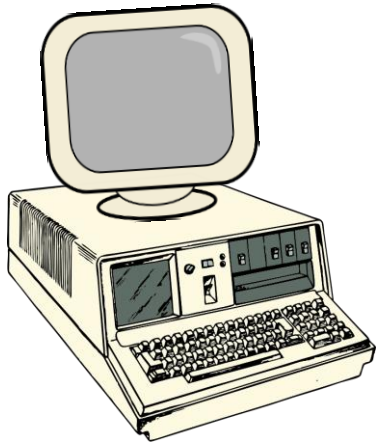
=> The Future of Intelligent Systems

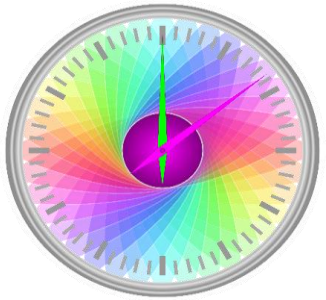
Agenda for This Lesson - I

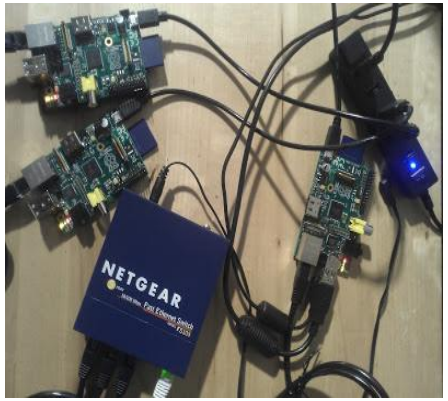
- Introduction to distributed reactive systems (Reactive Manifesto)
- State and behavior of distributed systems – event sourcing.
- Distributed state consistency guarantees – eventual consistency.
- CAP theorem.
- Technologies and architectures for Big Data processing in real time – Lambda and Cappa architectures.
- Web scale systems, data lakes and Zeta architecture.

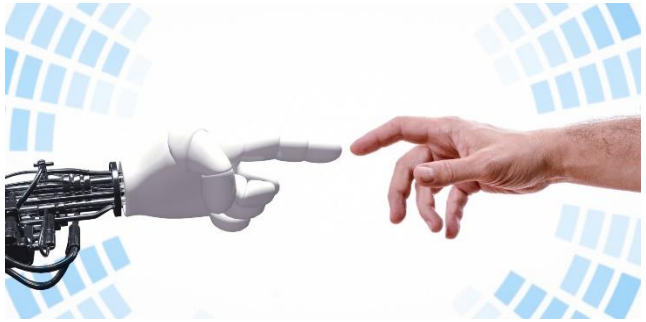
Asynchronous Event Streams in Real-time

The Evolution of Data Processing

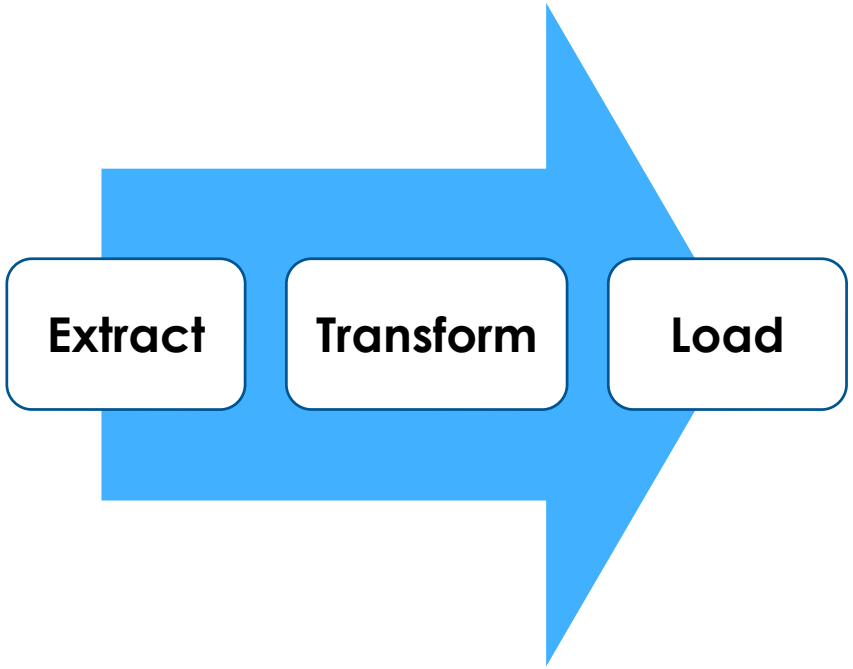


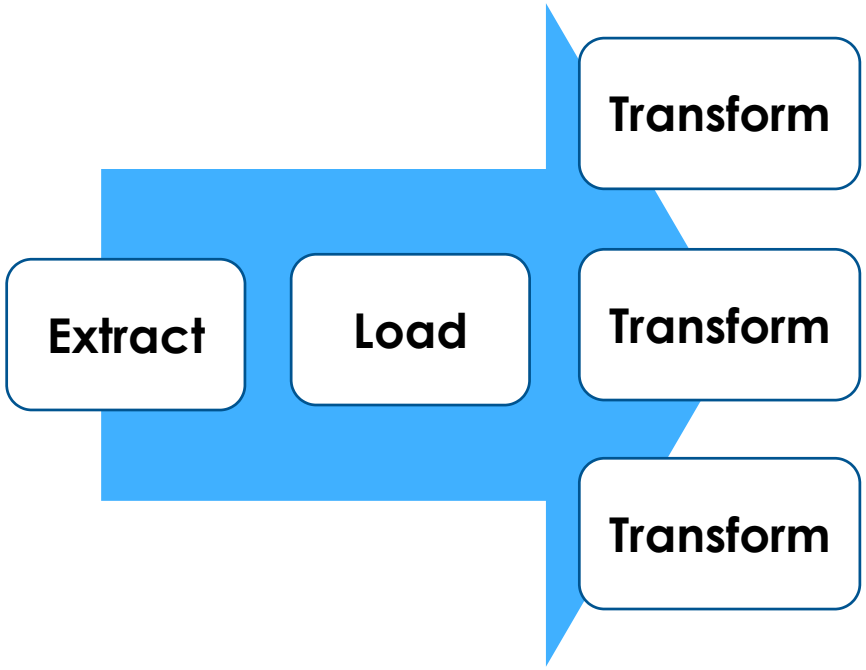






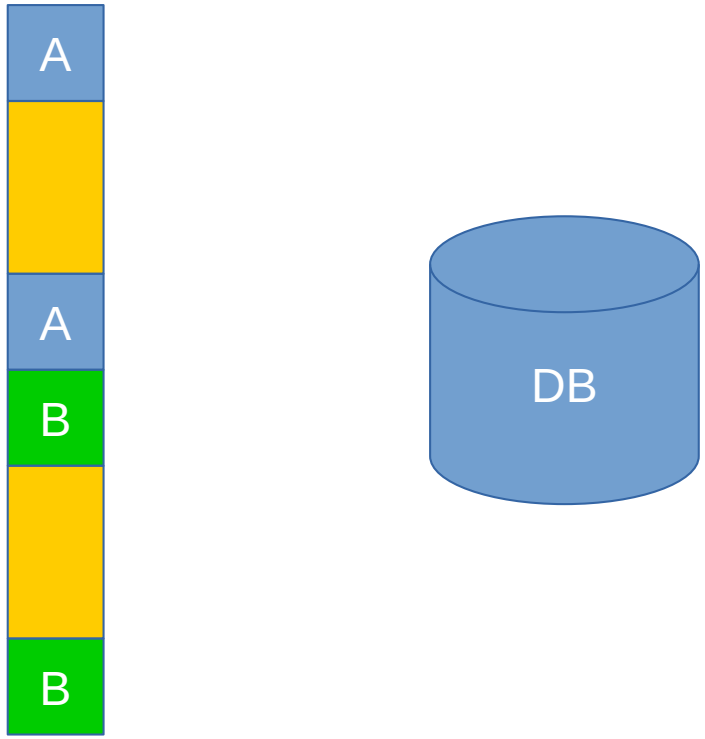
Batch Processing



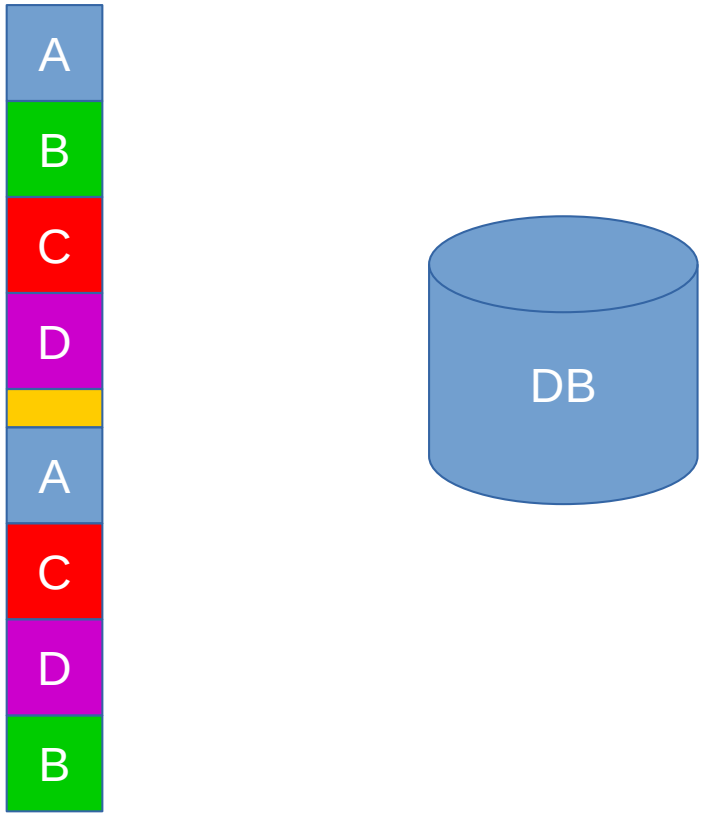


Synchronous vs. Asynchronous IO

Synchronous



Asynchronous



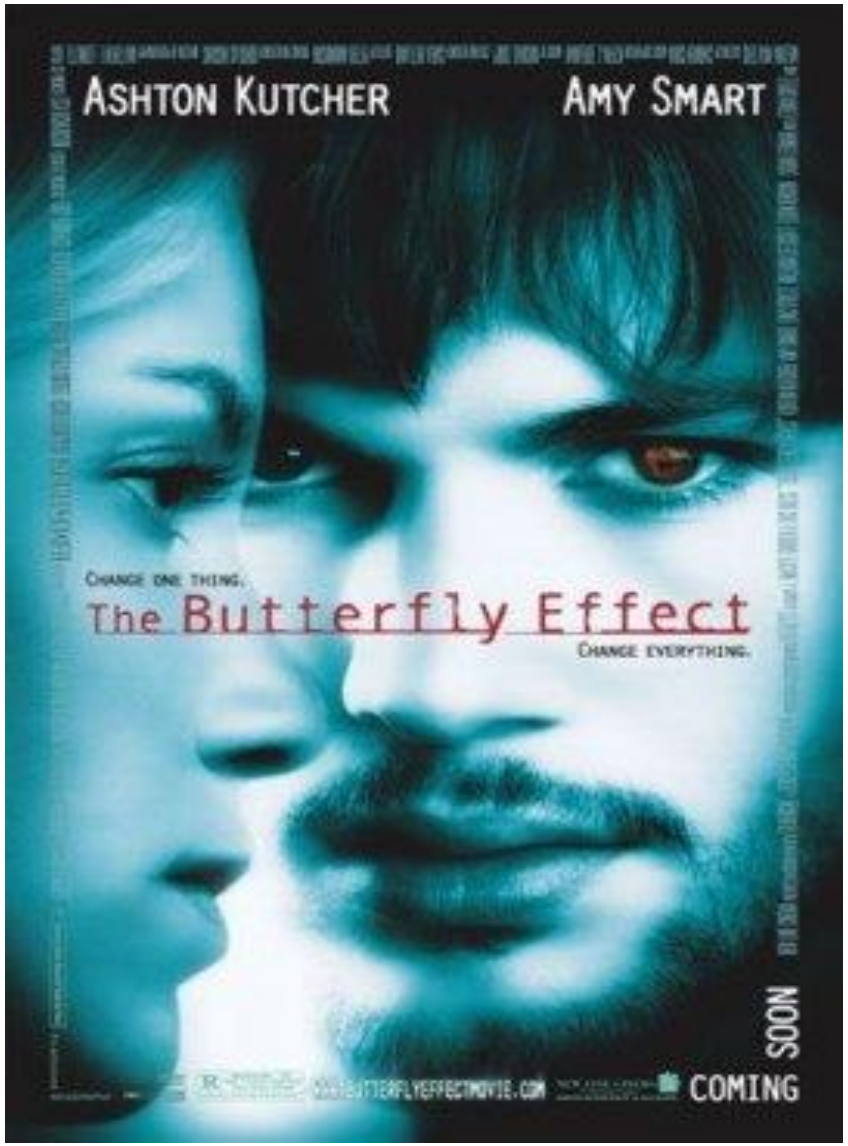
Functional Reactive Programming

Imperative and Reactive

We live in a Connected Universe

... there is hypothesis that all the things in the Universe are intimately connected, and you can not change a bit without changing all.

Action – Reaction principle is the essence of how Universe behaves.



Source: https://en.wikipedia.org/wiki/The_Butterfly_Effect#/media/File:Butterflyeffect_poster.jpg
Fair use, File:Butterflyeffect poster.jpg, Uploaded by Yaminator, Uploaded: 8 July 2008

Imperative vs. Reactive

- Reactive Programming: using static or dynamic data flows and propagation of change
- Example: a := b + c
- Functional Programming: evaluation of mathematical functions, avoids changing-state and mutable data, declarative programming
- Side effects free => much easier to understand and predict the program behavior.

Ex. (RxGo):

```
observable := rxgo.Just("Hello", "Reactive", "World", "from", "RxGo")().
        Map(ToUpper). // map to upper case
        Filter(LengthGreaterThan4) // greaterThan4 func filters values > 4
for item := range observable.Observe() {
        fmt.Println(item.V)
}
```

Functional Reactive Programming (FRP)

- According to Connal Elliot's (ground-breaking paper at Conference on Functional Programming, 1997), FRP is:

  (a) Denotative

  (b) Temporally continuous

- FRP is asynchronous data-flow programming using the building blocks of functional programming (map, reduce, filter, etc.) and explicitly modeling time

Reactive Programming

- ReactiveX (Reactive Extensions) - open source polyglot (http://reactivex.io):

  Rx = Observables + Flow transformations + Schedulers

- Go: RxGo, Kotlin: RxKotlin, Java: RxJava, JavaScript: RxJS, Python: RxPY, C#: Rx.NET, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby, ...
- Reactive Streams Specification (http://www.reactive-streams.org/):
  - Publisher – provider of potentially unbounded number of sequenced elements, according to Subscriber(s) demand (backpressure): onNext* (onError | onComplete)
  - Subscriber, Subscription
  - Processor = Subscriber + Publisher

ReactiveX: Observable vs. Iterable

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

`Iterable`

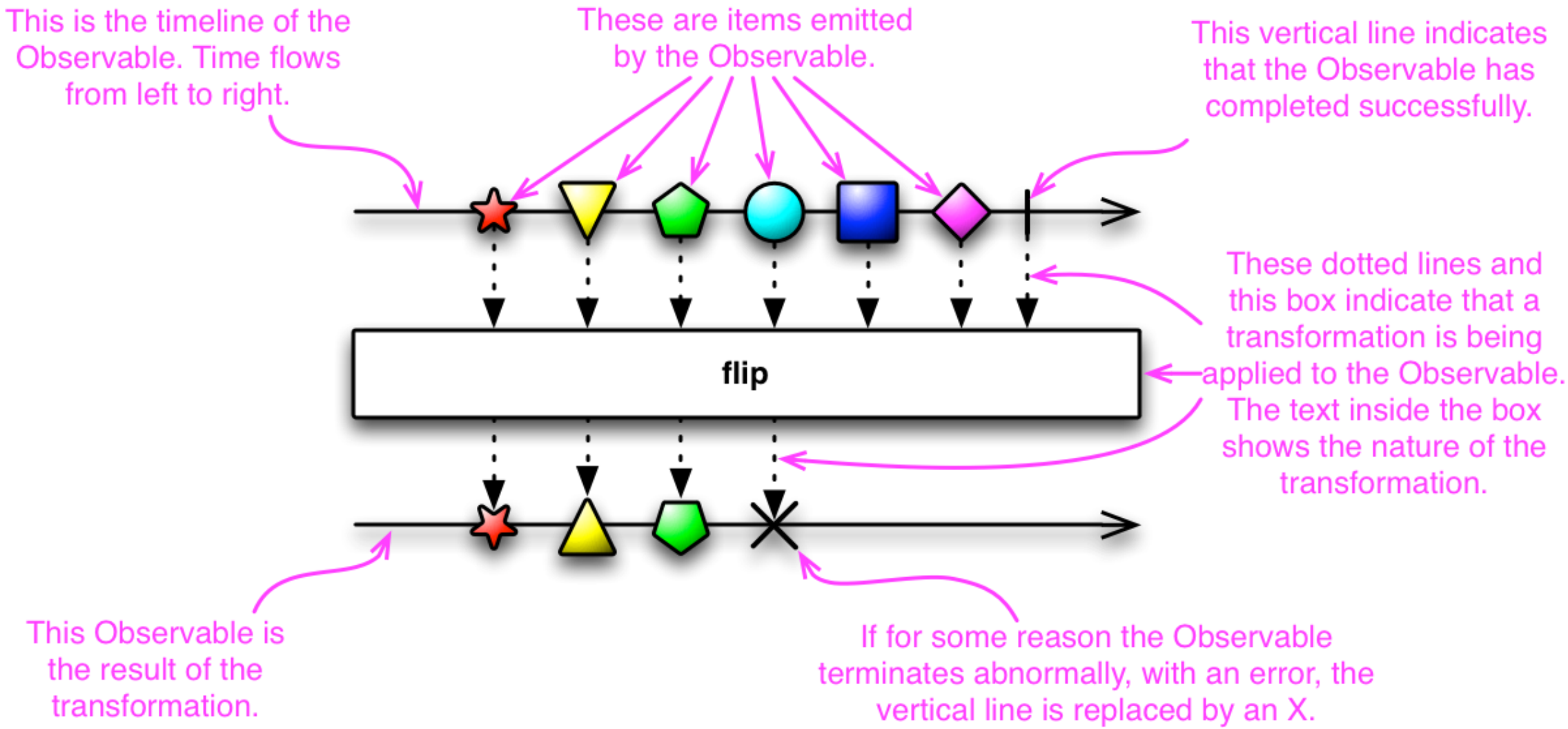
```
getDataFromLocalMemory()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transforme
d" })
  .forEach({ println "next => " + it
})
```

`Observable`

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

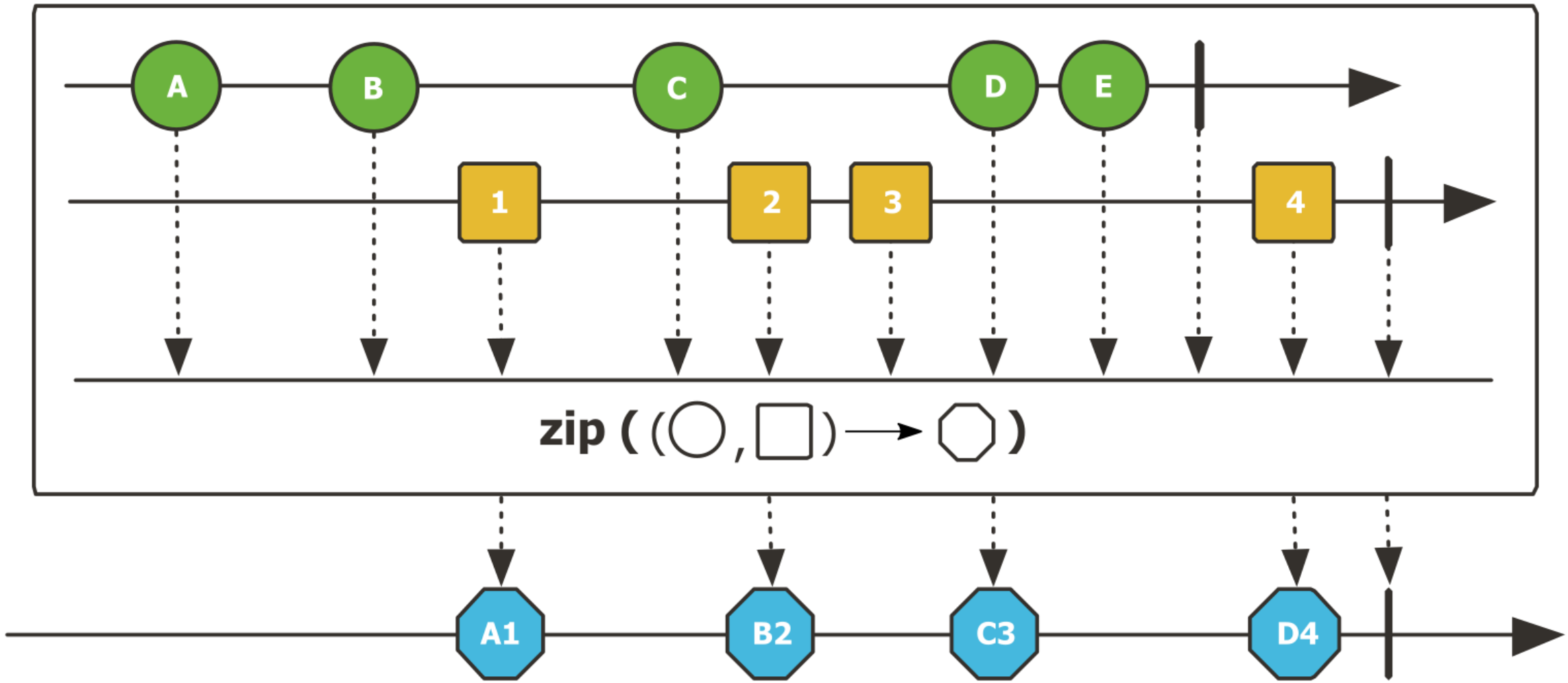
You can think of the Observable class as a **"push"** equivalent to Iterable, which is a **"pull."** With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast, with an Observable the producer pushes values to the consumer whenever values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

ReactiveX Observable – Marble Diagrams

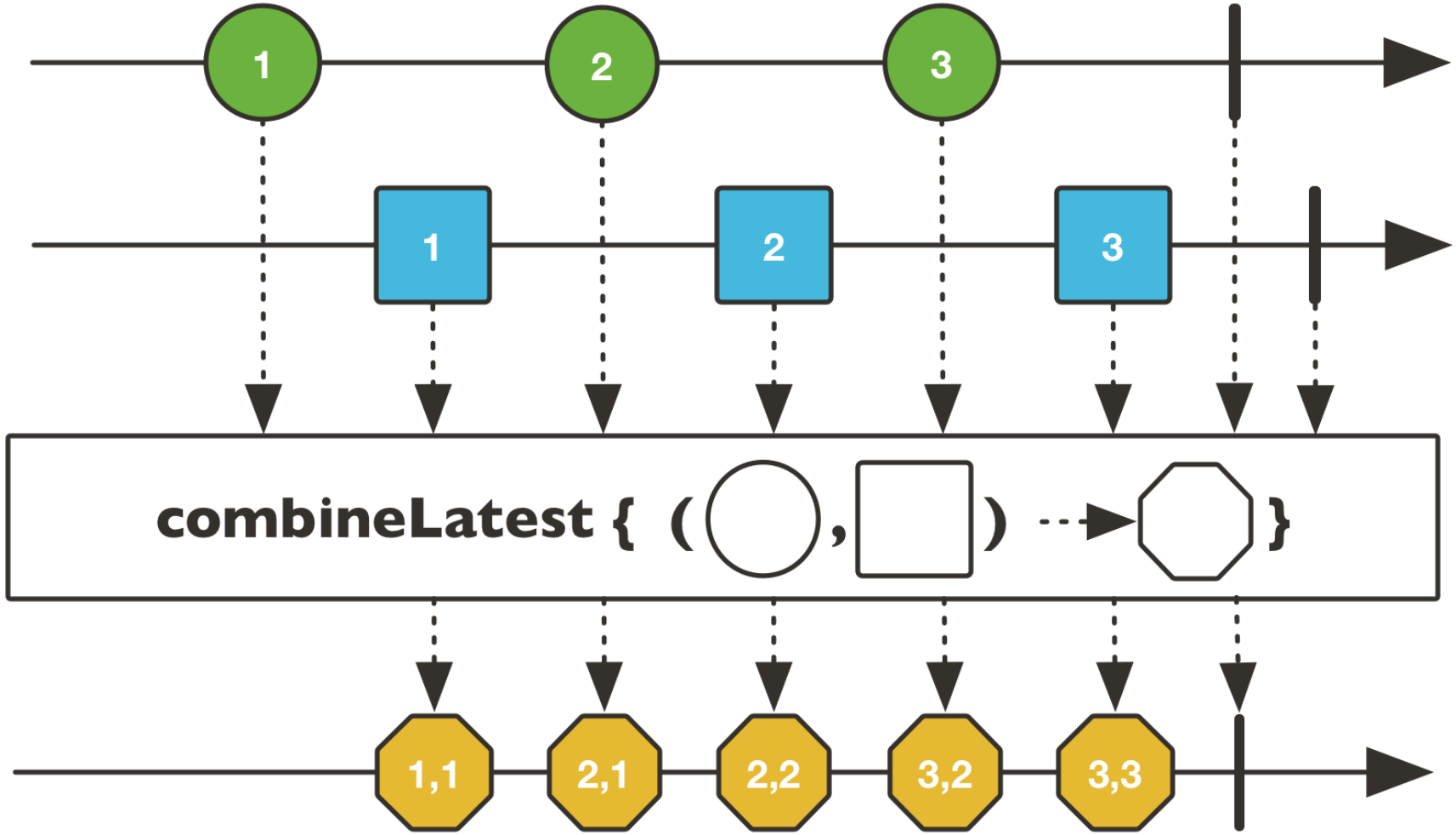


By ReactiveX - Creative Commons Attribution 3.0 License, Source: http://reactivex.io/documentation/observable.html

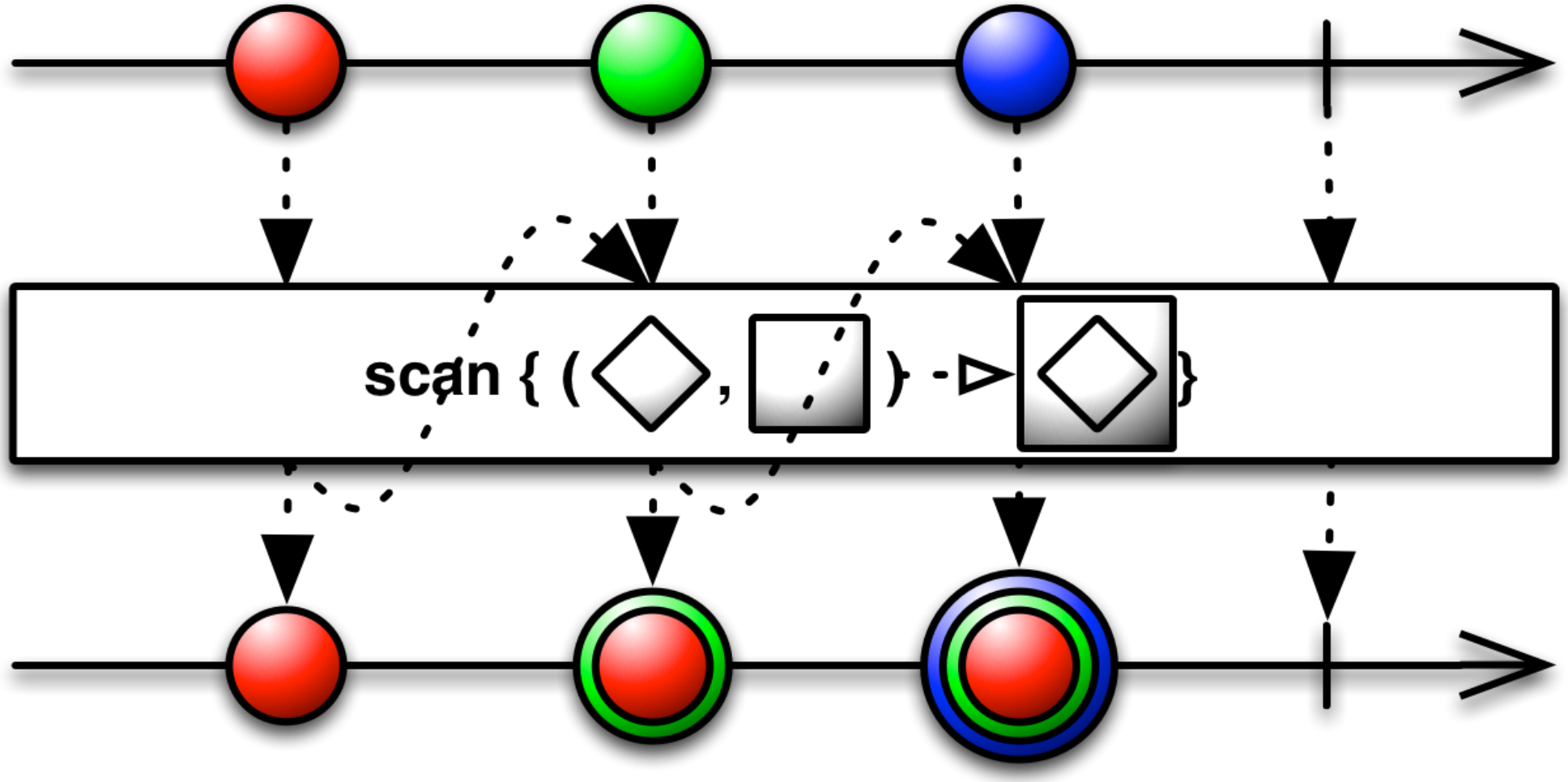
Example: Zip



Example: CombineLatest



Redux == Rx Scan Opearator



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

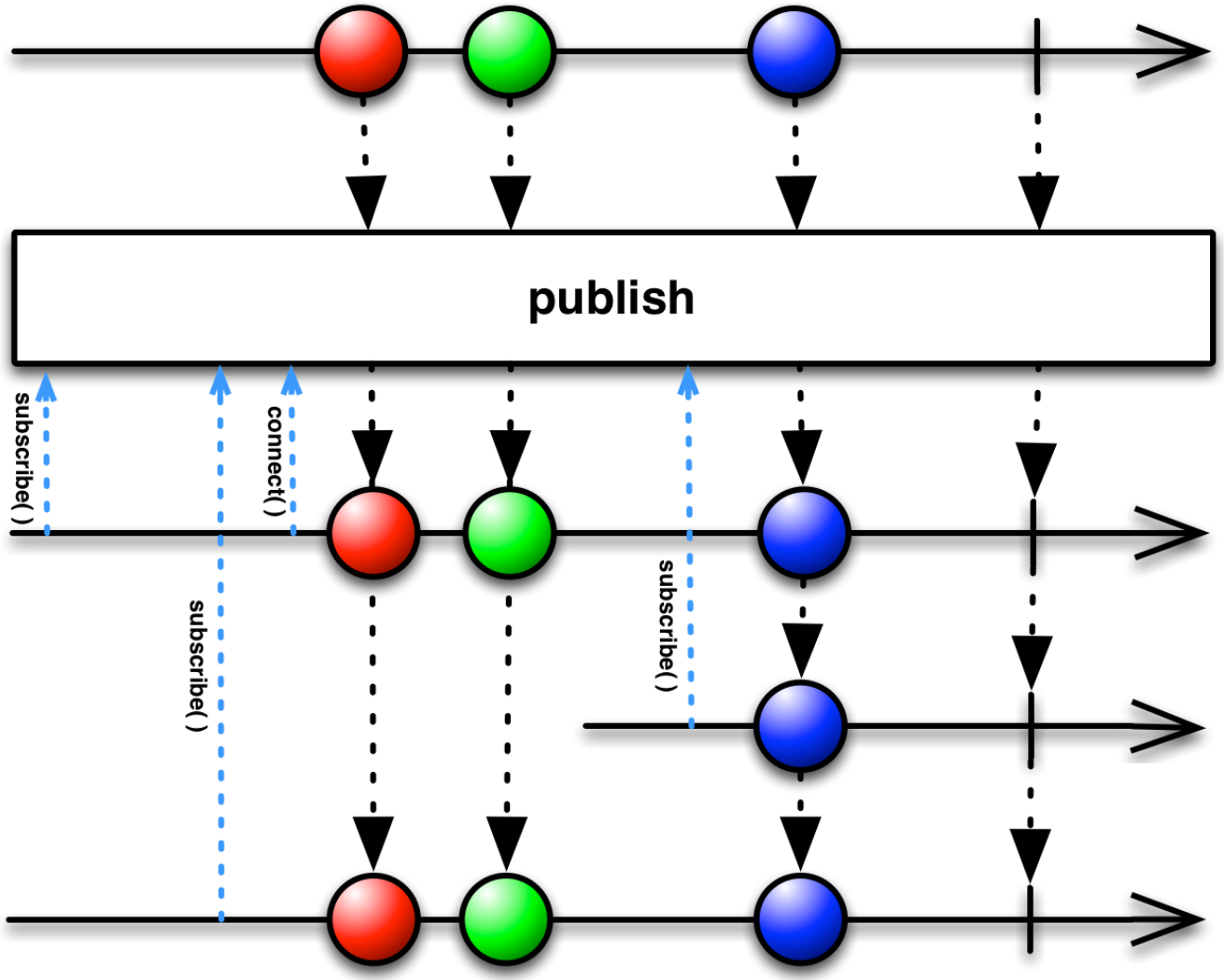
Hot and Cold Event Streams

- 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

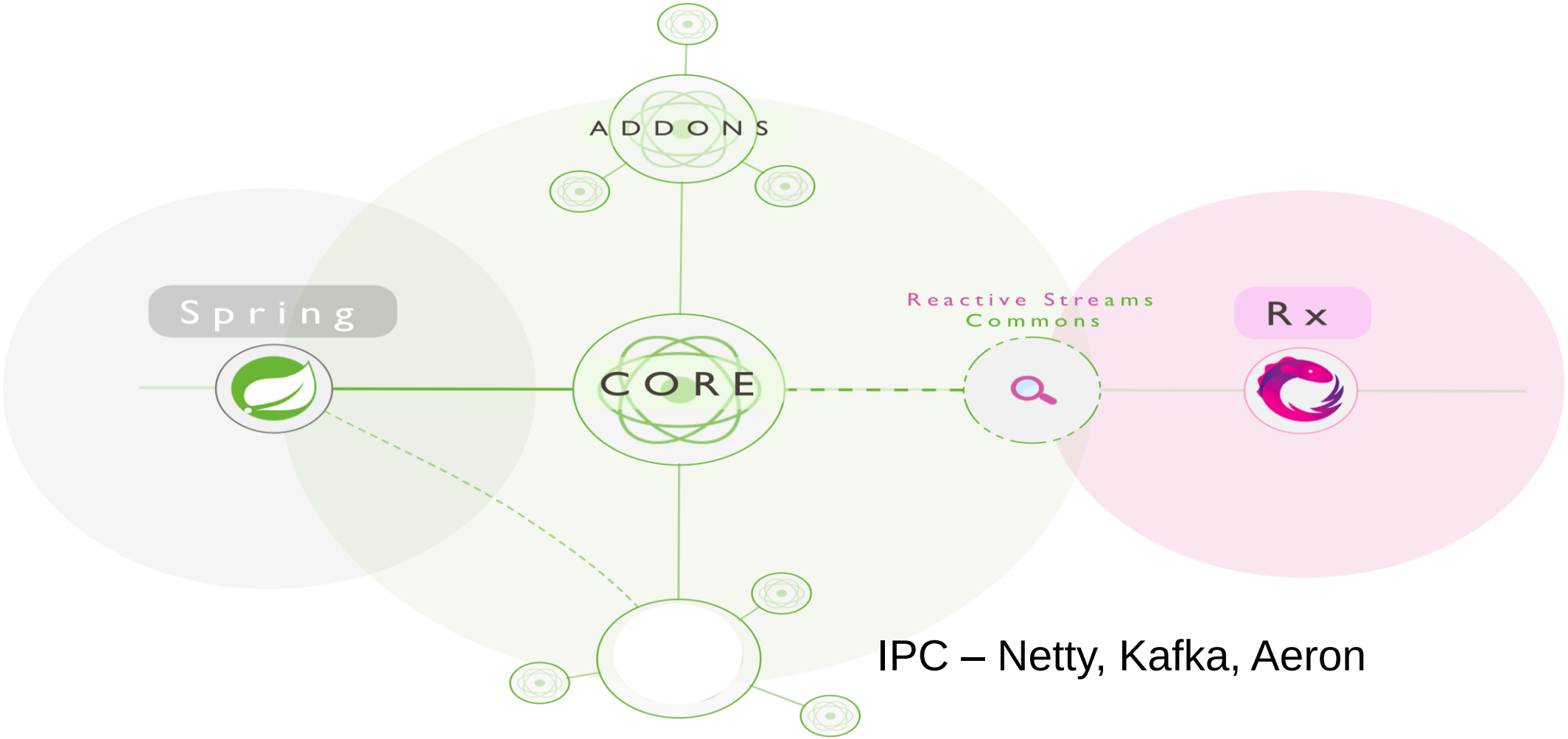


Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

Project Reactor

- Reactor project allows building high-performance (low latency high throughput) non-blocking asynchronous applications on JVM.
- Reactor is designed to be extraordinarily fast and can sustain throughput rates on order of 10's of millions of operations per second.
- Reactor has powerful API for declaring data transformations and functional composition.
- Makes use of the concept of Mechanical Sympathy built on top of Disruptor / RingBuffer.

Reactor Projects



IPC – Netty, Kafka, Aeron

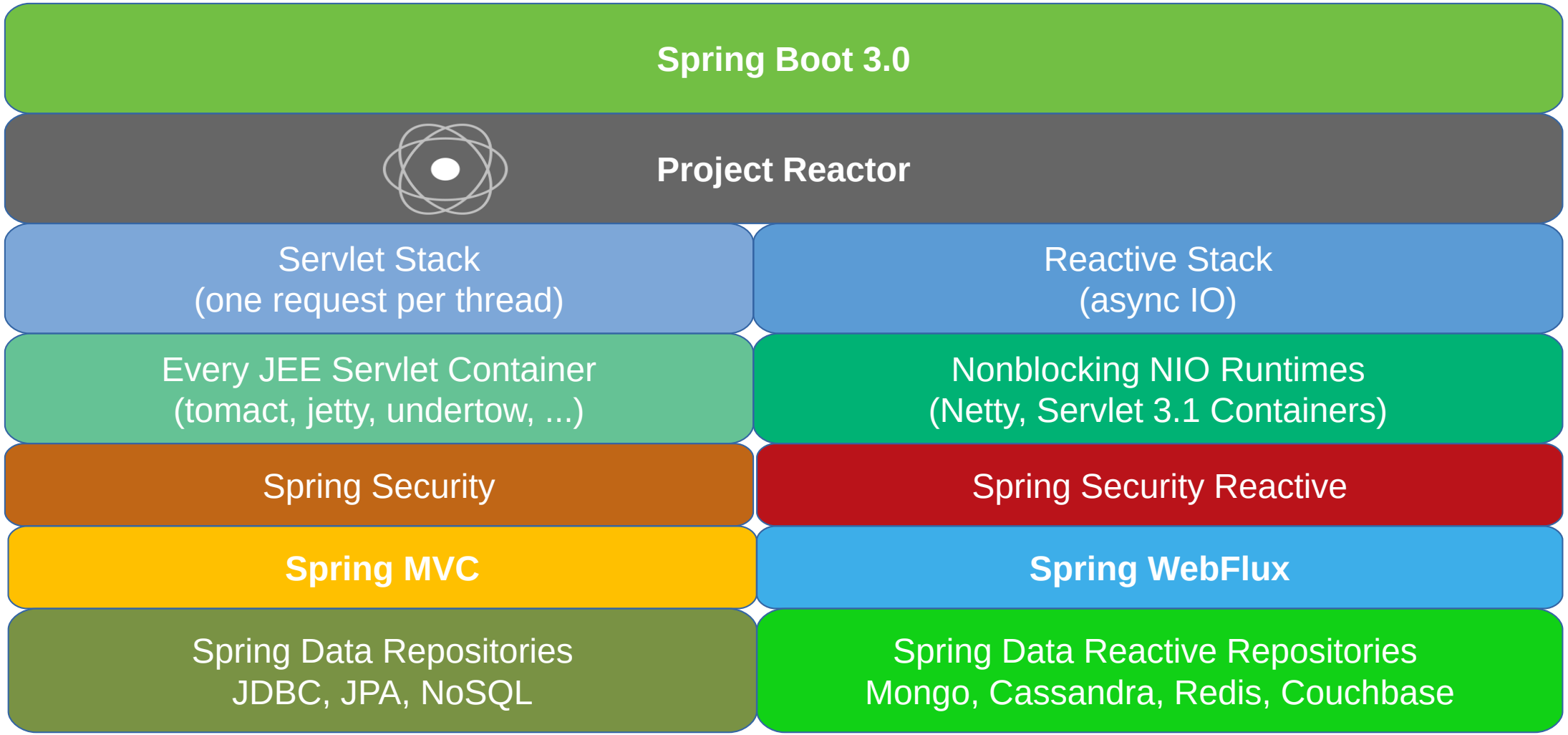
Hot Stream Example - Reactor

```
EmitterProcessor<String> emitter =
        EmitterProcessor.create();
FluxSink<String> sink = emitter.sink();
emitter.publishOn(Schedulers.single())
        .map(String::toUpperCase)
        .filter(s -> s.startsWith("HELLO"))
        .delayElements(Duration.of(1000, MILLIS))
.subscribe(System.out::println);
sink.next("Hello World!"); // emit - non blocking
sink.next("Goodbye World!");
sink.next("Hello Trayan!");
Thread.sleep(3000);
```

Top New Features in Spring 5

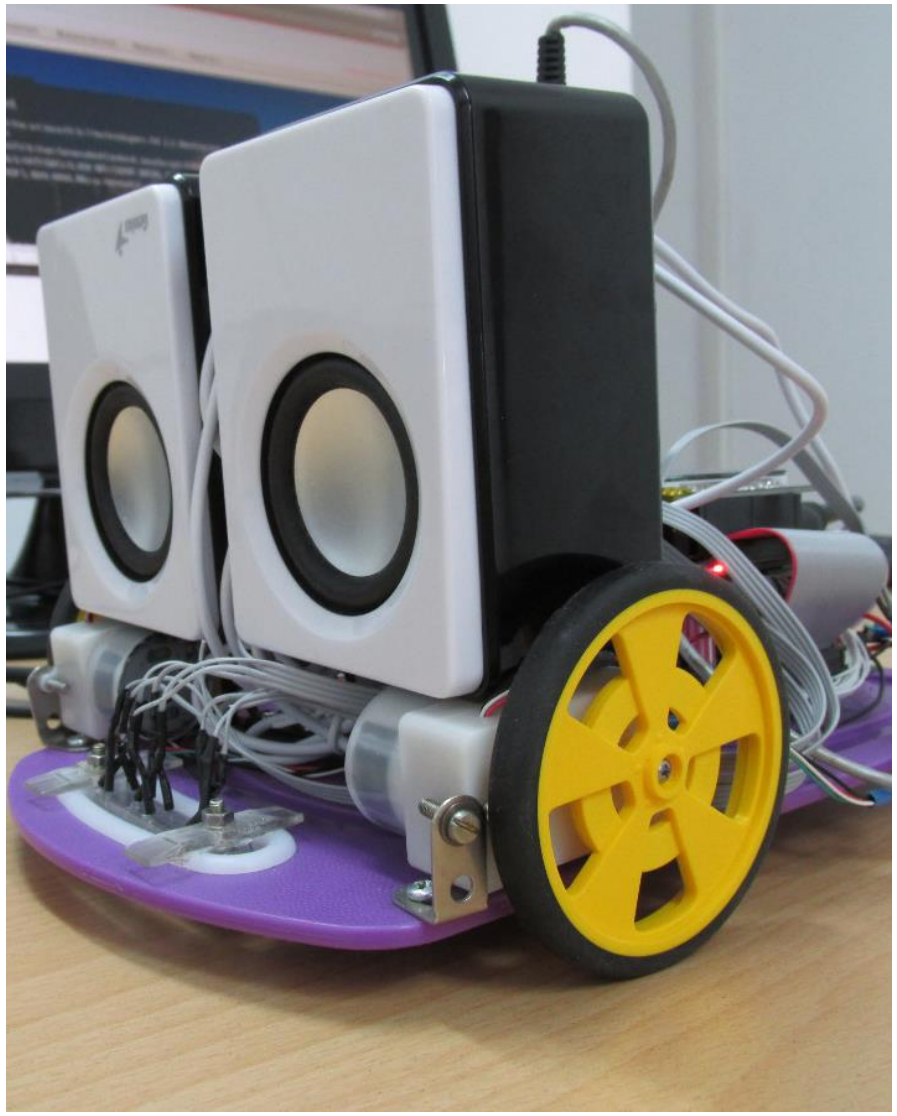
- Reactive Programming Model
- Spring Web Flux
- Reactive DB repositories & integrations + hot event streaming: MongoDB, CouchDB, Redis, Cassandra, Kafka
- JDK 8+ and Java EE 7+ baseline
- Testing improvements – WebTestClient (based on reactive WebFlux WebClient)
- Kotlin functional DSL

Spring 6 Main Building Blocks



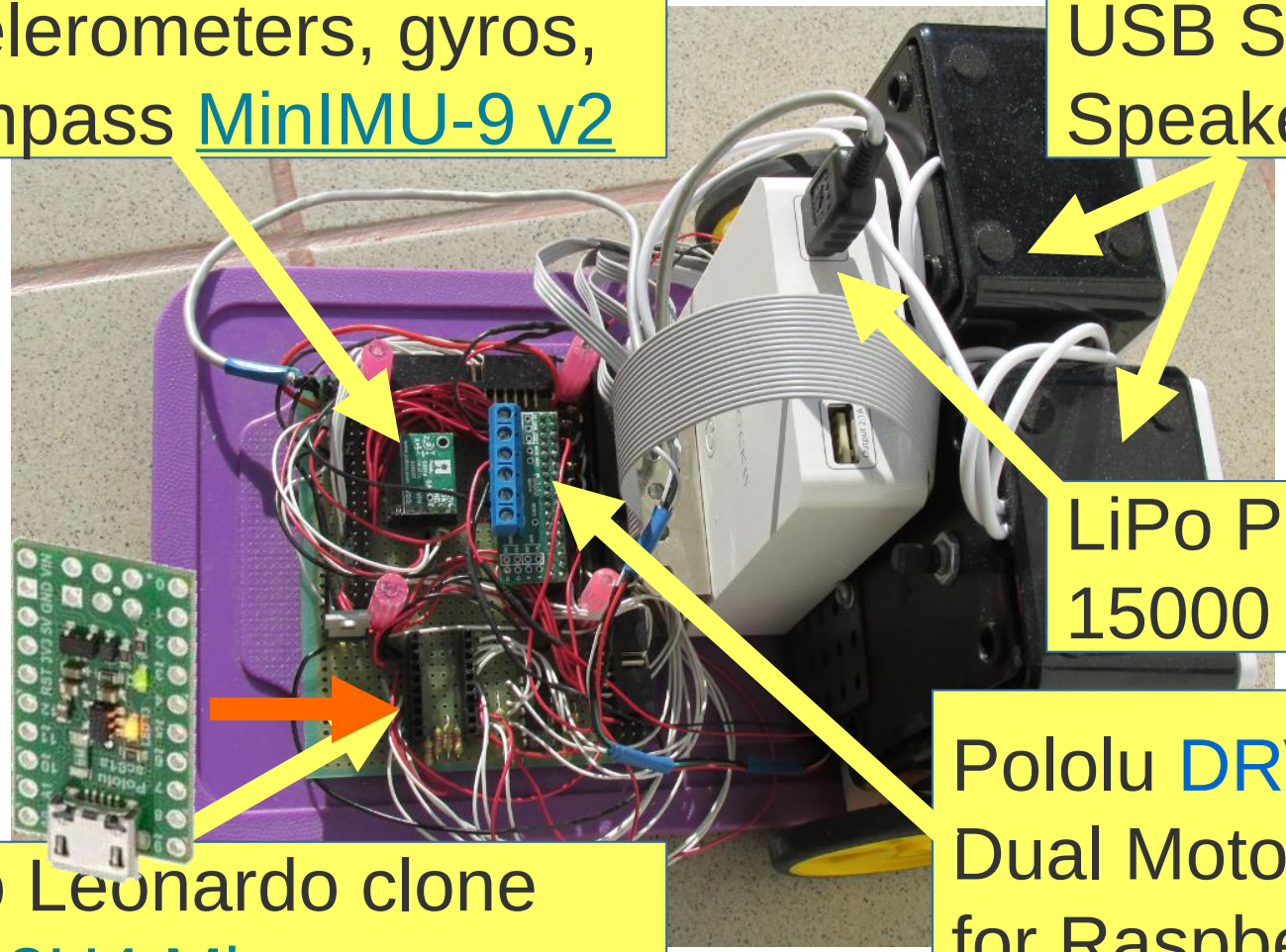
Source: https://spring.io

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



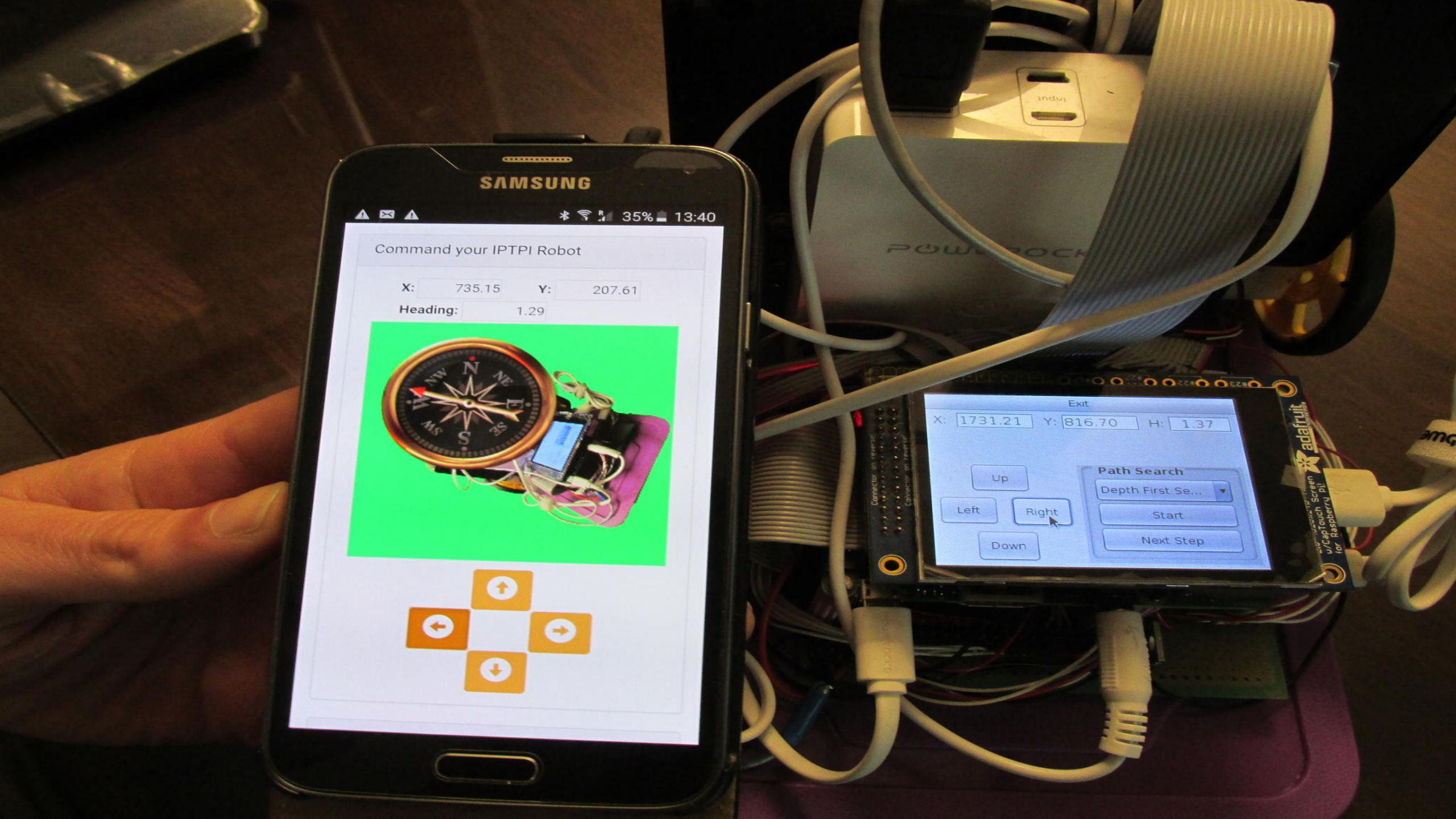
3D accelerometers, gyros, and compass MinIMU-9 v2

USB Stereo Speakers - 5V

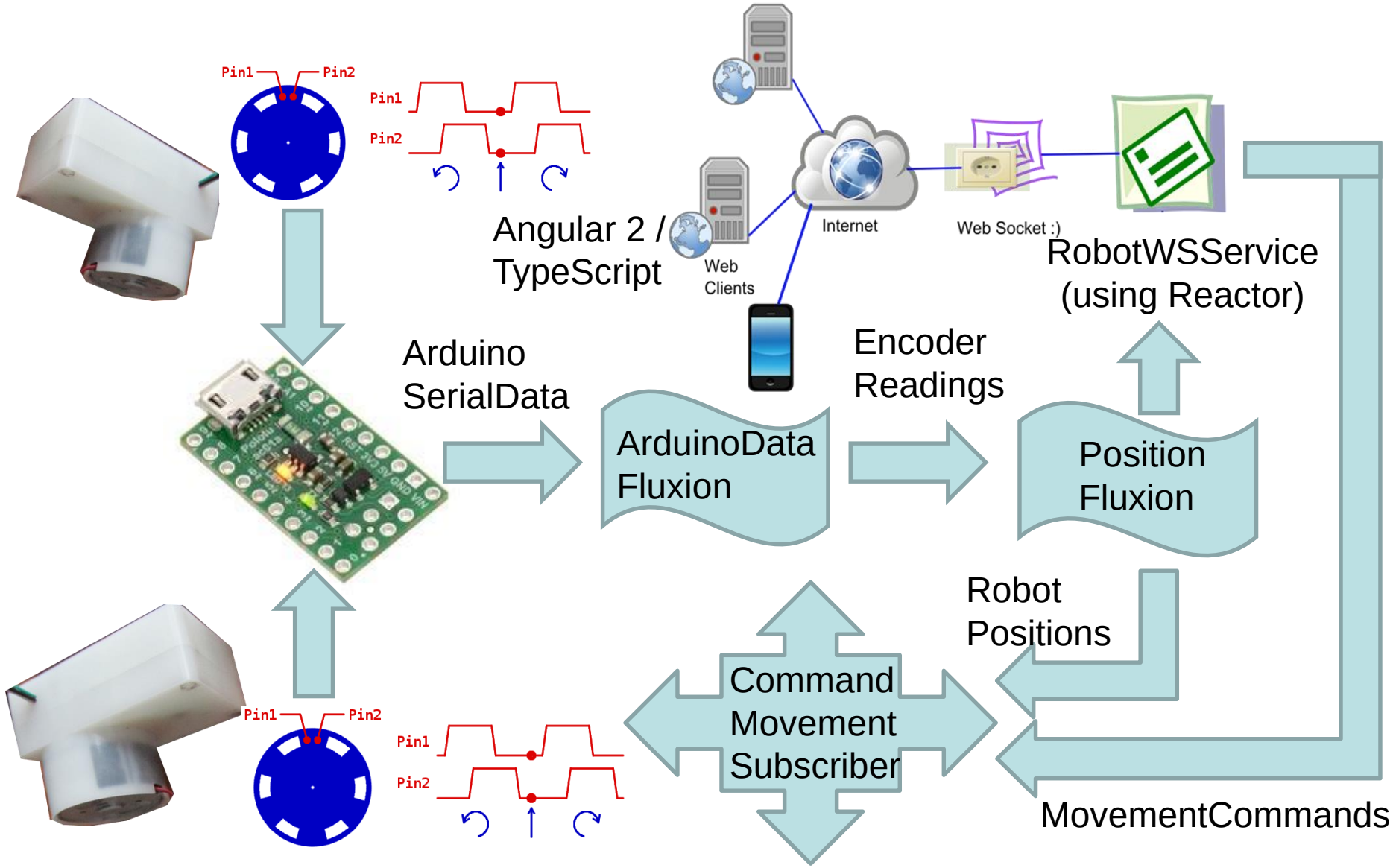
LiPo Powebank 15000 mAh

Pololu DRV8835 Dual Motor Driver for Raspberry Pi

Arduino Leonardo clone A-Star 32U4 Micro

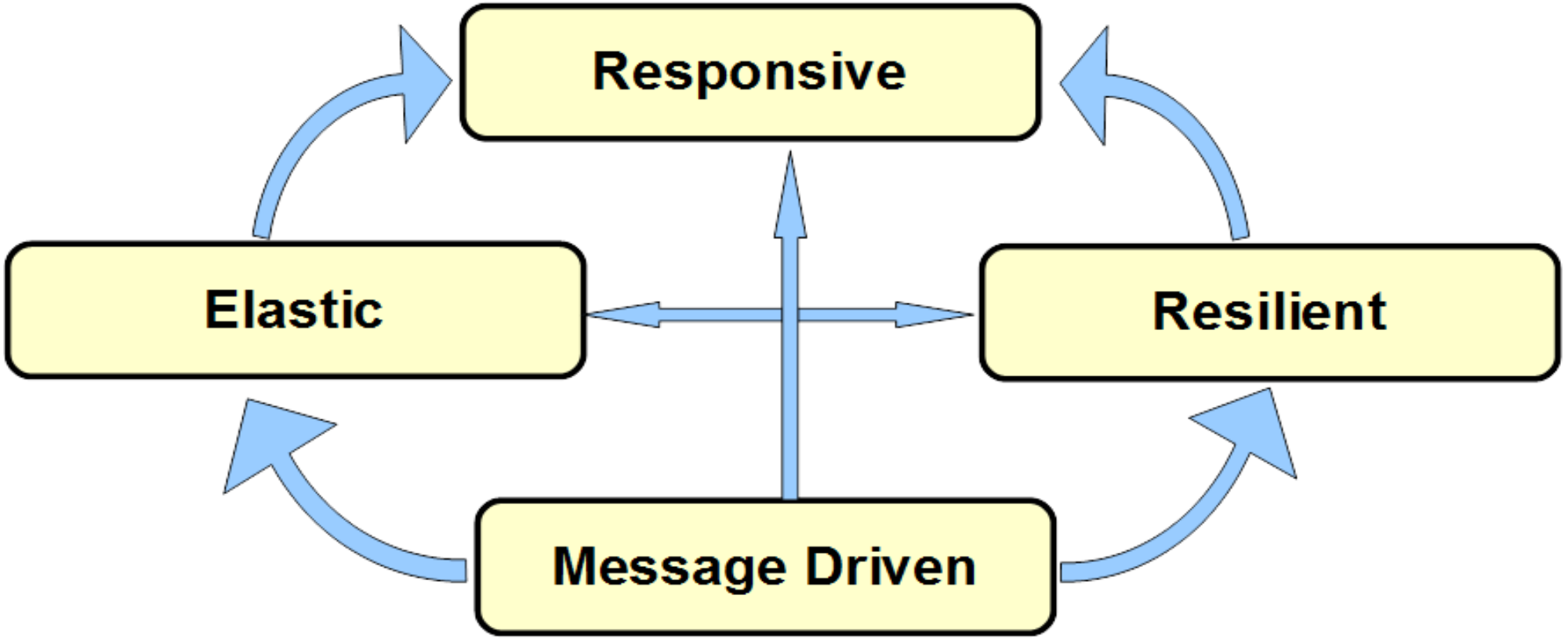

SAMSUNG
35% 13:40
Command your IPTPI Robot
X: 735.15
Y: 207.61
Heading: 1.29
Exit
X: 1731.21
Y: 816.70
H: 1.37
Up
Left
Right
Down
Path Search
Depth First Se...
Start
Next Step

IPTPI Reactive Streams



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**

What's High Performance?

- **Performance** is about 2 things (Martin Thompson – http://www.infoq.com/articles/low-latency-vp):
  - **Throughput** – units per second, and
  - **Latency** – response time
- **Real-time** – time constraint from input to response regardless of system load.
- **Hard real-time system** if this constraint is not honored then a total system failure can occur.
- **Soft real-time system** – low latency response with little deviation in response time
- **100 nano-seconds** to **100 milli-seconds**. [Peter Lawrey]

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

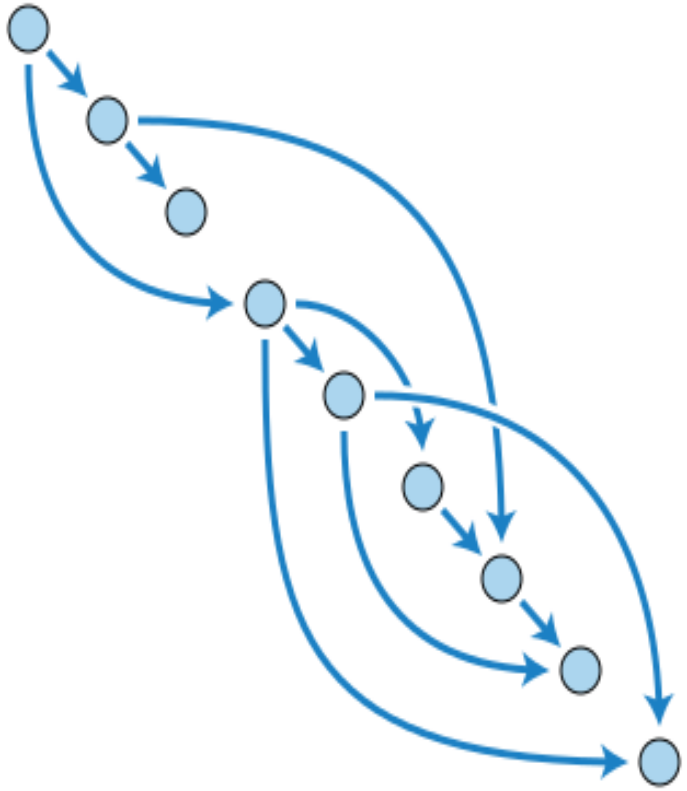
Data Stream Programming

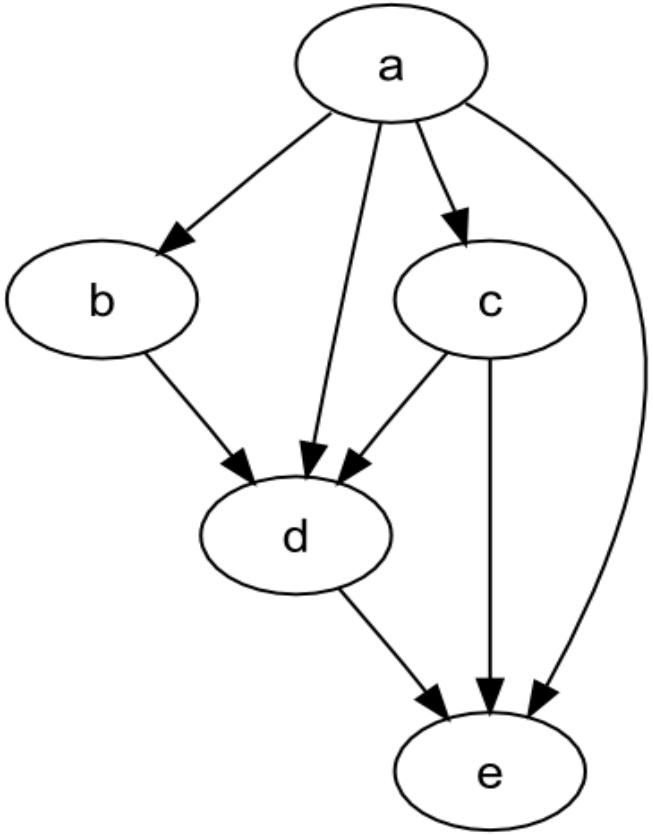
The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

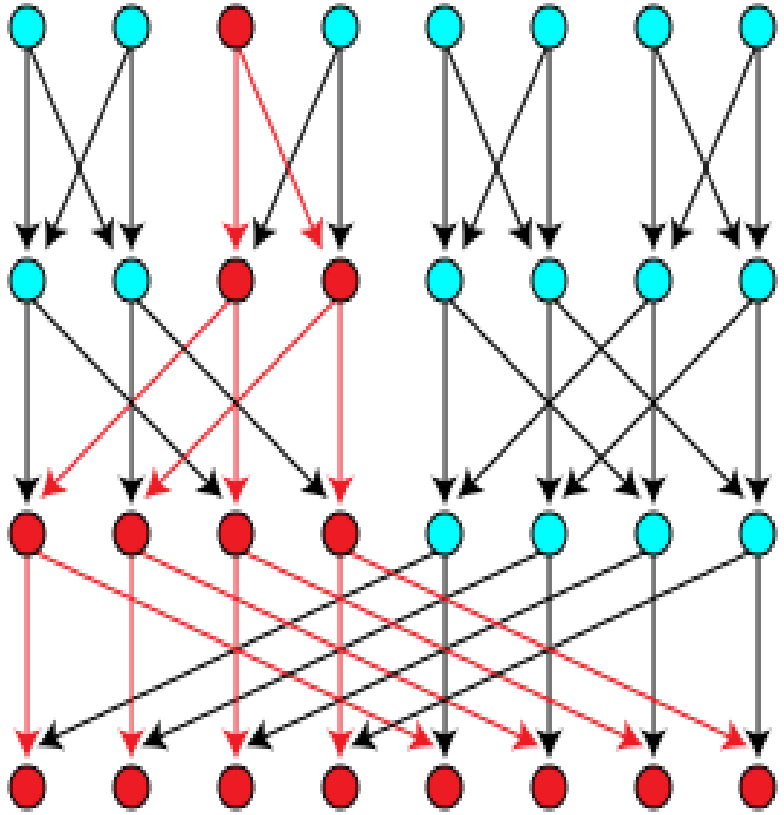
For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

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

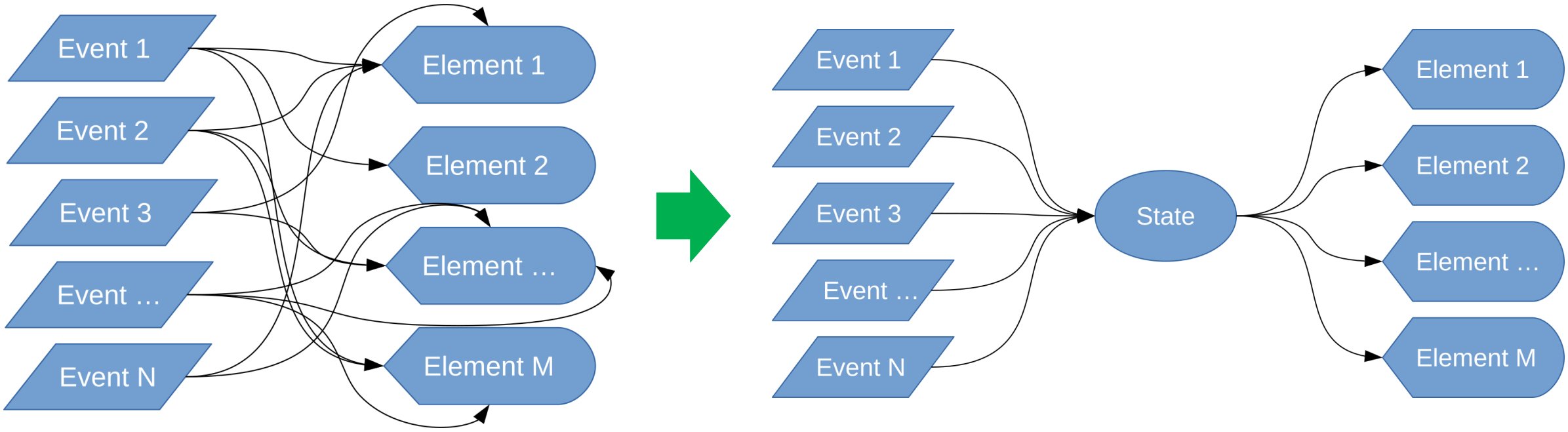
Direct Acyclic Graphs - DAG





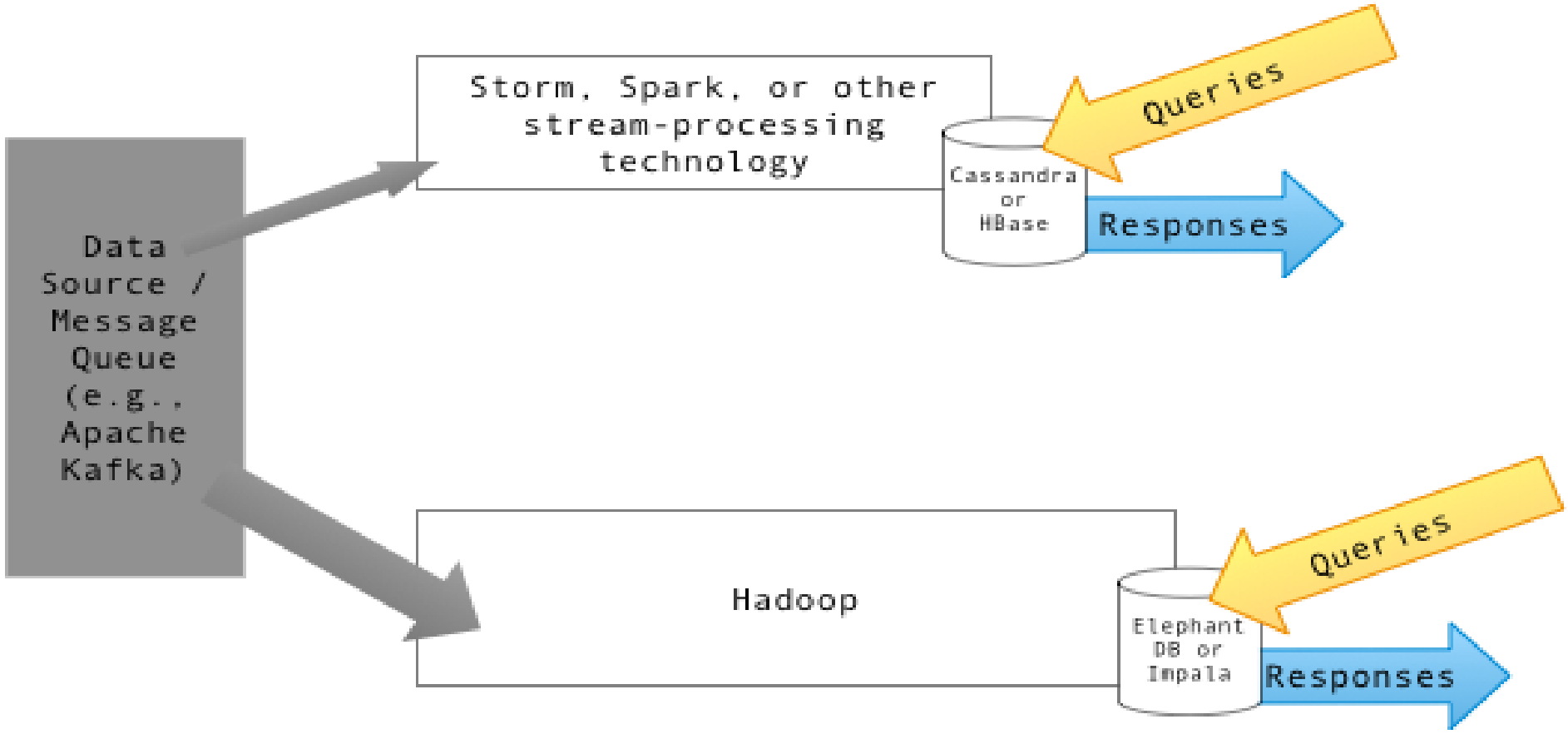


Event Sourcing – Events vs. Sate (Snapshots)



Lambda Architecture - I

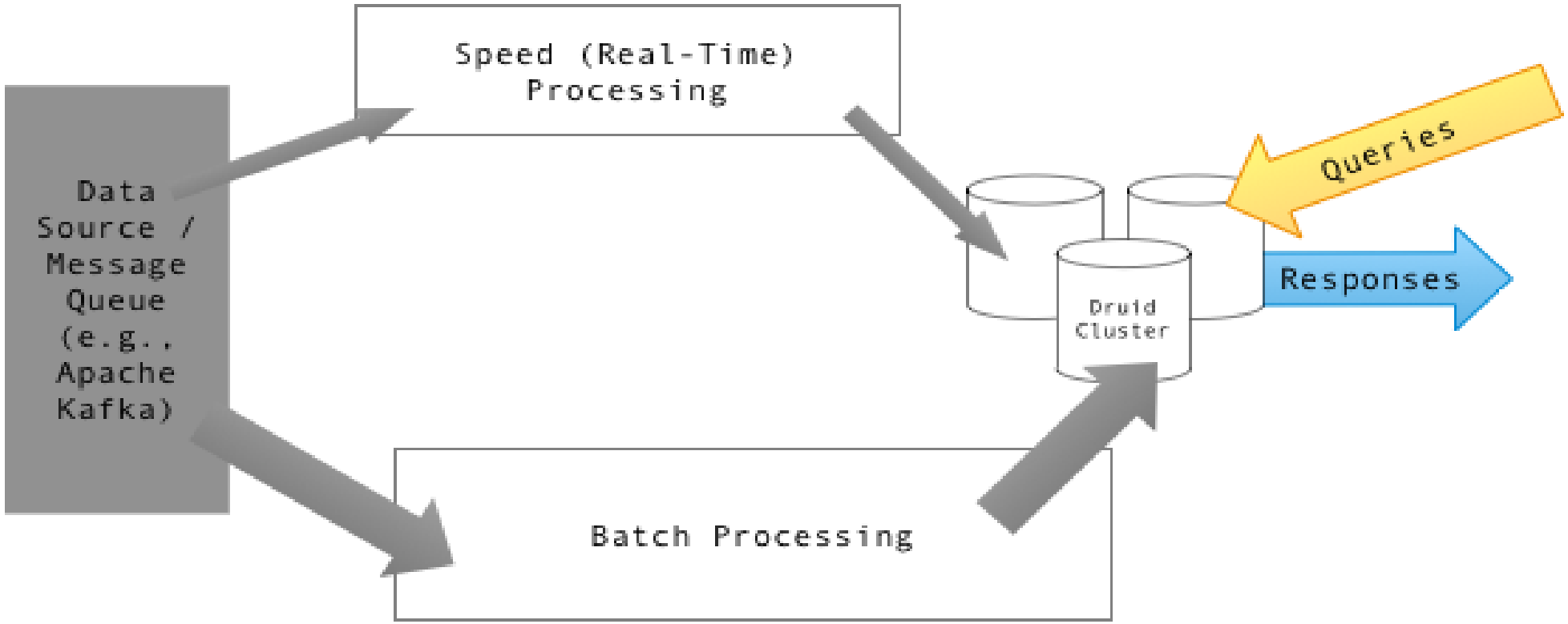
Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



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

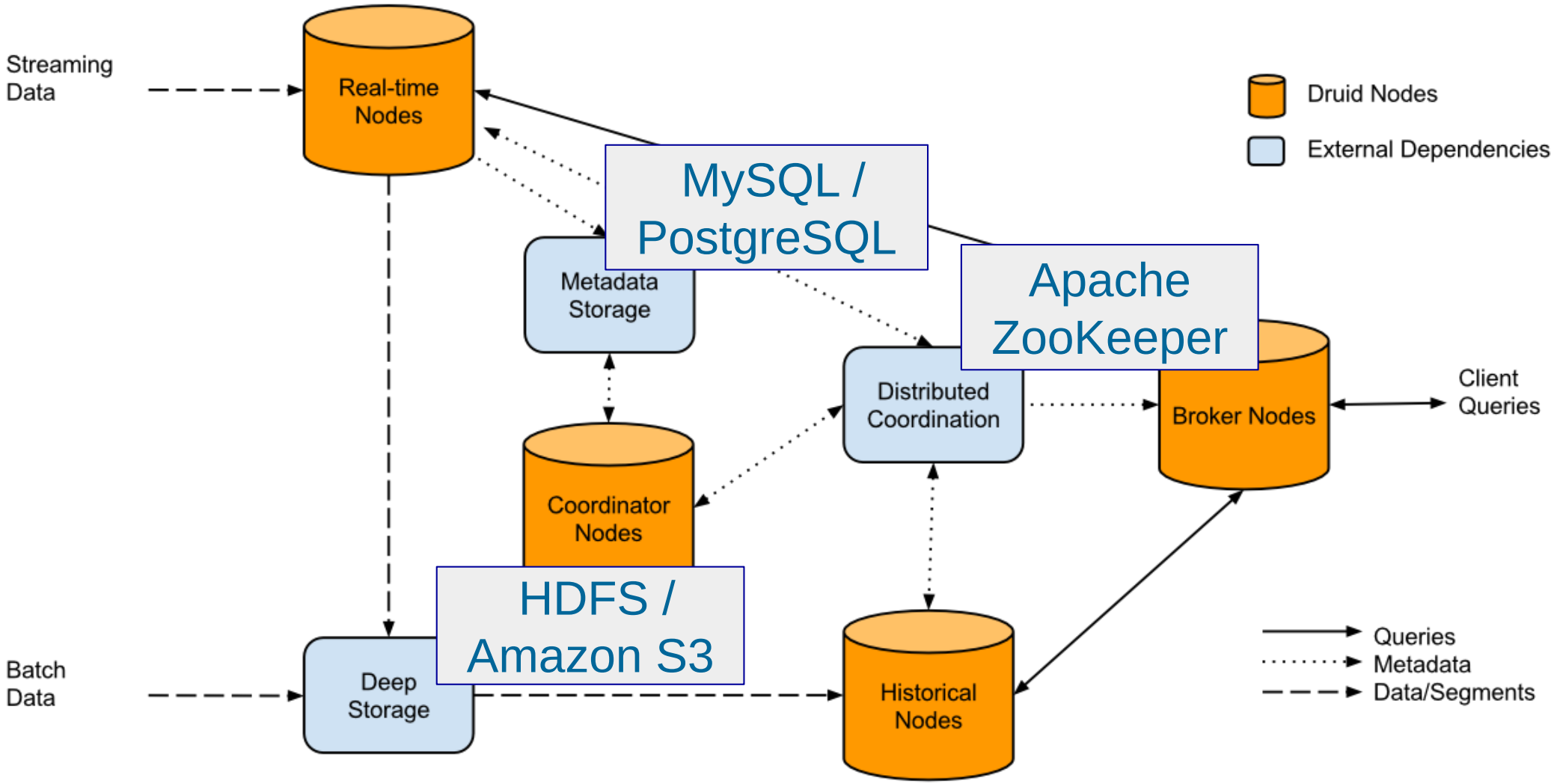
Lambda Architecture - II

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



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

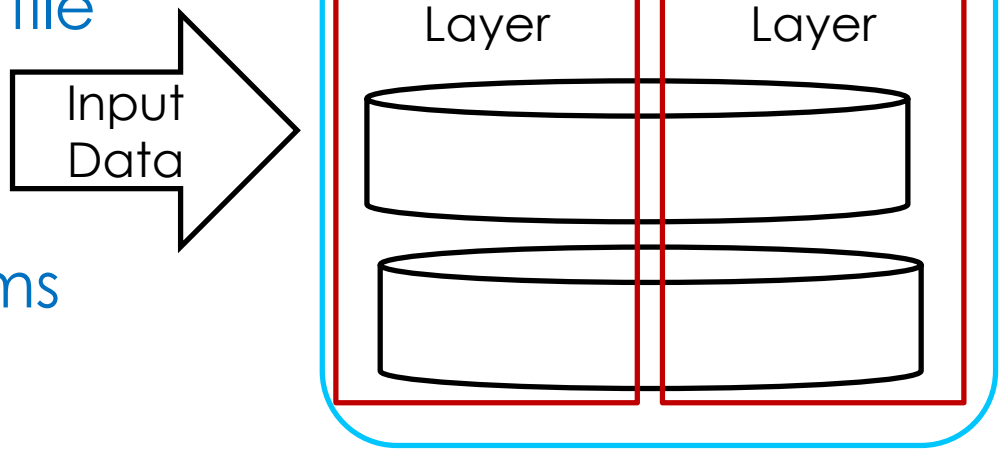
Query = K (New Data) = K (Live streaming data)

- 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:

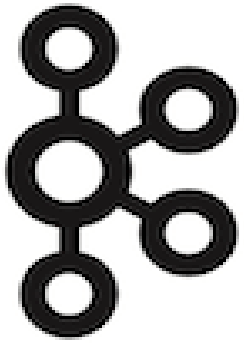
- Apache Spark is an open-source cluster-computing framework. Spark Streaming, Spark Mllib -> Spark ML

- Apache Storm is a distributed stream processing – streams DAG

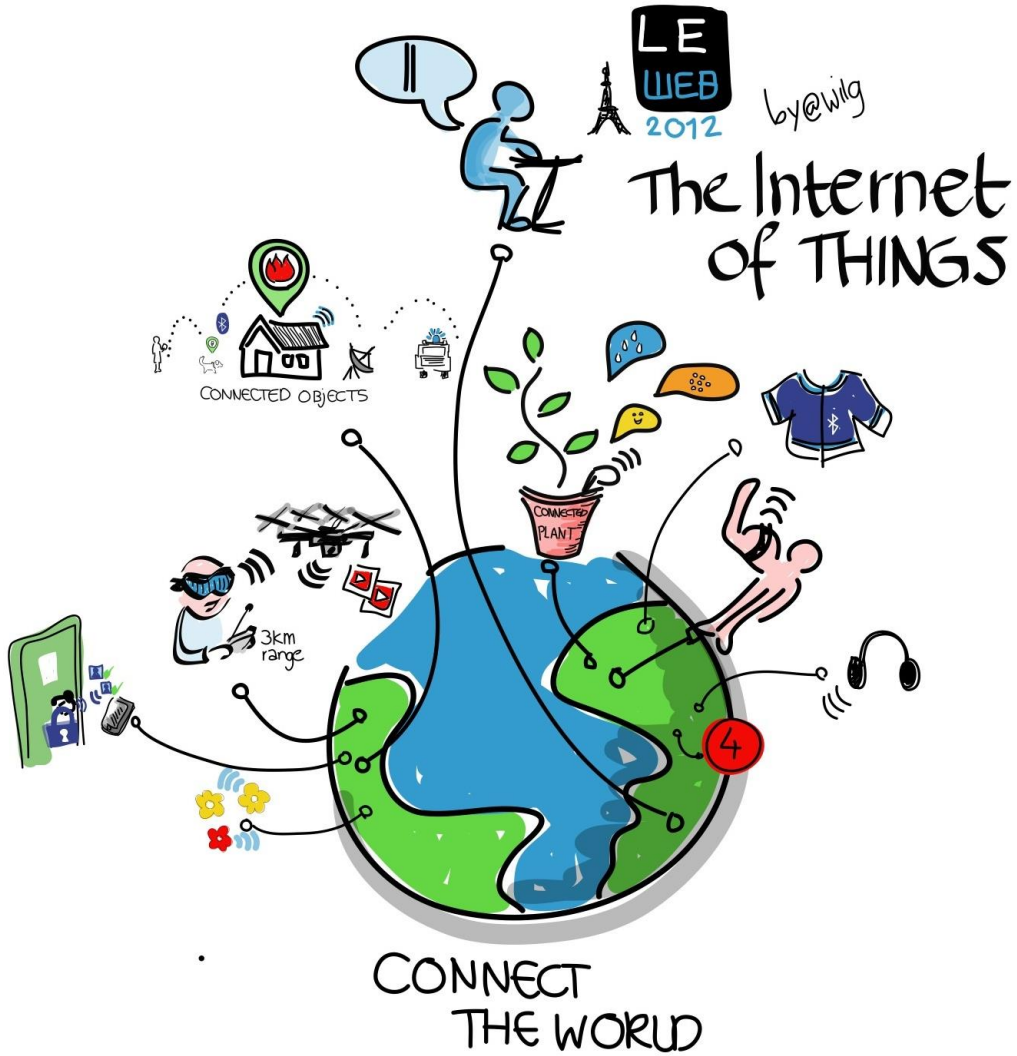
- Apache Samza is a distributed real-time stream processing framework.

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

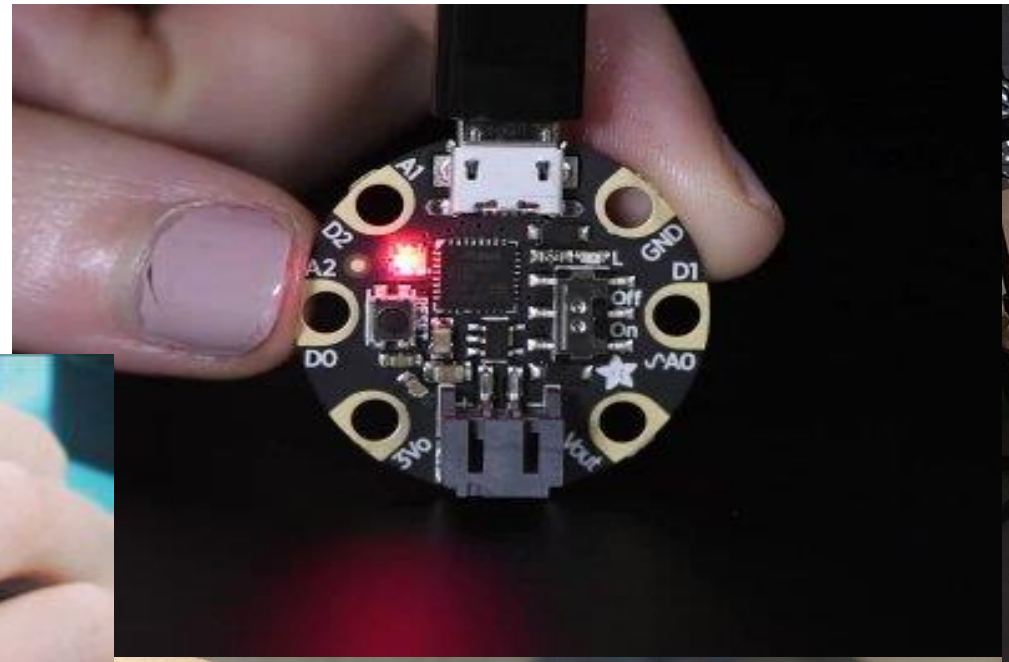


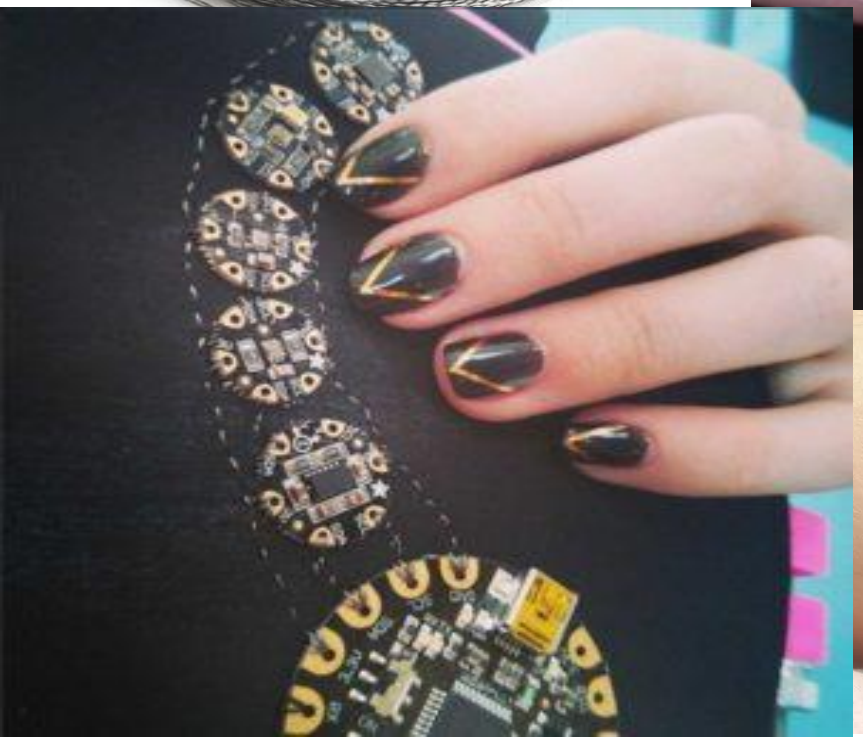
Example: Internet of Things (IoT)



CC BY 2.0, Source: https://www.flickr.com/photos/wilgengebroed/8249565455/ ,
Radar, GPS, lidar for navigation and obstacle avoidance (2007 DARPA Urban Challenge)

Wearable Electronics :)





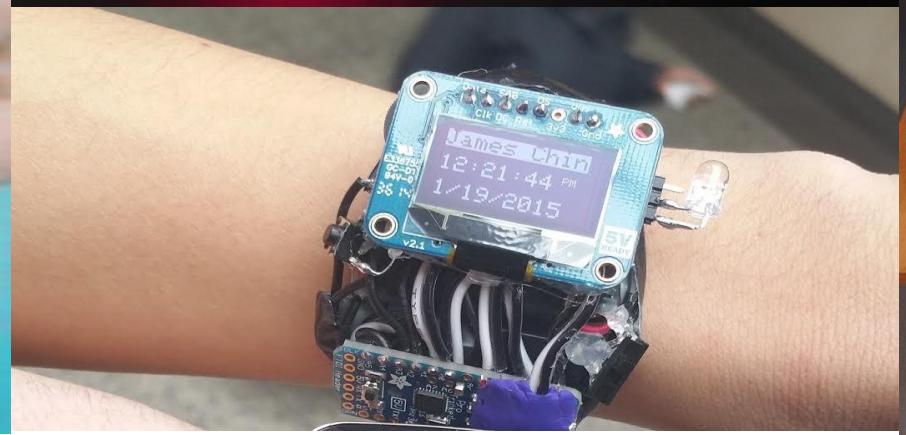
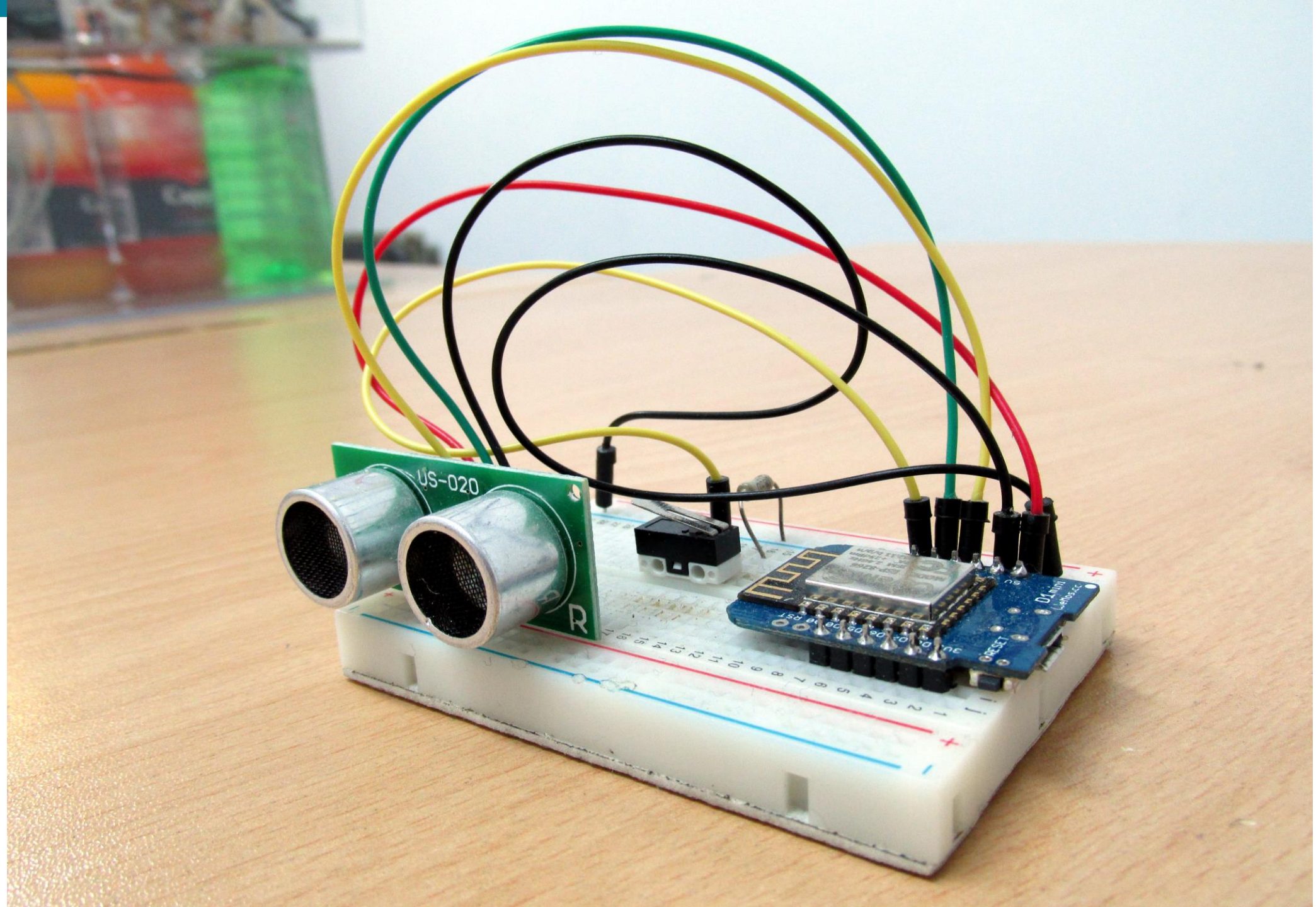
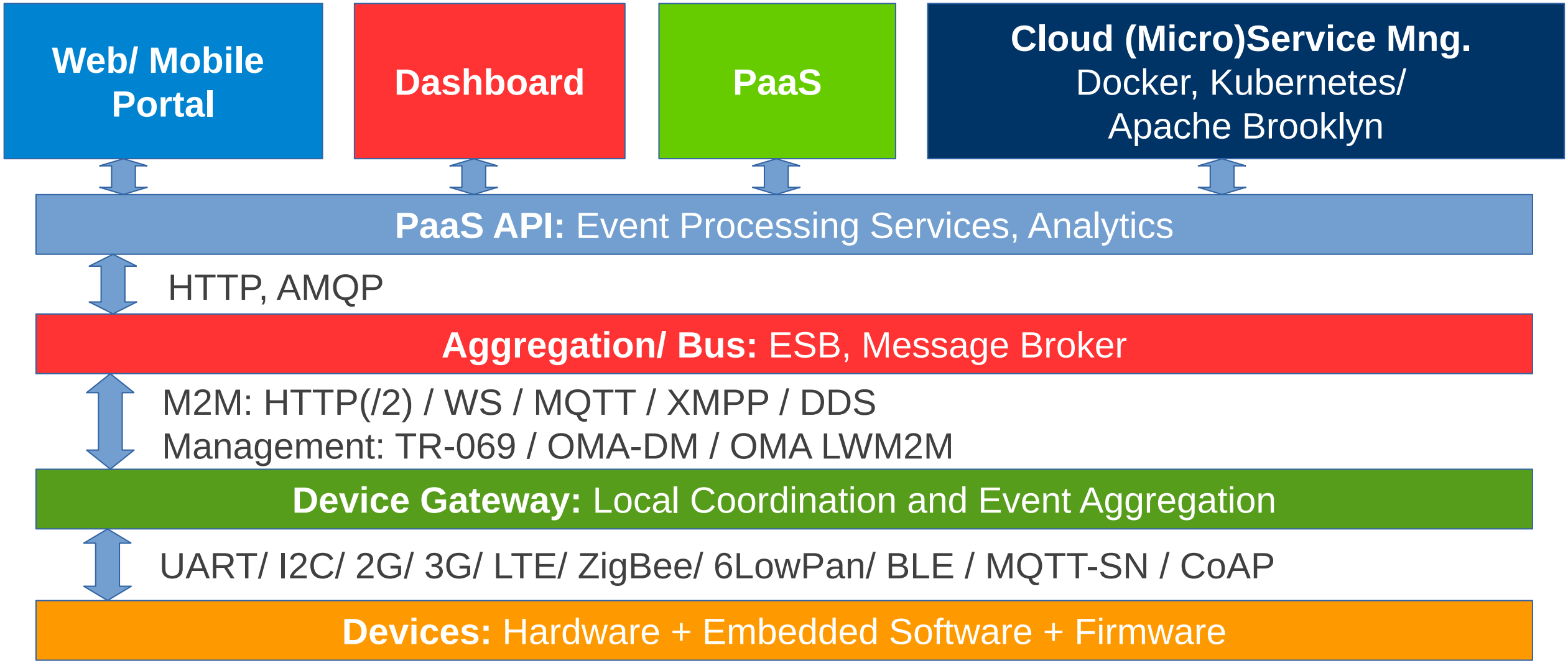


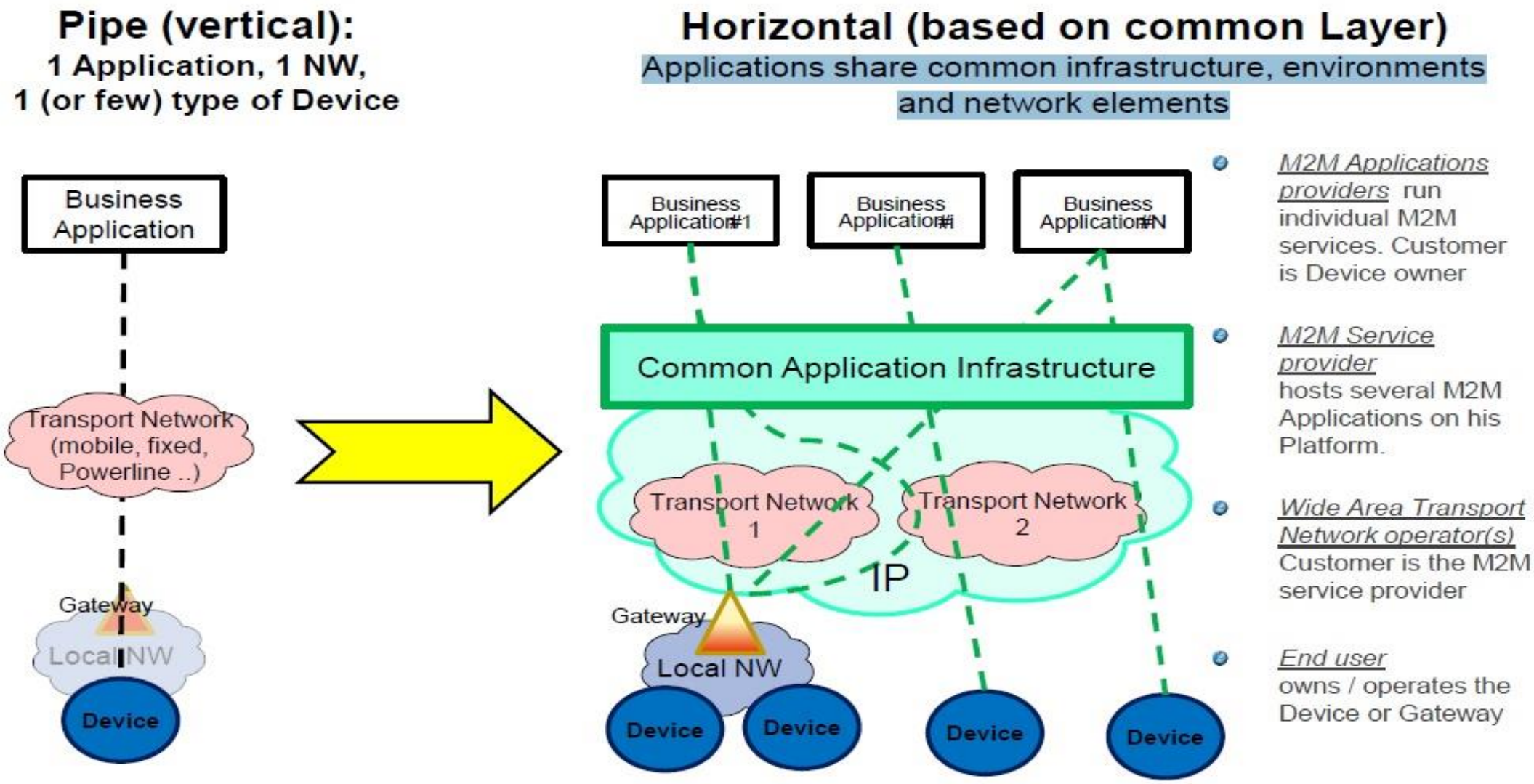
Photo credit: Adafruit, Becky Stern and others


US-020
R
D1 mini
GRESET

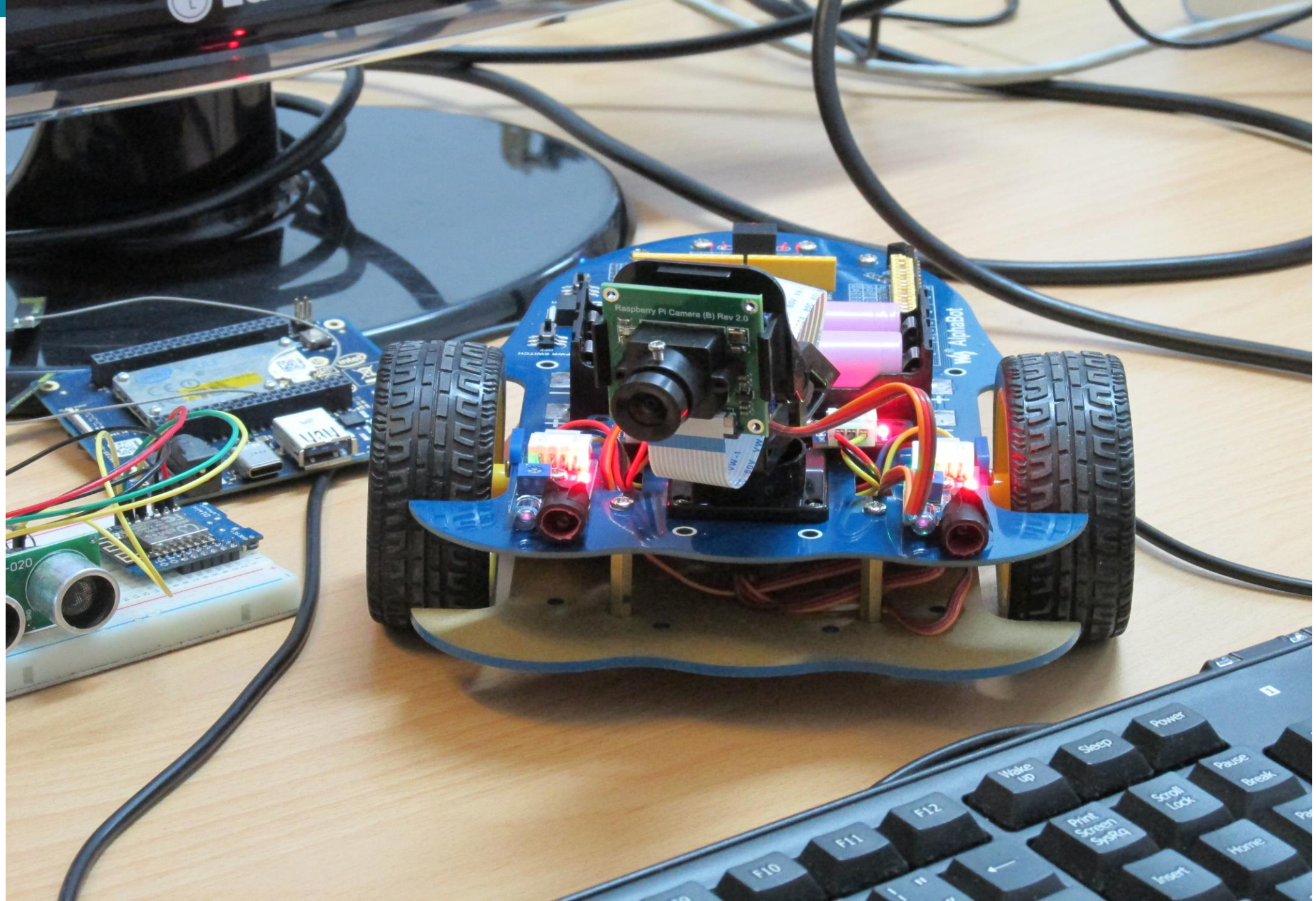
IoT Services Architecture

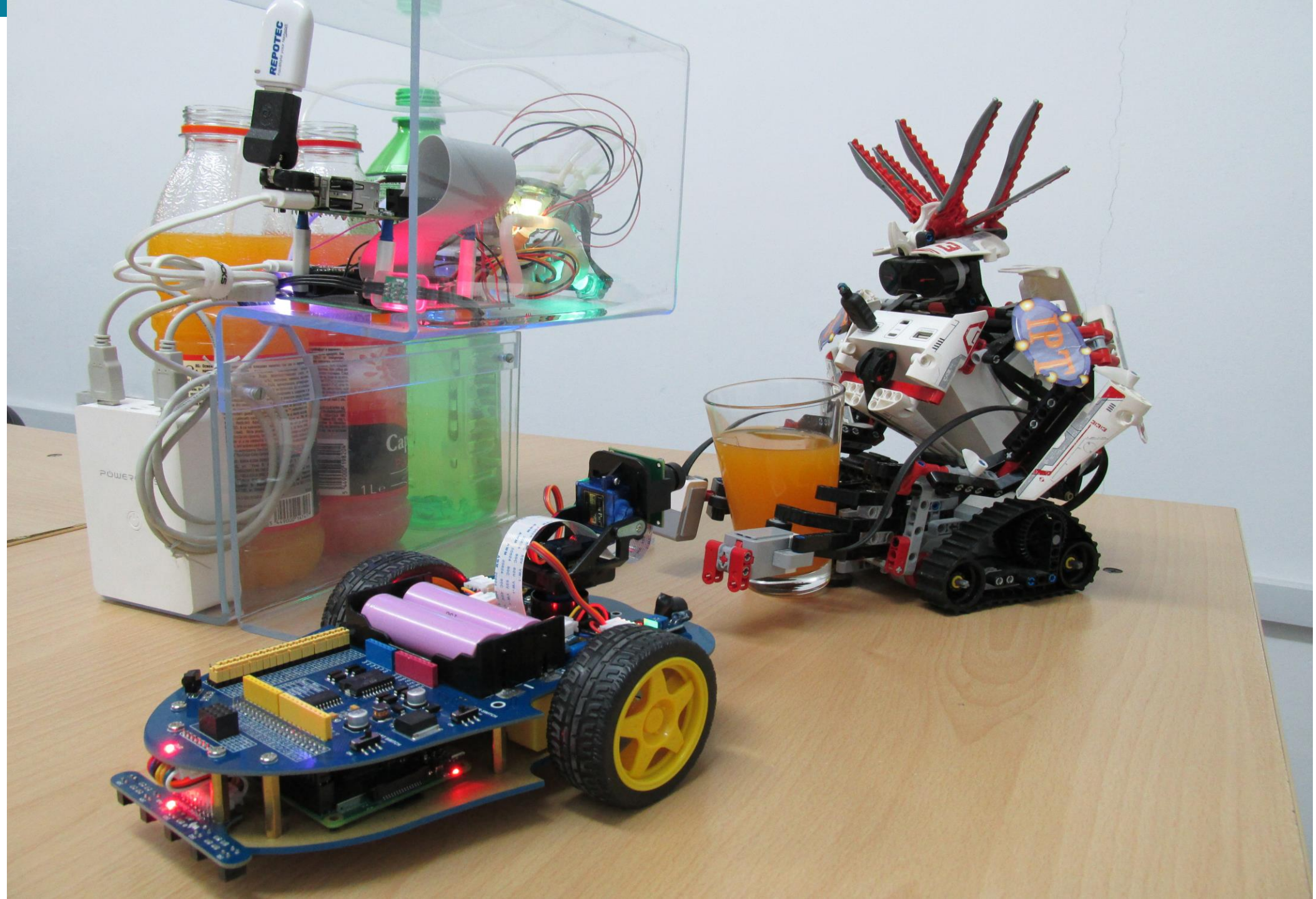


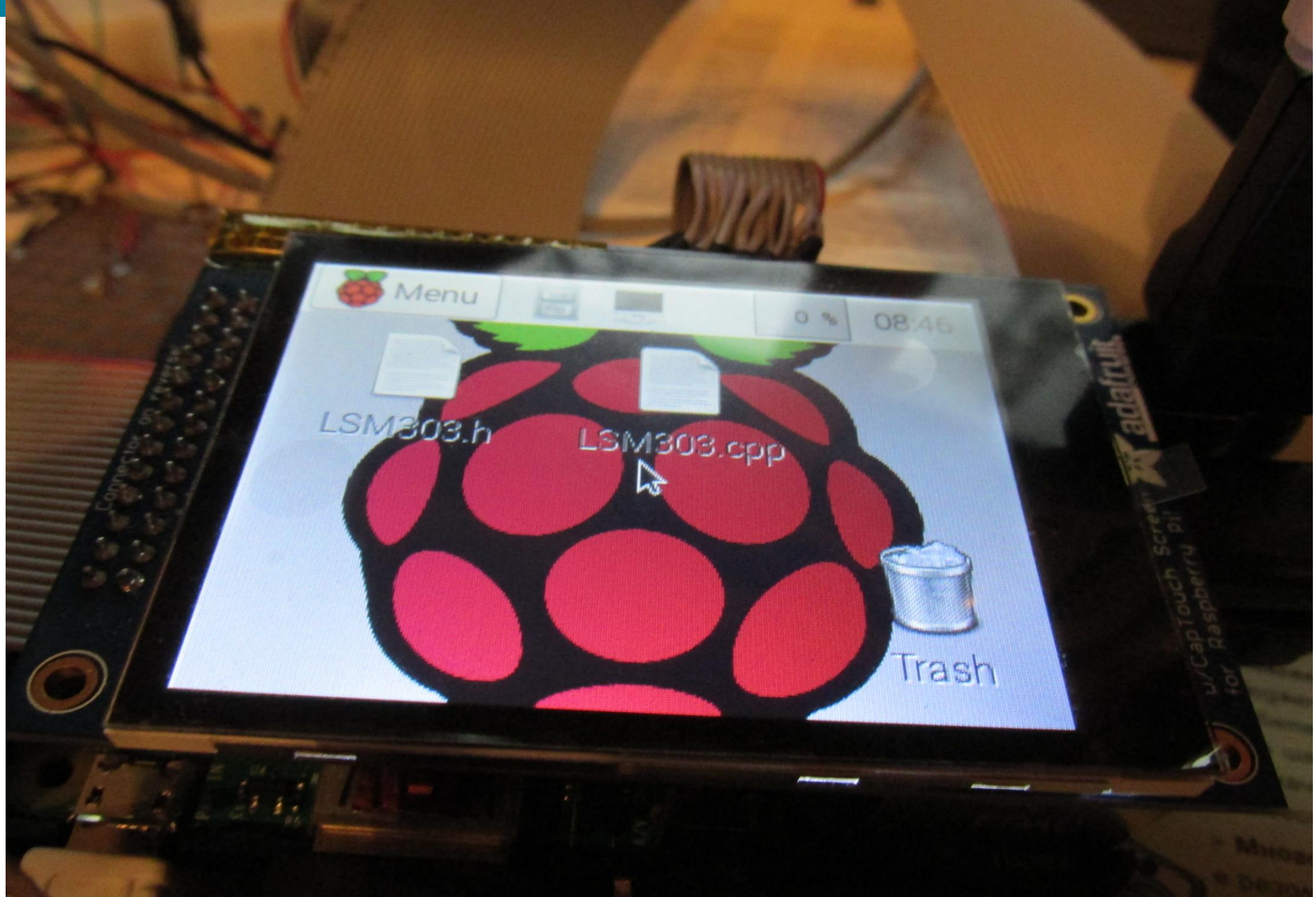
Vertical vs. Horizontal IoT



http://www.iot-week.eu/overview_2013/iot-week-2012/programme-1/tuesday-1/presentations/semantic-interoperability/K1_ETSI-M2M-oneM2M-and-the-need-for-semantics-IoTWeek2012.pdf, By Joerg Swetina (NEC)- ETSI M2M / oneM2M and the need for semantics


Raspberry Pi Camera (B) Rev 2.0
AlphaBot
Power
Sleep
Wake up
F12
F11
F10
Scroll Lock
Pause Break
Print Screen SysRq
Home
Insert

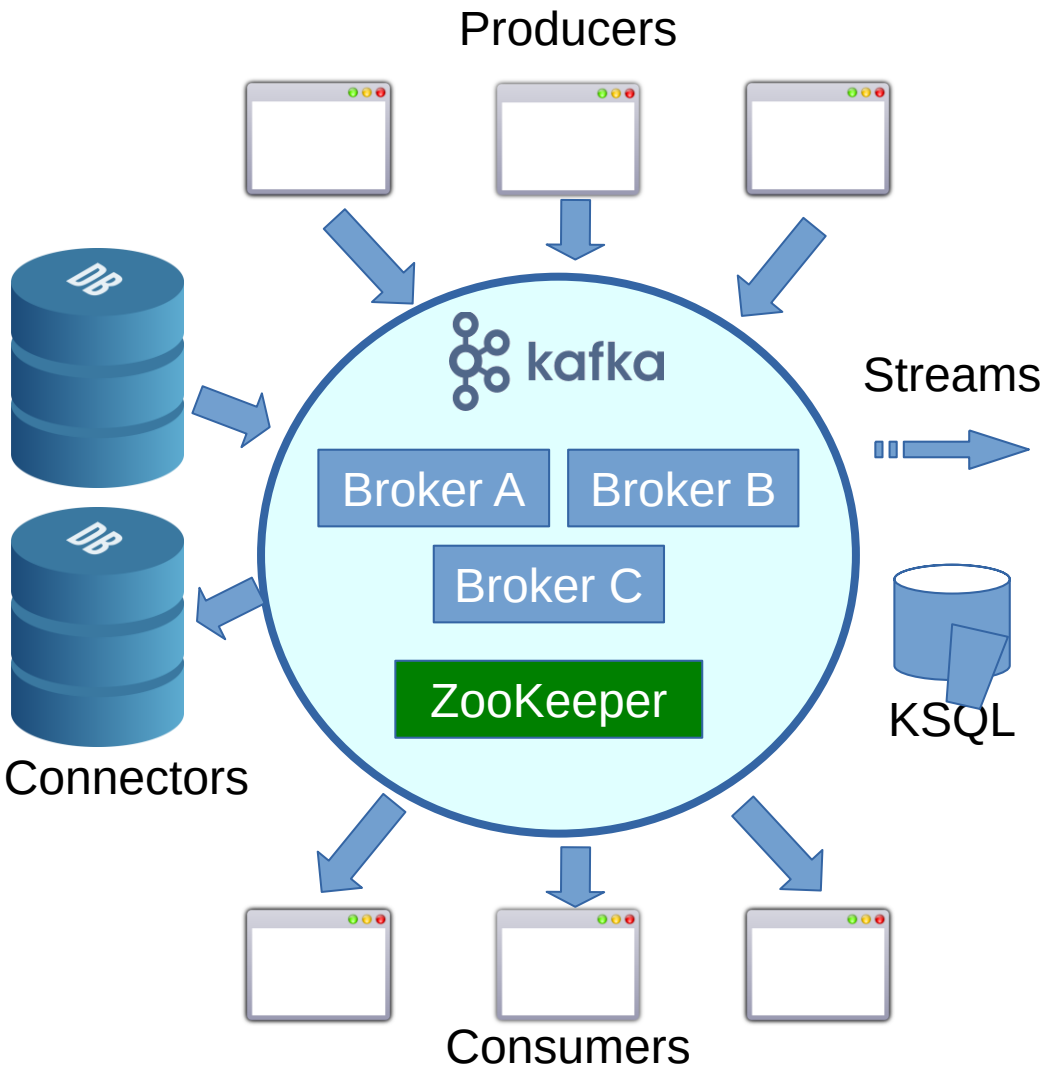

REPOTEC
IPT


Menu
0 %
08:46
LSM303.h
LSM303.cpp
Trash

Apache Kafka

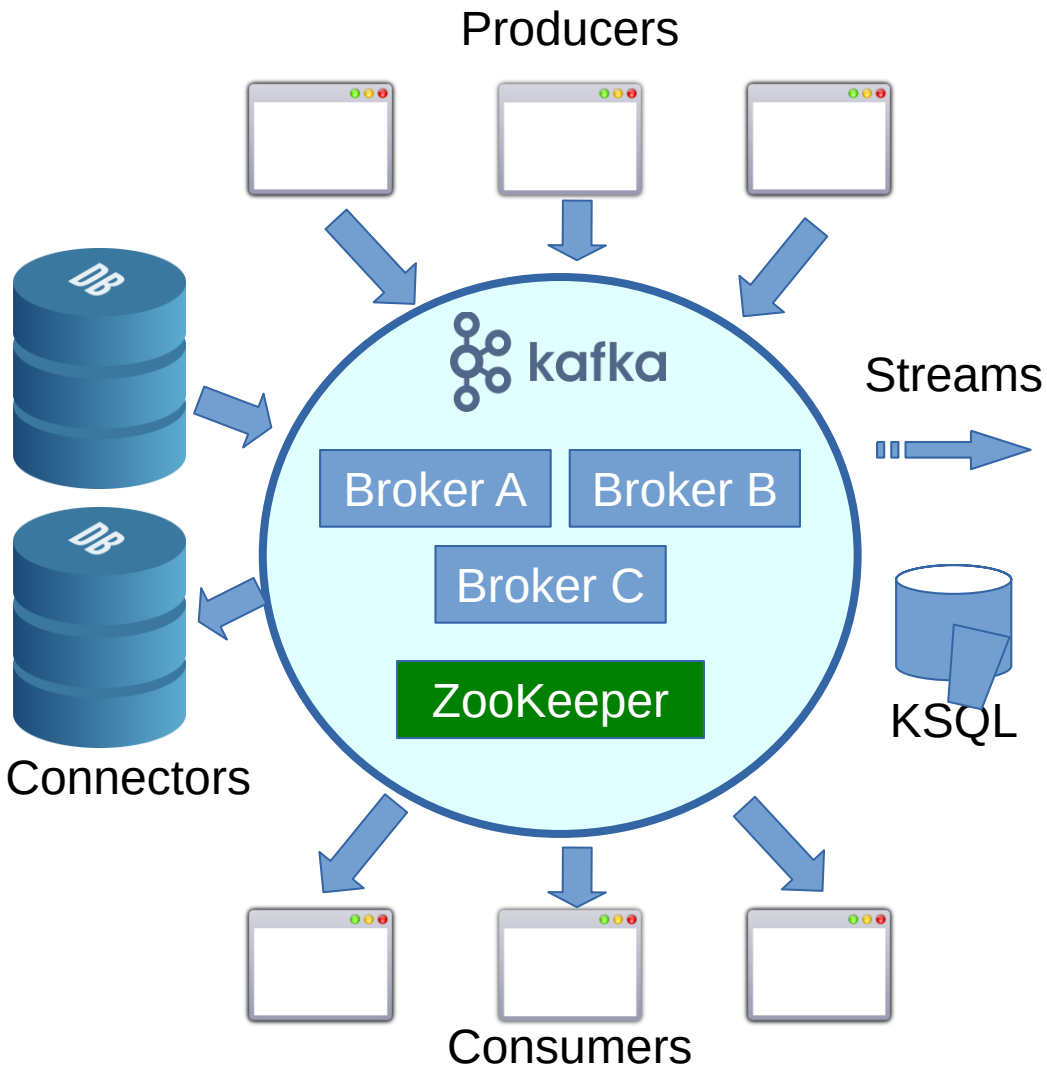
Kafka Main Concepts

- Kafka is run as a cluster on one or more servers (brokers) that can span multiple datacenters.
- The Kafka cluster stores streams of records in categories called topics.
- Each record consists of a key, value, and timestamp.



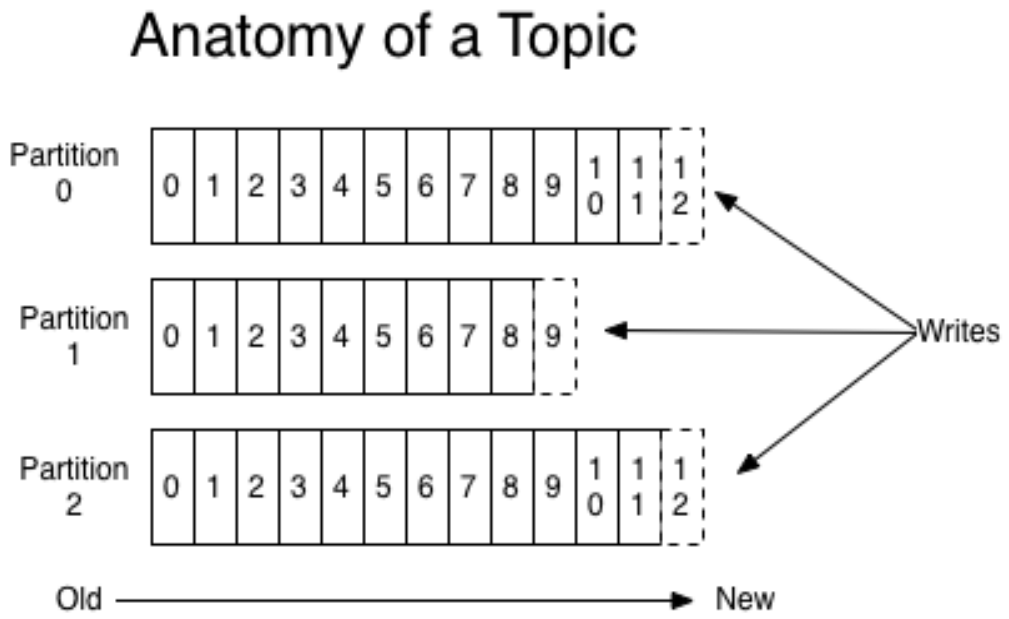
Kafka Core APIs

- The Producer API - publish a stream of records to one or more Kafka topics.
- The Consumer API - subscribe to one or more topics and process the stream of records produced to them.
- The Streams API - a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems – e.g. connector to a DB might capture every change in a table



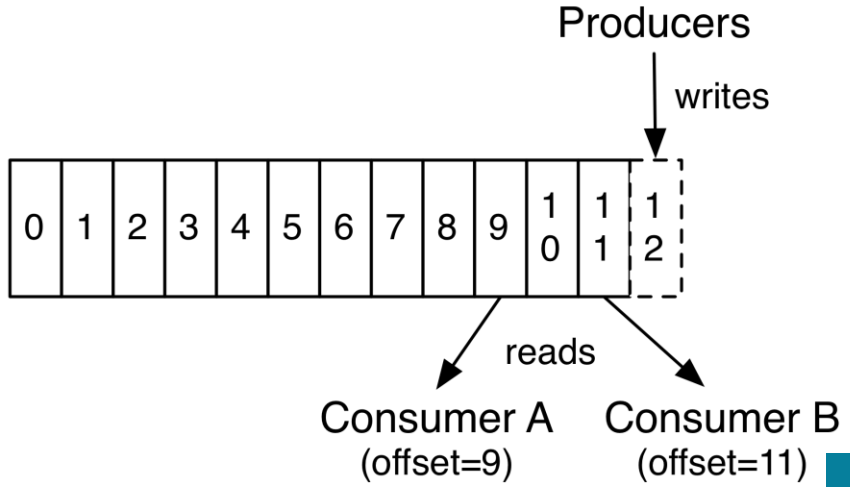
Topics and Logs

- Topic = stream of records
- A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data
- For each topic, the Kafka cluster maintains a partitioned log --->
- Each partition is an ordered, immutable sequence of records that is continually appended to - a structured commit log
- The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.



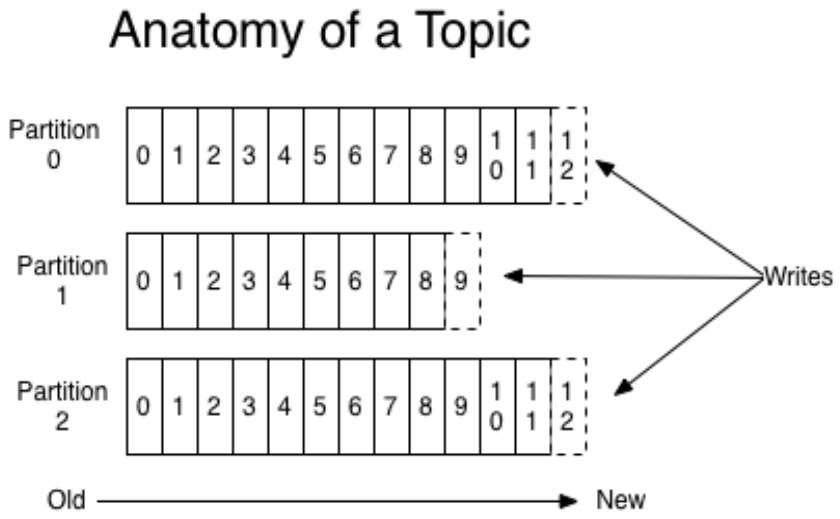
Consumers Offset and Data Retention

- Kafka cluster durably persists all published records - whether or not they have been consumed, using configurable retention period
- Kafka performance is effectively constant with respect to data size
- Offset or position of that consumer in the log – controlled by the consumer: normally a consumer will advance its offset linearly as it reads records, but, since the position is controlled by the consumer it can consume records in any order. E.g. a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now".
- Kafka consumers are very cheap - they can come and go without much impact on the cluster or on other consumers.

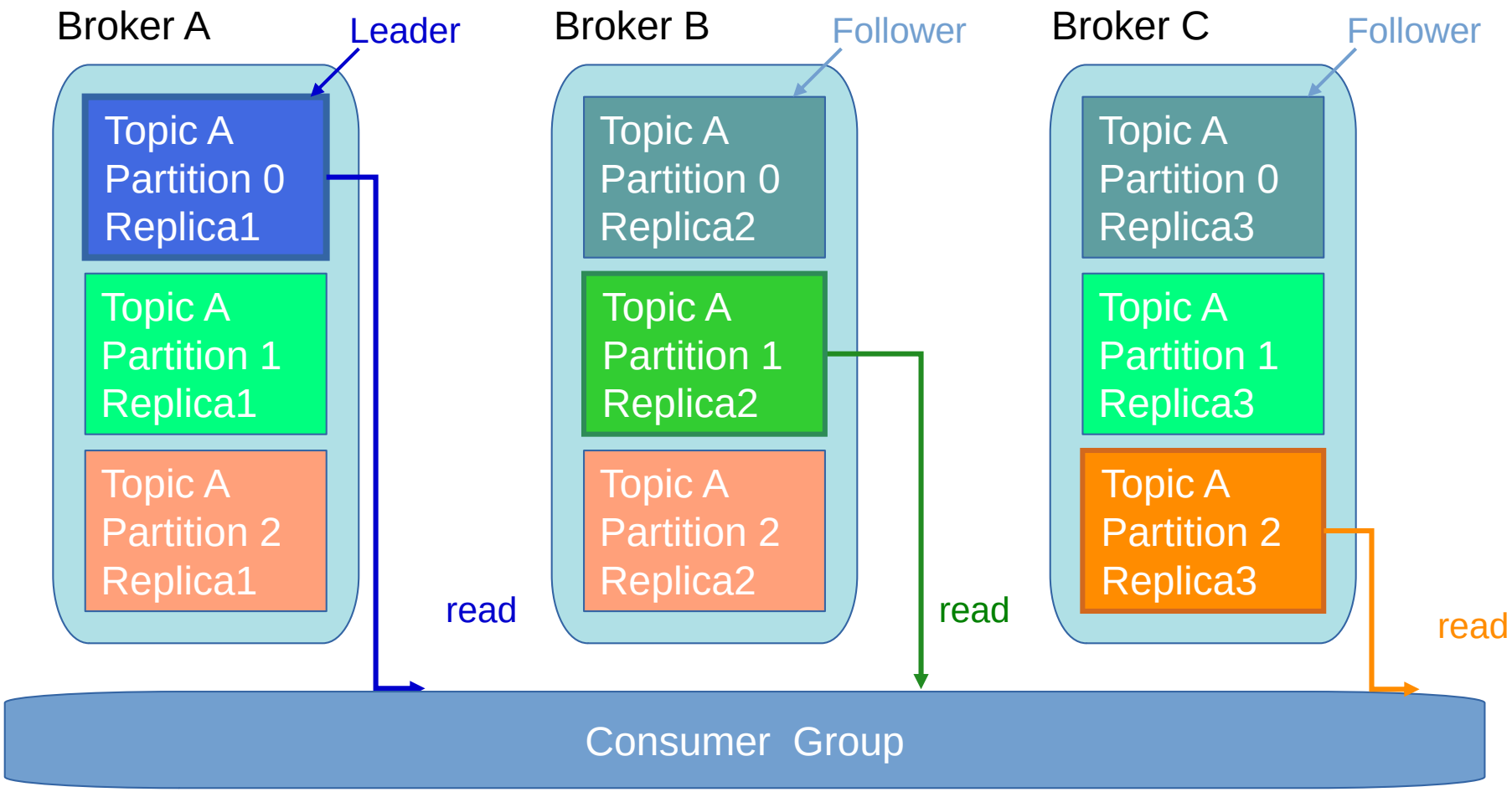


Kafka Partitions and Distribution

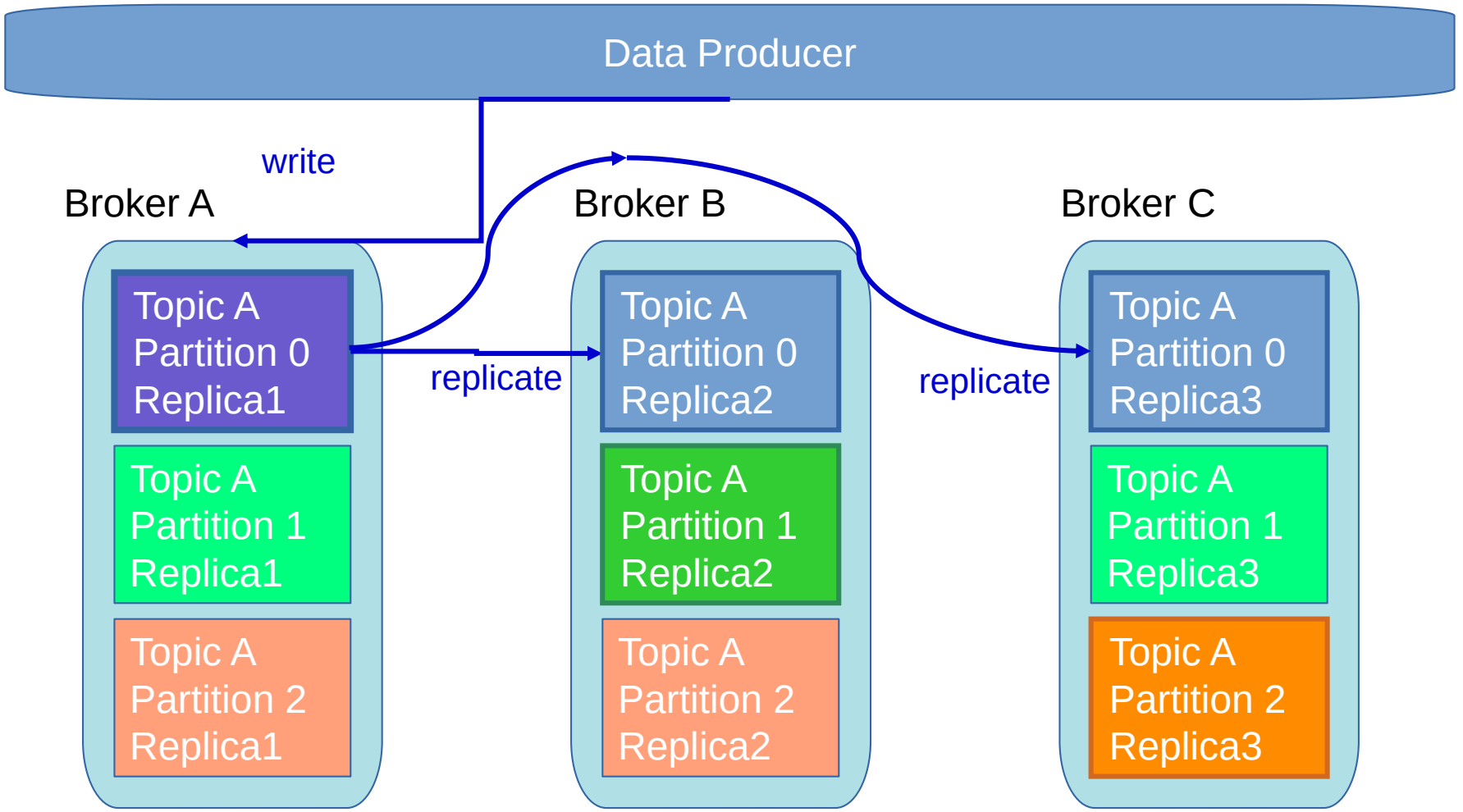
- The partitions in the log serve several purposes:
  1. Allow the log to scale beyond a size that will fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data.
  2. Act as the unit of parallelism—more on that in next slide



Distribution of Partitions and Read Balancing



Data Replication

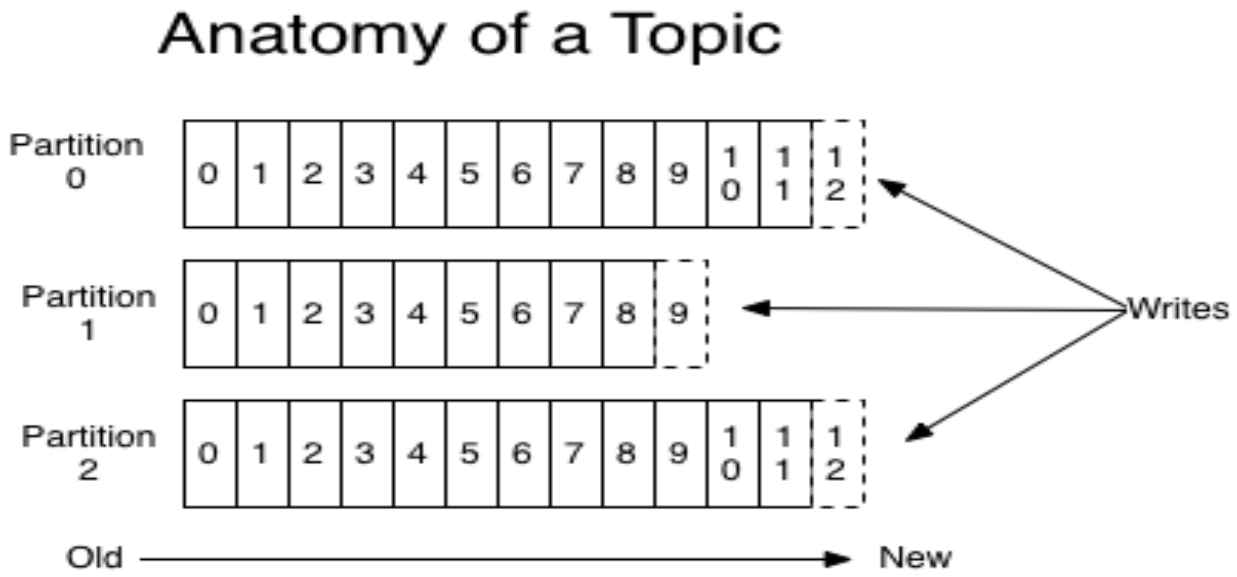


Kafka Brokers

- Kafka is maintained as clusters where each node within a cluster is called a Broker. Multiple brokers allow us to evenly distribute data across multiple servers and partitions. This load should be monitored continuously and brokers and topics should be reassigned when necessary.
- Each Kafka cluster will designate one of the brokers as the Controller which is responsible for managing and maintaining the overall health of a cluster, in addition to the basic broker responsibilities. Controllers are responsible for creating/deleting topics and partitions, taking action to rebalance partitions, assign partition leaders, and handle situations when nodes fail or get added. Controllers subscribe to receive notifications from ZooKeeper which tracks the state of all nodes, partitions, and replicas.

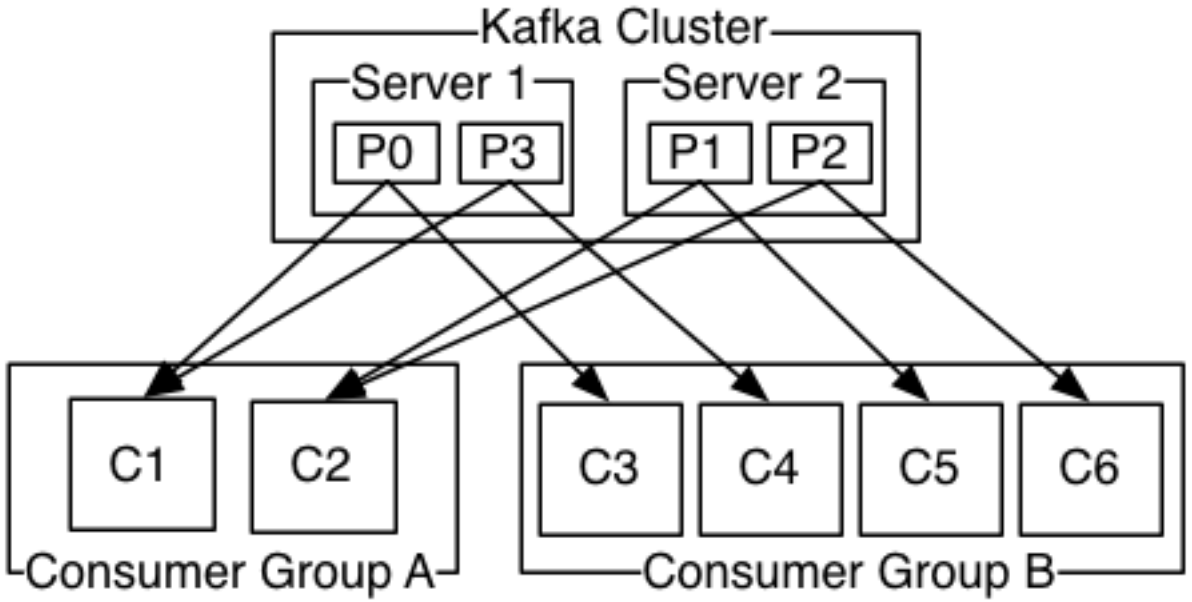
Kafka Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which record to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (e.g. based key's hash value)



Kafka Consumers

- Consumers label themselves with a consumer group, and each record published to a topic is delivered to one consumer instance within each consumer group. Can be separate processes/machines
- Same consumer group --> records will effectively be load balanced
- Different consumer groups, --> broadcast to all the consumers



Kafka Consumer Groups

- Consumer group = "logical subscriber" -> many consumer instances for scalability and fault tolerance = publish-subscribe semantics where the subscriber is a cluster of consumers instead of a single process.
- The way consumption is implemented in Kafka is by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a "share" of partitions at any point in time.
- Group membership is handled by the Kafka protocol dynamically. If new instances join the group they will take some partitions from other group members; if an instance dies, its partitions will be distributed.
- Kafka only provides a total order over records within a partition, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data by key is sufficient for most apps.
- If you require a total order over records --> use a topic that has only one partition = only one consumer process per consumer group.

Consumer API

Kafka Simple Consumer Example - I

```
public class DemoConsumer {
    private Properties props = new Properties();
    KafkaConsumer<String, String> consumer;

    public DemoConsumer() {
        props.setProperty("bootstrap.servers", "localhost:9093");
        props.setProperty("group.id", "dmlConsumer");
        props.setProperty("enable.auto.commit", "true");
        props.setProperty("auto.commit.interval.ms", "1000");
        props.setProperty("key.deserializer",
                          "org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty("value.deserializer",
                          "org.apache.kafka.common.serialization.StringDeserializer");
        consumer = new KafkaConsumer<>(props);
    }
```

Kafka Simple Consumer Example – II - Subscribe

```
public void run() {
    consumer.subscribe(Collections.singletonList("events"));
    try {
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            if (records.count() > 0) {
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf(
                            "offset = %d, key = %s, value = %s, headers = %s, topic = %s, partition = %d%n",
      record.offset(), record.key(), record.value(), record.headers(), record.topic(), record.partition());
                }
            }
        }
    } finally {
        consumer.close();
    }
}
```

Kafka Simple Consumer Example – II - Assign

```
public void run() {
    TopicPartition partition = new TopicPartition("events", 0);
    consumer.assign(List.of(partition));
    consumer.seek(partition, 0);
    try {
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            if (records.count() > 0) {
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf(
                            "offset = %d, key = %s, value = %s, headers = %s, topic = %s, partition = %d%n",
      record.offset(), record.key(), record.value(), record.headers(), record.topic(), record.partition());
                }
            }
        }
    } finally {
        consumer.close();
    }
}
```

Kafka Simple Consumer Example - III

```
    public static void main(String[] args) {
        DemoConsumer consumer = new DemoConsumer();
        consumer.run();
    }
}
```

Producer API

Kafka Producer send() Method

- Asynchronously sends a record to a topic:

new ProducerRecord(*TOPIC*, reading.getId(), reading)

- Allows sending many records without blocking for a broker response
- send() method returns a Future
- Two forms:
  - send() method without a callback
  - send() method with a callback – the callback gets invoked when the broker has acknowledged the send [ACK = -1 (all ISR), 0 or 1(leader)]
- Callbacks for records sent to same partition are executed in the sent order
- Callback receives RecordMetadata which contains a record's partition, offset, and timestamp, and possible Exception if there was an error sending.

Kafka send() Method Exceptions

- InterruptException - If thread is interrupted while blocking
- SerializationException - If key or value can not be serialized using configured serializers
- TimeoutException – when fetching metadata or allocating memory exceeds max.block.ms, or getting **acks** from Broker exceed timeout.ms, etc.
- KafkaException - when Kafka error occurs, but not in public API
- AuthenticationException - if authentication fails
- AuthorizationException - the producer is not allowed to write
- IllegalStateException - if a transactional.id has been configured and no transaction has been started, or when send() invoked on closed producer

Kafka Producer flush() and close() Methods

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    executor.shutdown();
    try {
        executor.awaitTermination(200, TimeUnit.MILLISECONDS);
        log.info("Flushing and closing producer");
        producer.flush();
        producer.close(10_000, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        log.warn("shutting down", e);
    }
}));
```

Kafka Producer partitionsFor() Method

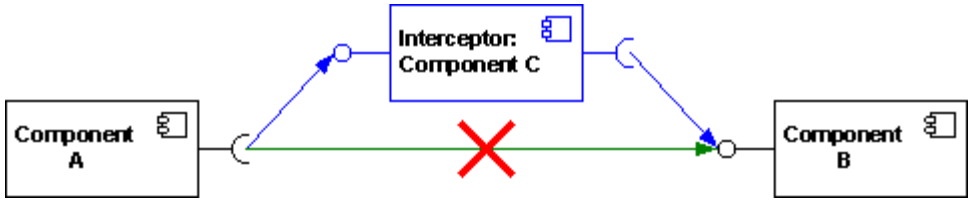
- partitionsFor(topic) - returns meta-data for partitions:

public List<PartitionInfo> partitionsFor(String topic)

- Used by producers that implement their own partitioning – for custom partitioning
- PartitionInfo consists of topic, partition, leader node (Node), replica nodes (Node[]) and inSyncReplica nodes.
- Node consists of id, host, port, and rack

Kafka Producer Interceptors

- Interceptor design pattern:



- Activate interceptors by adding them to interceptor.classes property of the producer

```
props.put(
        ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
        CountingProducerInterceptor.class.getName());
```

- Producer interceptor methods:

```
public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)
public void onAcknowledgement(RecordMetadata metadata, Exception exception)
```

Kafka Producer flush() and close() Methods

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    executor.shutdown();
    try {
        executor.awaitTermination(200, TimeUnit.MILLISECONDS);
        log.info("Flushing and closing producer");
        producer.flush();
        producer.close(10_000, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        log.warn("shutting down", e);
    }
}));
```

Kafka Producer metrics() Method

- metrics() - used to get a map of metrics:

public Map<MetricName,? extends Metric> metrics()

- Returns a full set of producer metrics.
- MetricName consists of name, group, description, and tags (Map).
- Metric consist of a MetricName and a Measurable value (double) or Object (gauge).

Kafka Transactions Simple Example

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer =
                                    new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());
producer.initTransactions();
try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Integer.toString(i)));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // We can't recover from these exceptions, so our only option is to close the producer and exit.
    producer.close();
} catch (KafkaException e) {
    // For all other exceptions, just abort the transaction and try again.
    producer.abortTransaction();
}
producer.close();
```

Kafka Transactions – Atomic Read & Write to Topic

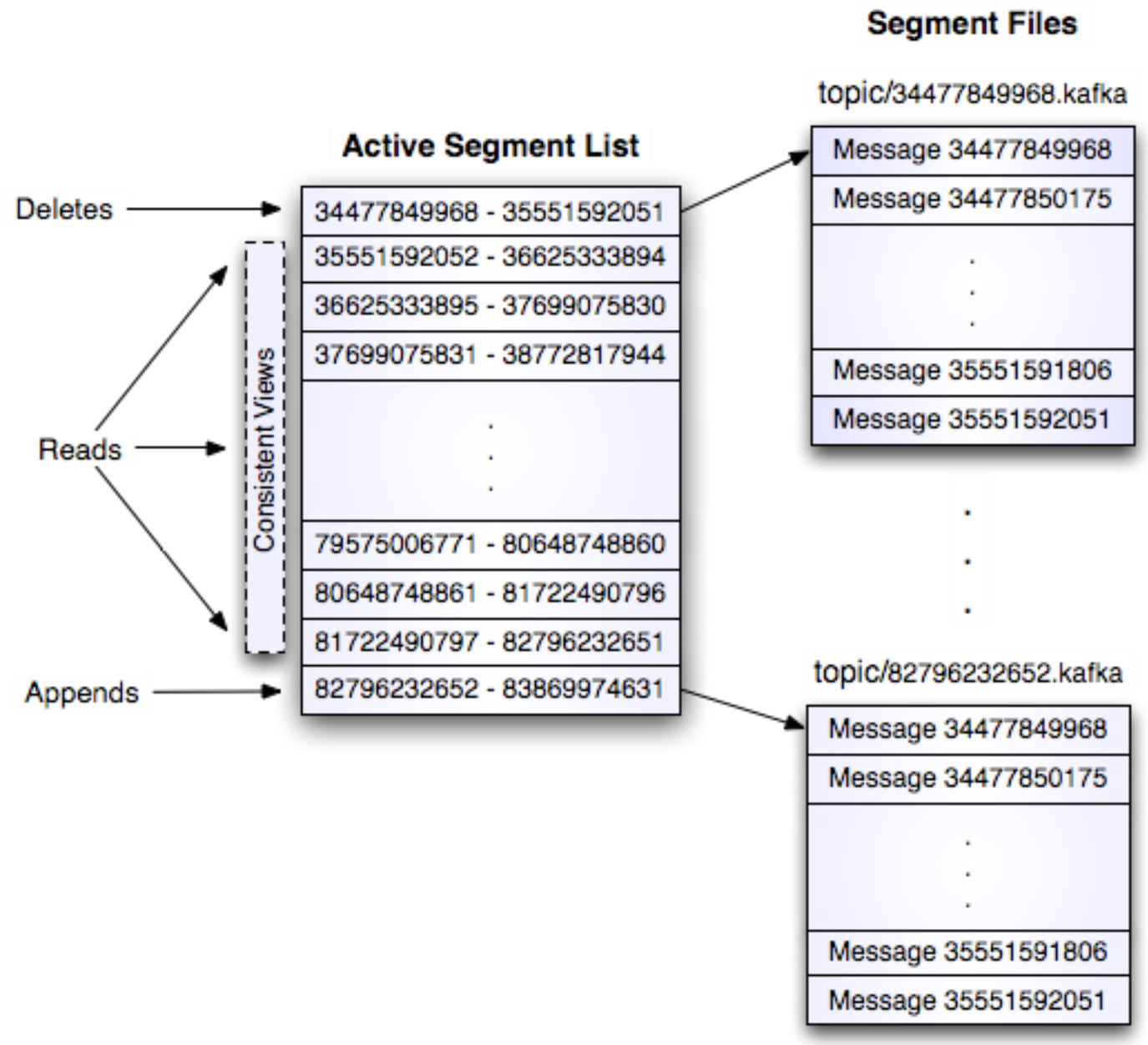
```
// Poll and validate deposit events
Deposits = validate(consumer.poll(100));

// Atomically send valid deposits and commit offsets
producer.beginTransaction();
producer.send(validatedDeposits);
producer.sendOffsetsToTransaction(offsets(consumer));
producer.endTransaction();
```

Kafka Details

Kafka Log Format

- Messages are stored inside topics within a log structured format, where the data gets written sequentially.
- A message can have a maximum size of 1MB by default, and while this is configurable, Kafka was not designed to process large size records. It is recommended to split large payloads into smaller messages, using identical key values so they all get saved in the same partition as well as assigning part numbers to each split message in order to reconstruct it on the consumer.
- Messages (aka Records) are always written in record batches, Record batches and records have their own headers. The detailed format of each is described in: http://kafka.apache.org/documentation/#recordbatch

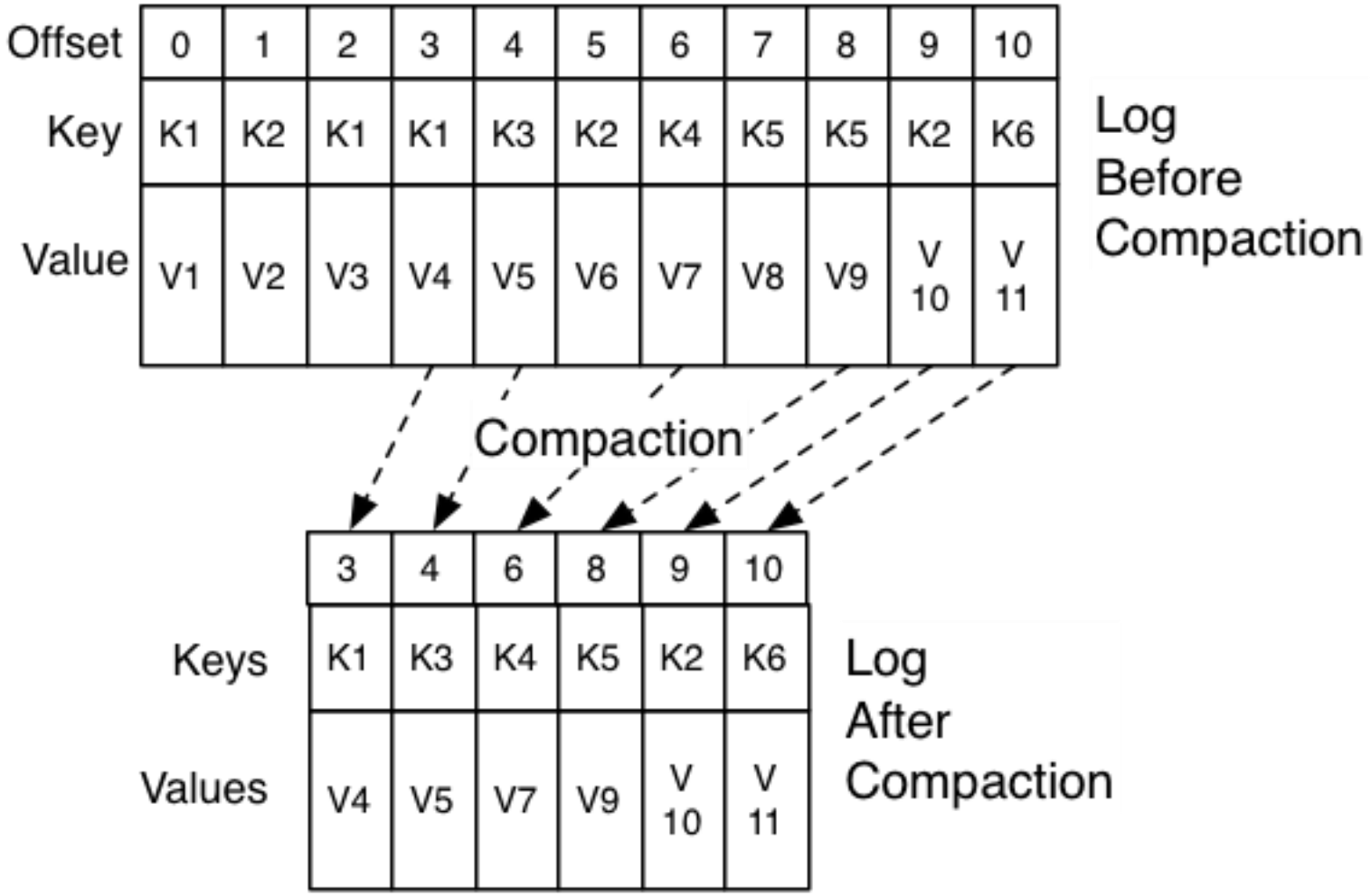
Kafka Log Implementation



Log Compaction - I

- Log compaction ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition.
- It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance – e.g. database change subscription, event sourcing, journaling for high-availability.
- An important class of data streams are the log of changes to keyed, mutable data (for example, the changes to a DB table).
- Example: 123 => bill@microsoft.com; ...
- 123 => bill@gatesfoundation.org ...
- 123 => bill@gmail.com ...

Log Compaction - III



Kafka Guarantees

1. Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a record M1 is sent by the same producer as a record M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.

2. A consumer instance sees records in the order they are stored in the log.

3. For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

Kafka as a Storage System

- Any message queue that allows publishing messages decoupled from consuming them is effectively acting as a storage system for the in-flight messages. Kafka is a very good storage system.
- Data written to Kafka is written to disk and replicated for fault-tolerance. Kafka allows producers to wait on acknowledgement so that a write isn't considered complete until it is fully replicated and guaranteed to persist even if the server written to fails.
- The disk structures Kafka uses scale well – Kafka will perform the same whether you have 50 KB or 50 TB of persistent data.
- Efficient storage + allowing the clients to control their read position => Kafka becomes a special purpose distributed filesystem dedicated to high-performance, low-latency commit log storage, replication, and propagation.

Using Kafka Command Line Tools

- kafka-topics.bat --list --bootstrap-server localhost:9092
- kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic my-new-topic
- kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic events
- kafka-topics.bat --list --bootstrap-server localhost:9092
- kafka-topics.bat --describe --bootstrap-server localhost:9092 --topic events
- kafka-console-producer.bat --bootstrap-server localhost:9092 --topic events
- kafka-console-producer.bat --bootstrap-server localhost:9092 --topic events –sync
- kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic events --from-beginning

Replicated Topic Using Kafka

copy config\server.properties config\server-1.properties

copy config\server.properties config\server-2.properties

config/server-1.properties:
    broker.id=1
    listeners=PLAINTEXT://:9093
    log.dirs=D:\\CourseKafka\\kafka_2.12-2.2.1\\kafka-logs-1

config/server-2.properties:
    broker.id=2
    listeners=PLAINTEXT://:9094
    log.dirs=D:\\CourseKafka\\kafka_2.12-2.2.1\\kafka-logs-2

bin\windows\kafka-server-start config\server-1.properties

bin\windows\kafka-server-start config\server-2.properties

kafka-topics --describe --bootstrap-server localhost:9092 --topic my-replicated-topic

wmic process where "caption = 'java.exe' and commandline like '%server-1.properties%'" get processid

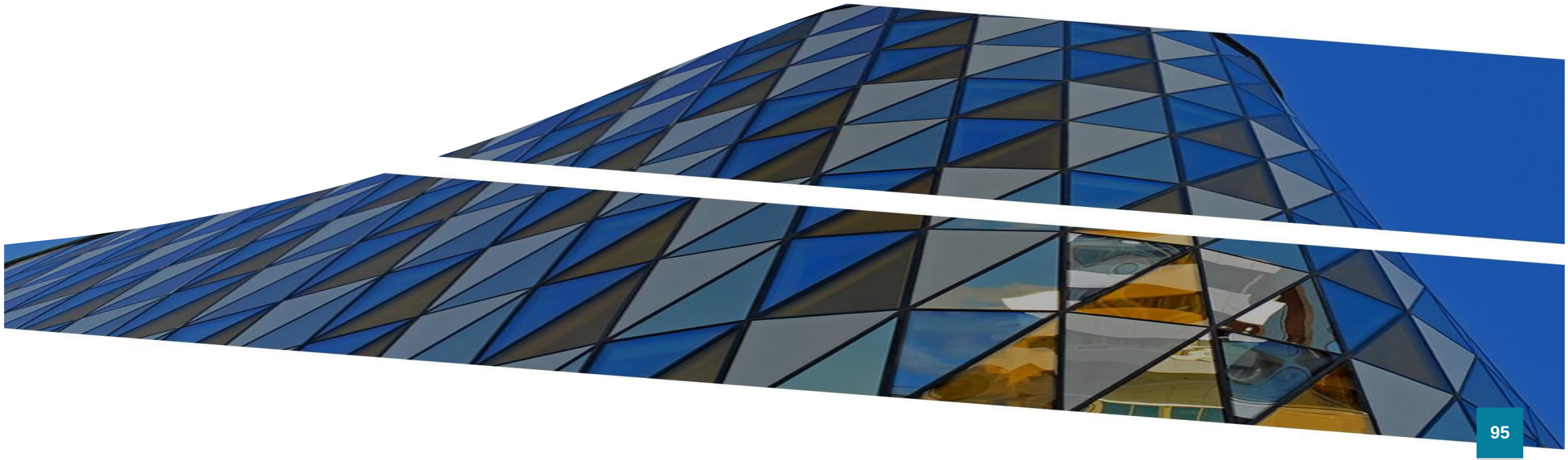
taskkill /F /PID pid_number

Consumers & Consumer Groups

[https://kafka.apache.org/documentation/#basic_ops_consumer_group]

- kafka-consumer-groups --bootstrap-server localhost:9092 –list
- kafka-consumer-groups --bootstrap-server localhost:9092 --describe --group event-consumer

Kafka Streams API



Kafka Streams

- By combining storage and low-latency subscriptions, streaming applications can treat both past and future data the same way. That is a single application can process historical, stored data but rather than ending when it reaches the last record it can keep processing as future data arrives. This is a generalized notion of stream processing that subsumes batch processing as well as message-driven applications ==> Kappa architecture
- Likewise for streaming data pipelines the combination of subscription to real-time events make it possible to use Kafka for very low-latency pipelines; but the ability to store data reliably make it possible to use it for critical data where the delivery of data must be guaranteed or for integration with offline systems that load data only periodically or may go down for extended periods of time for maintenance. The stream processing facilities make it possible to transform data as it arrives.

Why you'll love using Kafka Streams?

- Elastic, highly scalable, fault-tolerant
- Deploy to containers, VMs, bare metal, cloud
- Equally viable for small, medium, & large use cases
- Fully integrated with Kafka security
- Write standard Java and Scala applications
- Exactly-once processing semantics
- No separate processing cluster required
- Develop on Mac, Linux, Windows

https://kafka.apache.org/documentation/streams/

Kafka Streams Advantages

- Designed as a **simple and lightweight client library**, which can be easily embedded in any Java application and integrated with any existing packaging, deployment and operational tools that users have for their streaming applications.
- Has **no external dependencies on systems other than Apache Kafka itself** as the internal messaging layer; notably, it uses Kafka's partitioning model to horizontally scale processing while maintaining strong ordering guarantees.
- Supports **fault-tolerant local state**, which enables very fast and efficient stateful operations like windowed joins and aggregations.

Kafka Streams Advantages - II

- Supports exactly-once processing semantics to guarantee that each record will be processed once and only once even when there is a failure on either Streams clients or Kafka brokers in the middle of processing.
- Employs one-record-at-a-time processing to achieve millisecond processing latency, and supports event-time based windowing operations with out-of-order arrival of records.
- Offers necessary stream processing primitives, along with a high-level Streams DSL and a low-level Processor API.

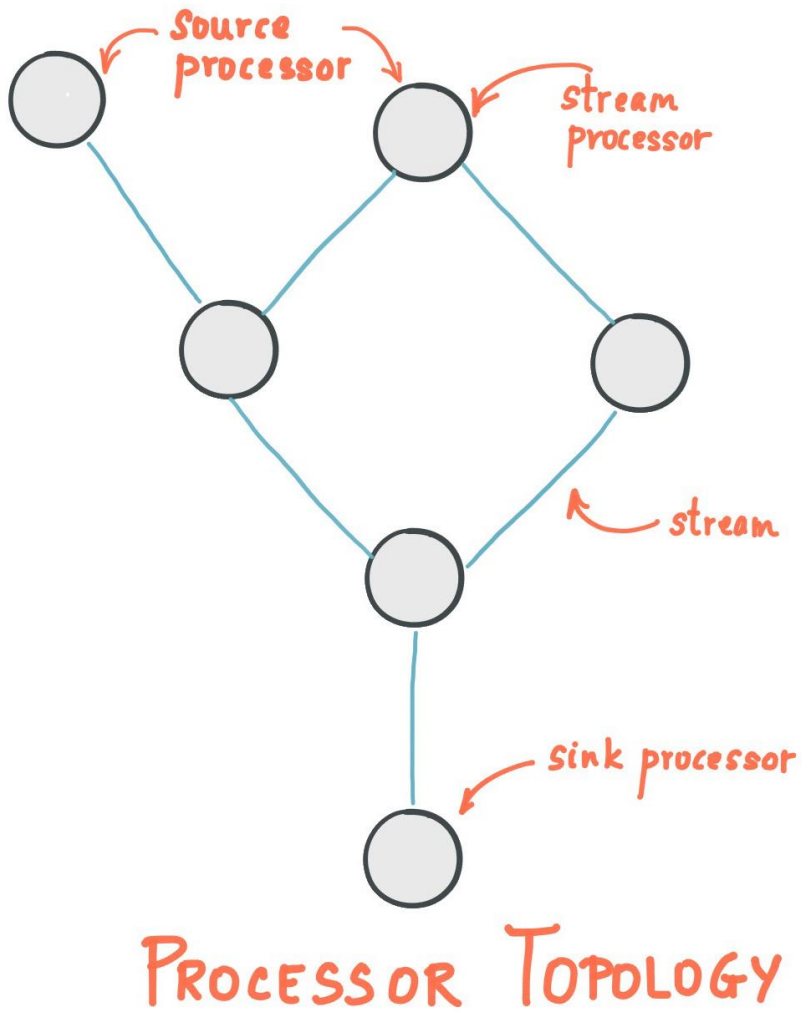
Stream Processing Topology - I

- A stream is the most important abstraction provided by Kafka Streams: it represents an unbounded, continuously updating data set. A stream is an ordered, replayable, and fault-tolerant sequence of immutable data records, where a data record is defined as a key-value pair.
- A stream processing application is any program that makes use of the Kafka Streams library. It defines its computational logic through one or more processor topologies, where a processor topology is a graph of stream processors (nodes) that are connected by streams (edges).
- A stream processor is a node in the processor topology; it represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in the topology, applying its operation to it, and may subsequently produce one or more output records to its downstream processors.

Stream Processing Topology - II

- There are two special processors in the topology:
- Source Processor: A source processor is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors.
- Sink Processor: A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.
- Note that in normal processor nodes other remote systems can also be accessed while processing the current record. Therefore the processed results can either be streamed back into Kafka or written to an external system.

Kafka Stream Processing - DAG



https://kafka.apache.org/documentation/streams/

Time in Kafka Streams - I

- A critical aspect in stream processing is the notion of time, and how it is modeled and integrated. For example, some operations such as windowing are defined based on time boundaries. Common notions of time in streams are:
- Event time - the point in time when an event or data record occurred, i.e. was originally created "at the source".
- Processing time - the point in time when the event or data record happens to be processed by the stream processing application, i.e. when the record is being consumed.
- Example: Imagine an analytics application that reads and processes the geo-location data reported from car sensors to present it to a fleet management dashboard. Here, processing-time in the analytics application might be milliseconds or seconds (e.g. for real-time pipelines based on Apache Kafka and Kafka Streams) or hours (e.g. for batch pipelines based on Apache Hadoop or Apache Spark) after event-time.

https://kafka.apache.org/documentation/streams/

Time in Kafka Streams - II

- Ingestion time - The point in time when an event or data record is stored in a topic partition by a Kafka broker. The difference to event time is that this ingestion timestamp is generated when the record is appended to the target topic by the Kafka broker, not when the record is created "at the source". The difference to processing time is that processing time is when the stream processing application processes the record. For example, if a record is never processed, there is no notion of processing time for it, but it still has an ingestion time.

- The choice between event-time and ingestion-time is actually done through the configuration of Kafka (not Kafka Streams) – timestamps are automatically embedded into Kafka messages. Depending on Kafka's configuration these timestamps represent event-time or ingestion-time. The respective Kafka configuration setting can be specified on the broker level or per topic. The default timestamp extractor in Kafka Streams will retrieve these embedded timestamps as-is. Hence, the effective time semantics of your application depend on the effective Kafka configuration for these embedded timestamps.

https://kafka.apache.org/documentation/streams/

Time in Kafka Streams - III

- **log.message.timestamp.type** – define whether the timestamp in the message is message create time or log append time. The value should be either `CreateTime` or `LogAppendTime`
- **log.message.timestamp.difference.max.ms** – The maximum difference allowed between the timestamp when a broker receives a message and the timestamp specified in the message. If log.message.timestamp.type=CreateTime, a message will be rejected if the difference in timestamp exceeds this threshold. This configuration is ignored if log.message.timestamp.type=LogAppendTime. The maximum timestamp difference allowed should be no greater than log.retention.ms to avoid unnecessarily frequent log rolling.

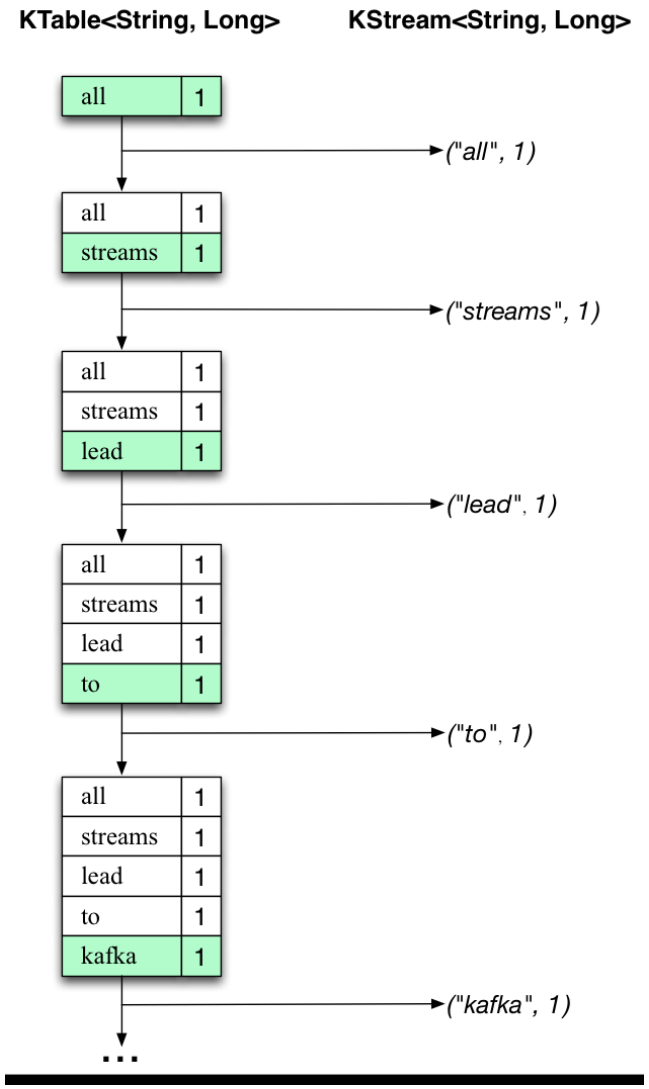
https://kafka.apache.org/documentation/streams/

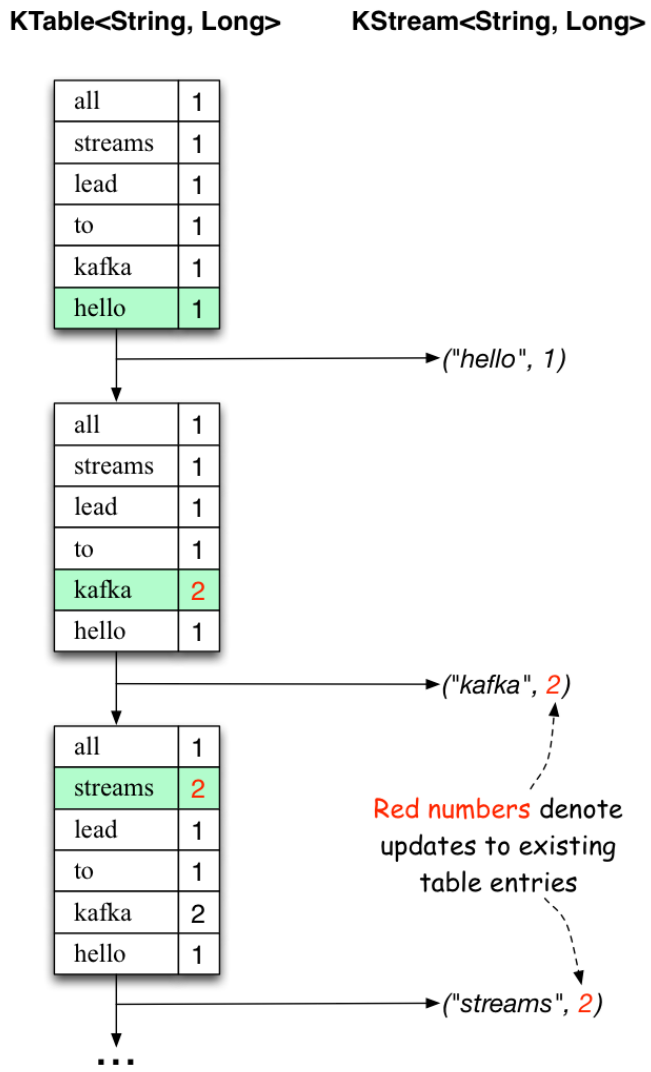
Custom TimestampExtractor

```
@Slf4j
public class CustomTimeExtractor implements TimestampExtractor {
    @Override
    public long extract(ConsumerRecord<Object, Object> record, long partitionTime)  {
        final long timestamp = record.timestamp();

        // `TemperatureReading` is your own custom class, which we assume has a method that returns
        // the embedded timestamp (in milliseconds).
        var myReading = (TemperatureReading) record.value();
        if (myReading != null) {
            return java.sql.Timestamp.valueOf(myReading.getTimestamp()).getTime();
        }
        else {
            // Kafka allows `null` as message value.  How to handle such message values
            // depends on your use case.  In this example, we decide to fallback to
            // wall-clock time (= processing-time).
            return System.currentTimeMillis();
        }
    }
}
```

https://kafka.apache.org/documentation/streams/

Kafka Stream Processing Example





Kafka Streams Dependencies

```
dependencies {
    implementation 'org.apache.kafka:kafka-clients:3.6.0'
    implementation 'org.apache.kafka:kafka-streams:3.6.0‘
    …
}
```

https://kafka.apache.org/documentation/streams/

Kafka Streams Code Skeleton

```
public static void main(String[] args) {
    // Use the builders to define the actual processing topology, e.g. to specify from which input topics to read,
    // which stream operations (filter, map, etc.) should be called, and so on.

    StreamsBuilder builder = ...;  // when using the DSL
    Topology topology = builder.build();
    //
    // OR
    //
    Topology topology = ...; // when using the Processor API

    // Use the configuration to tell your application where the Kafka cluster is,
    // which Serializers/Deserializers to use by default, to specify security settings,and so on.
    Properties props = ...;
    KafkaStreams streams = new KafkaStreams(topology, props);

   // Add shutdown hook to stop the Kafka Streams threads. You can optionally provide a timeout to `close`.
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));

}
```

https://kafka.apache.org/documentation/streams/

Stream Partitions and Tasks - II

- Kafka messaging layer partitions data for storing and transporting it.
- Kafka Streams partitions data for processing it.
- In both cases, this partitioning is what enables data locality, elasticity, scalability, high performance, and fault tolerance. Kafka Streams uses the concepts of partitions and tasks as logical units of its parallelism model based on Kafka topic partitions.
- Each stream partition is a totally ordered sequence of data records and maps to a Kafka topic partition.
- A data record in the stream maps to a Kafka message from that topic.
- The keys of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

https://kafka.apache.org/documentation/streams/

Stream Partitions and Tasks - II

- An application's processor topology is scaled by breaking it into multiple tasks.
- Kafka Streams creates a fixed number of tasks based on the input stream partitions for the application, with each task assigned a list of partitions from the input streams (i.e., Kafka topics).
- The assignment of partitions to tasks never changes so that each task is a fixed unit of parallelism of the application.
- Tasks can then instantiate their own processor topology based on the assigned partitions; they also maintain a buffer for each of its assigned partitions and process messages one-at-a-time from these record buffers.
- As a result stream tasks can be processed independently and in parallel without manual intervention.

https://kafka.apache.org/documentation/streams/

Stream Partitions and Tasks - III

- Kafka Streams is NOT a resource manager, but a library that "runs" anywhere its stream processing application runs.
- Multiple instances of the application are executed either on the same machine, or spread across multiple machines and tasks can be distributed automatically by the library to those running application instances.
- Assignment of partitions to tasks never changes - if an application instance fails, all its assigned tasks will be automatically restarted on other instances and continue to consume from the same stream partitions.
- Topic partitions are assigned to tasks, and tasks are assigned to all threads over all instances, in a best-effort attempt to trade off load-balancing and stickiness of stateful tasks. For this assignment, Kafka Streams uses the StreamsPartitionAssignor class and doesn't let you change to a different assignor. If you try to use a different assignor, Kafka Streams ignores it.

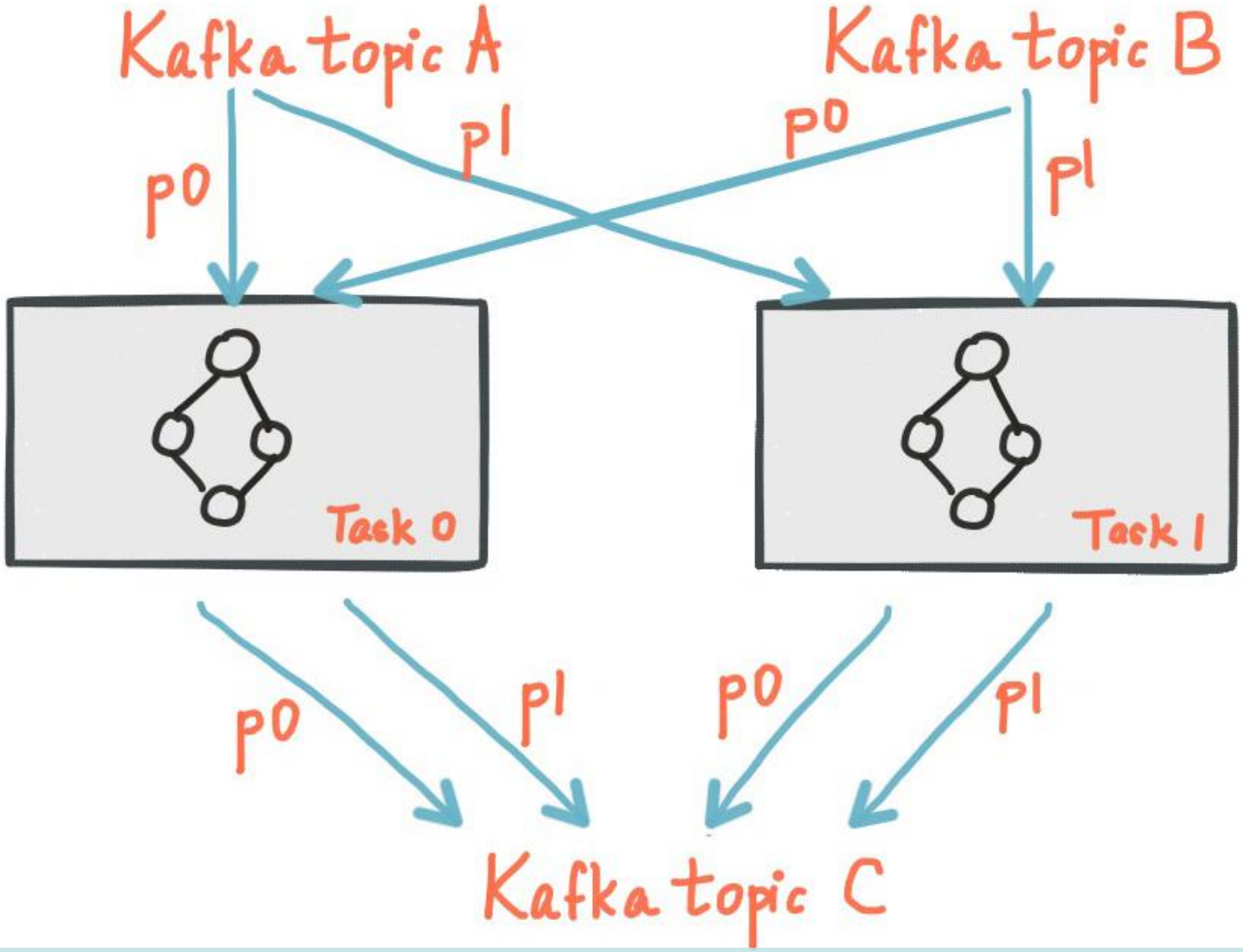
https://kafka.apache.org/documentation/streams/

StreamsPartitionAssignor Tasks Assignment Algorithm

1. Decode the subscriptions to assemble the metadata for each client and check for version probing.
2. Check all repartition source topics and use internal topic manager to make sure they have been created with the right number of partitions. Also verify and/or create any changelog topics with the correct number of partitions.
3. Use the partition grouper to generate tasks along with their assigned partitions, then use the configured TaskAssignor to construct the mapping of tasks to clients.
4. Construct the global mapping of host to partitions to enable query routing.
5. Within each client, assign tasks to consumer clients.

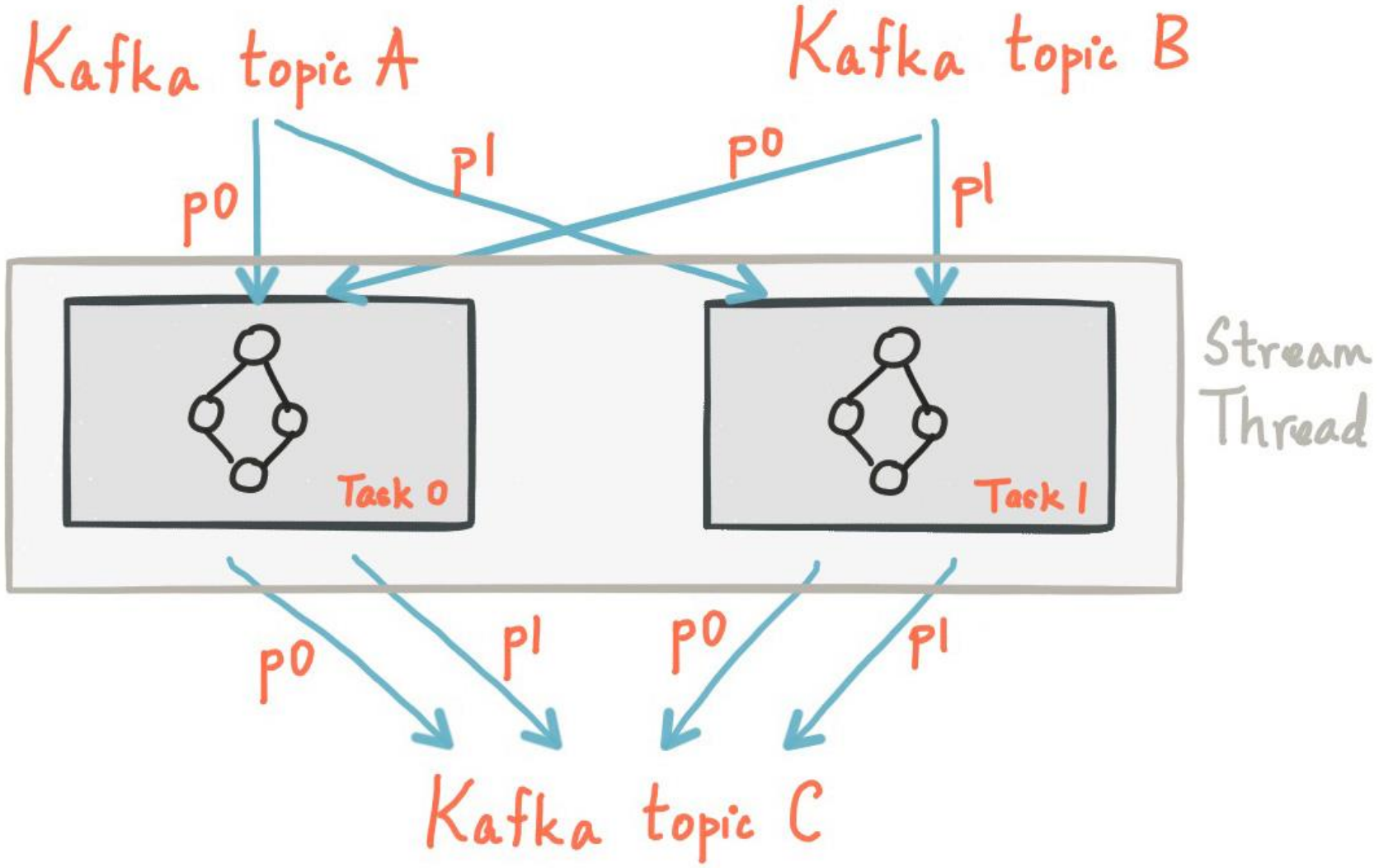
https://kafka.apache.org/documentation/streams/

Kafka Streams Partitions and Tasks - I



https://kafka.apache.org/documentation/streams/

Kafka Streams Partitions and Tasks - II



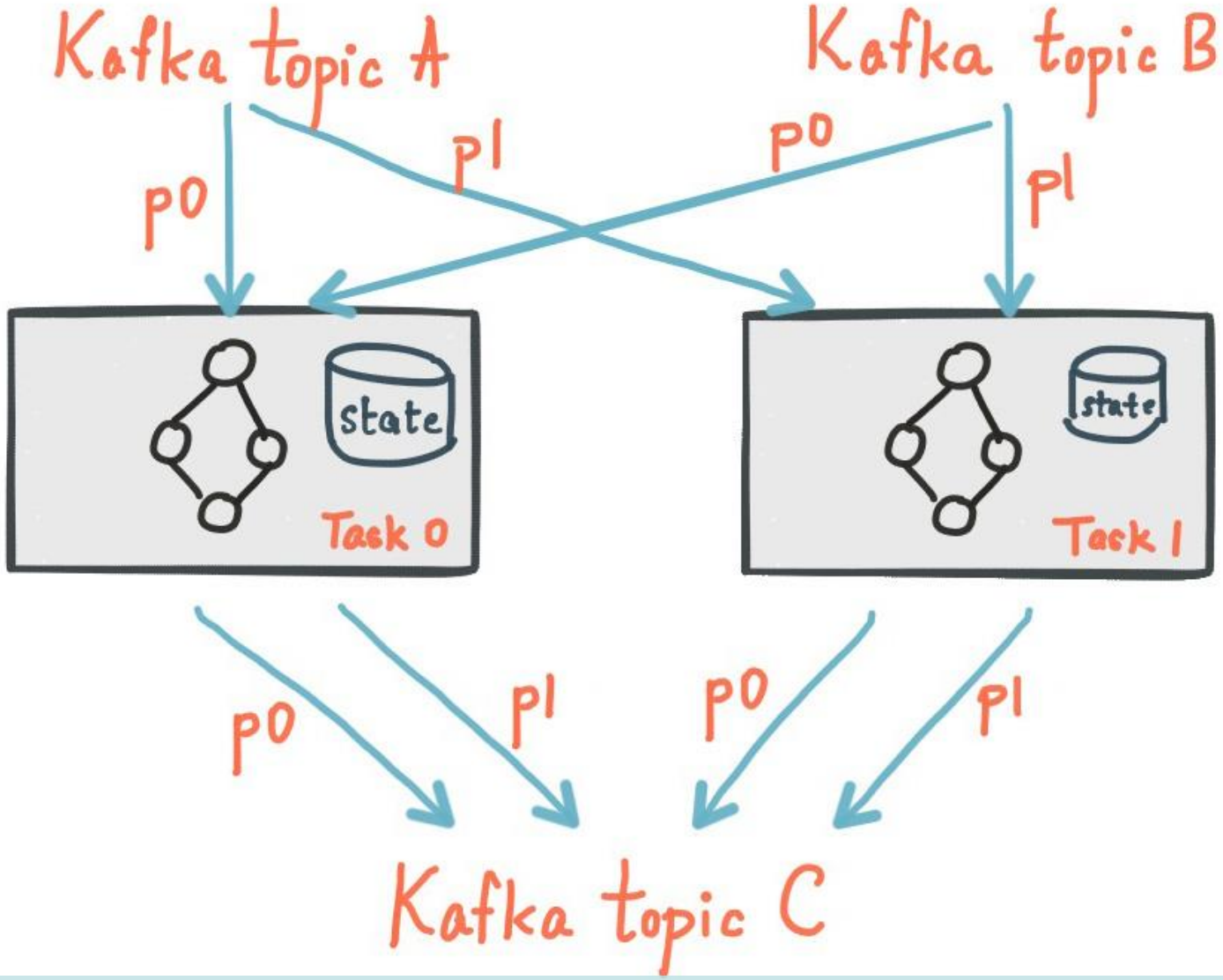
https://kafka.apache.org/documentation/streams/

Tasks Threading Model

- Starting more stream threads or more instances of the application merely amounts to replicating the topology and having it process a different subset of Kafka partitions, effectively parallelizing processing.
- It is worth noting that there is no shared state amongst the threads, so no inter-thread coordination is necessary.
- This makes it very simple to run topologies in parallel across the application instances and threads.
- The assignment of Kafka topic partitions amongst the various stream threads is transparently handled by Kafka Streams leveraging Kafka's coordination functionality.
- You can start as many threads of the application as there are input topic partitions so that, across all running instances of an application, every thread (or rather, the tasks it runs) has at least one input partition to process.

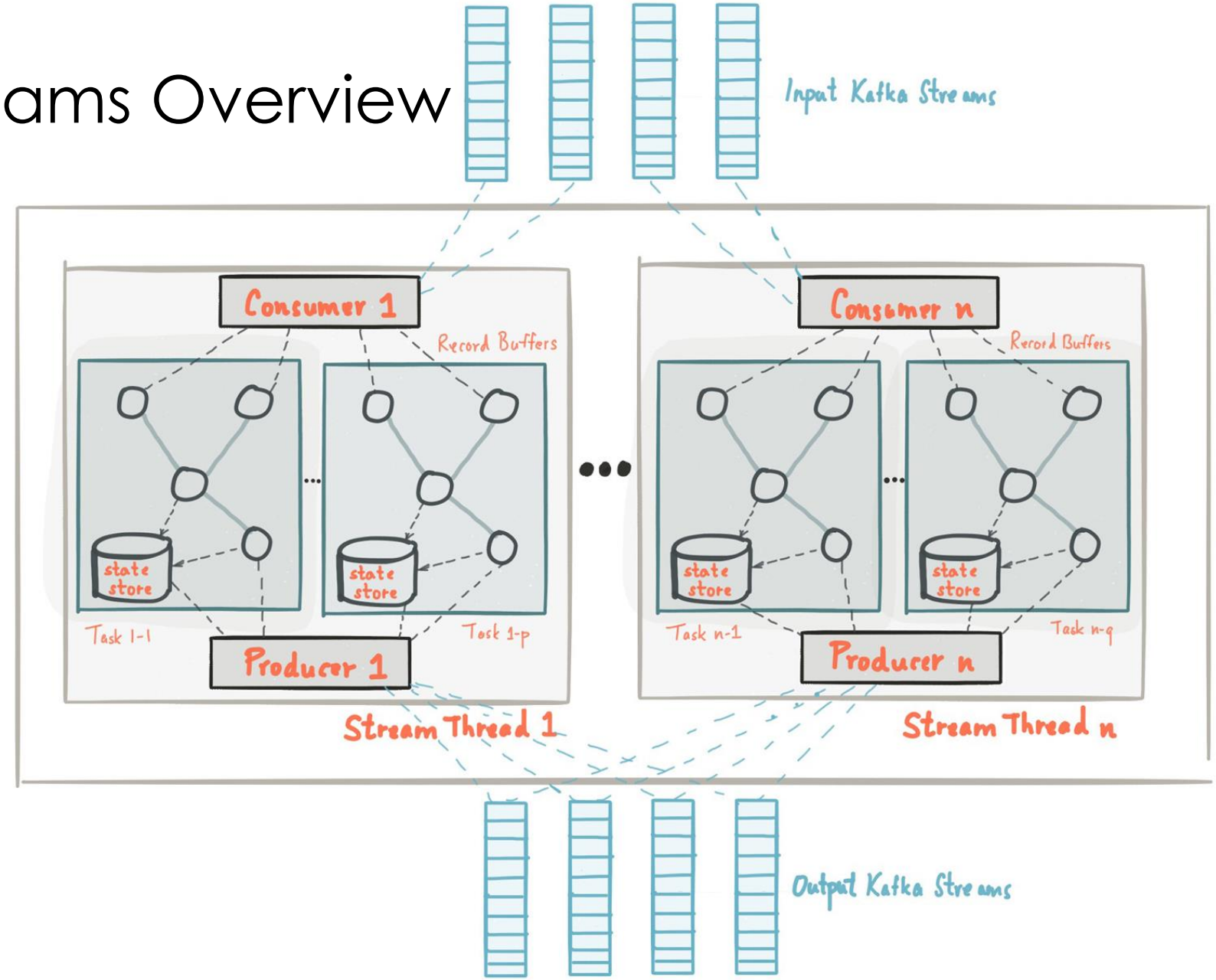
https://kafka.apache.org/documentation/streams/

Kafka Streams Partitions and Tasks - III



https://kafka.apache.org/documentation/streams/

Kafka Streams Overview



https://kafka.apache.org/documentation/streams/

Kafka Streams DSL & Processor API

- Processor API - allows developers to define and connect custom processors and to interact with state stores. With the Processor API, you can define arbitrary stream processors that process one received record at a time, and connect these processors with their associated state stores to compose the processor topology that represents a customized processing logic.
- Processor API can be used to implement both stateless as well as stateful operations, where the latter is achieved through the use of state stores.
- Kafka Streams DSL (Domain Specific Language) is built on top of the Streams Processor API. It is the recommended for most users, especially beginners. Most data processing operations can be expressed in just a few lines of DSL code.
- Combining the DSL and the Processor API – you can combine the convenience of the DSL with the power and flexibility of the Processor API as described in the section Applying processors and transformers (Processor API integration).

https://kafka.apache.org/documentation/streams/

Kafka Streams DSL: KStreams

- Only the Kafka Streams DSL has the notion of a KStream.
- KStream is an abstraction of a record stream, where each data record represents a self-contained datum in the unbounded data set. Using the table analogy, data records in a record stream are always interpreted as an "INSERT" -- think: adding more entries to an append-only ledger -- because no record replaces an existing row with the same key. Examples are a credit card transaction, a page view event, or a server log entry.
- To illustrate, let's imagine the following two data records are being sent to the stream: **("alice", 1) --> ("alice", 3)**
- If your stream processing application were to sum the values per user, it would return 4 for alice. Why? Because the second data record would not be considered an update of the previous record. Compare this behavior of KStream to KTable in next slide, which would return 3 for alice.

https://kafka.apache.org/documentation/streams/

Kafka Streams DSL: KTables

- Only the Kafka Streams DSL has the notion of a KTable.
- KTable is an abstraction of a **changelog stream**, where each data record represents an update. More precisely, the value in a data record is interpreted as an "UPDATE" of the last value for the same record key, if any (if a corresponding key doesn't exist yet, the update will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT aka INSERT/UPDATE because any existing row with the same key is overwritten. Also, null values (tombstones) are interpreted in a special way: a record with a null value represents a "DELETE" or tombstone for the record's key.
- To illustrate, let's imagine the following two data records are being sent to the stream: **("alice", 1) --> ("alice", 3)**
- If your stream processing application were to sum the values per user, it would return 3 for alice. Why? Because the second data record would be considered an update of the previous record.

https://kafka.apache.org/documentation/streams/

KTables and Log Compaction

- Another way of thinking about KStream and KTable is as follows: If you were to store a KTable into a Kafka topic, you'd probably want to enable Kafka's log compaction feature, e.g. to save storage space.
- However, it would not be safe to enable log compaction in the case of a KStream because, as soon as log compaction would begin purging older data records of the same key, it would break the semantics of the data. E.g. you'd suddenly get a 3 for alice instead of a 4 because log compaction would have removed the ("alice", 1) data record. Hence log compaction is perfectly safe for a KTable (changelog stream) but it is a mistake for a KStream (record stream).
- Example: Change Data Capture (CDC) records in the changelog of a relational DB, representing which row in database table was inserted, updated, or deleted.
- KTable also provides an ability to look up current values of data records by keys. This table-lookup functionality is available through join operations (see also Joining in the Developer Guide) as well as through Interactive Queries.

https://kafka.apache.org/documentation/streams/

Kafka Streams DSL: GlobalKTable

- GlobalKTable is an abstraction of a **changelog** stream, where each data record represents an update.
- GlobalKTable differs from a KTable in the data that they are being populated with, i.e. which data from the underlying Kafka topic is being read into the respective table. Slightly simplified, imagine you have an input topic with 5 partitions. In your application, you want to read this topic into a table. You want to run your application across 5 application instances for maximum parallelism.
- If input topic read into a KTable, then "local" KTable instance of each application instance will be populated with data from only 1 partition of the topic 5 partitions.
- If input topic read into a GlobalKTable, then the local GlobalKTable instance of each application instance will be populated with data from all topic.
- GlobalKTable provides the ability to look up current values of data records by keys. This table-lookup functionality is available through join operations. Note that a GlobalKTable has no notion of time in contrast to a KTable.

https://kafka.apache.org/documentation/streams/

Benefits and Downsides of Using GlobalKTable

- More convenient and/or efficient joins: Notably, global tables allow you to perform star joins, they support "foreign-key" lookups (i.e., you can lookup data in the table not just by record key, but also by data in the record values), and they are more efficient when chaining multiple joins. Also, when joining against a global table, the input data does not need to be co-partitioned.
- Can be used to "broadcast" information to all the running instances of your application.

Downsides of global tables:

- Increased local storage consumption compared to the (partitioned) KTable because the entire topic is tracked.
- Increased network and Kafka broker load compared to the (partitioned) KTable because the entire topic is read.

https://kafka.apache.org/documentation/streams/

Streams DSL: Creating a Stream

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.KStream;

public class Temp {
    public static void main(String[] args) {
        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, Long> wordCounts = builder.stream(
                "word-counts-input-topic", /* input topic */
                Consumed.with(
                        Serdes.String(), /* key serde */
                        Serdes.Long()   /* value serde */
                ));
    }
}
```

Streams DSL: Creating GlobalKTable

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.utils.Bytes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.GlobalKTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.state.KeyValueStore;

public class Temp {
    public static void main(String[] args) {
        StreamsBuilder builder = new StreamsBuilder();
        GlobalKTable<String, Long> wordCounts = builder.globalTable(
                "word-counts-input-topic",
                Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as(
                        "word-counts-global-store" /* table/store name */)
                    .withKeySerde(Serdes.String()) /* key serde */
                    .withValueSerde(Serdes.Long()) /* value serde */
        );
    }
}
```

Streams DSL KStream and KTable Transformations

- KStream is an abstraction of a record stream of KeyValue pairs, i.e., each record is an independent entity/event in the real world. For example a user X might buy two items I1 and I2, and thus there might be two records <K:I1>, <K:I2> in the stream.
- A KStream is either defined from one or multiple Kafka topics that are consumed message by message, or the result of a KStream transformation.
- A KTable can also be converted into a KStream.
- A KStream can be transformed record by record, joined with another KStream, KTable, GlobalKTable, or can be aggregated into a KTable. Kafka Streams DSL can be mixed-and-matched with Processor API (PAPI) (c.f. Topology) via process(...), transform(...), and transformValues(...).

https://kafka.apache.org/documentation/streams/

Processor API (PAPI) Example - I

```
public class WordCountProcessor implements Processor<String, String, String, String> {
    private KeyValueStore<String, Long> kvStore;
    private ProcessorContext<String, String> context;

    @Override
    public void init(ProcessorContext<String, String> context) {
        this.context = context;
        kvStore = context.getStateStore("inmemory-word-counts");
    }

    @Override
    public void close() {
    }
```

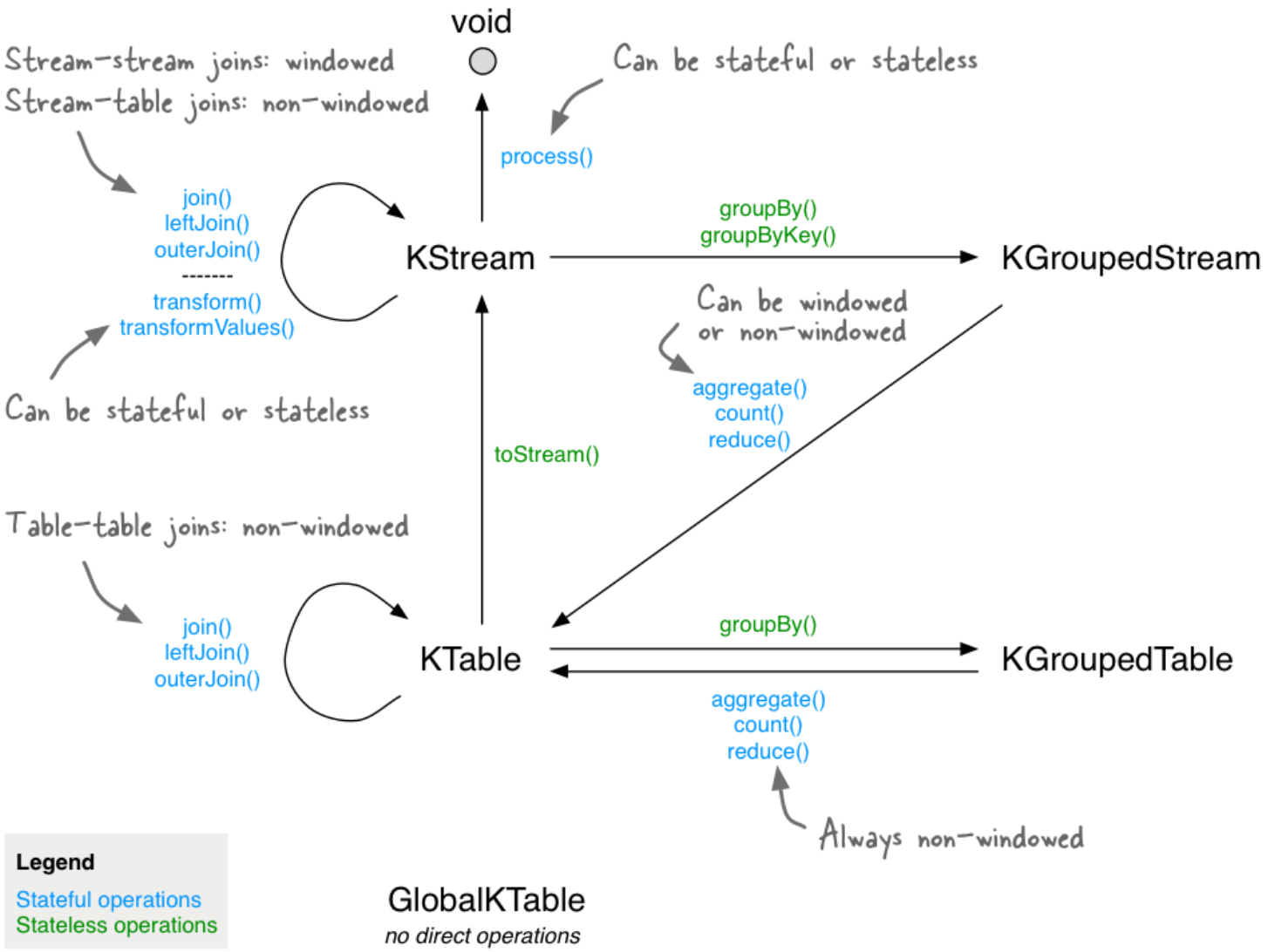
Processor API (PAPI) Example - II

```
@Override
    public void process(Record<String, String> record) {
        final String[] words = record.value().toLowerCase().split("\\W+");

        for (final String word : words) {
            Long oldVal = kvStore.get(word);
            if (oldVal == null) {
                oldVal = 0L;
            }
            kvStore.put(word, oldVal + 1);
            context.forward(new Record<>(
                    word,
                    String.format("%-15s -> %4d", word, oldVal + 1),
                    record.timestamp()
            ));
        }
    }
}
```

Streams DSL: Statefless Transformations

- Stateless transformations do not require state for processing and they do not require a state store associated with the stream processor. Kafka allows you to materialize the result from a stateless KTable transformation. This allows the result to be queried through interactive queries. To materialize a KTable, each of stateless operations can be augmented with an optional queryableStoreName argument:
- Branch: KStream → BranchedKStream
- Filter: KStream → Kstream, Filter: KTable → Ktable
- Inverse Filter filterNot: KStream → Kstream, filterNot: KTable → Ktable
- FlatMap: KStream → Kstream, FlatMap (values only): KStream → Kstream
- Foreach: KStream → void | KStream → void | KTable → void
- GroupByKey: KStream → KGroupedStream, GroupBy: KStream → KGroupedStream
- Cogroup: KGroupedStream → CogroupedKStream |CogroupedKStream → CogroupedKStream

Streams DSL: Stateful Transformations



https://kafka.apache.org/documentation/streams/

Streams DSL: Stateful Transformations

- Stateful transformations depend on state for processing inputs and producing outputs and require a state store associated with the stream processor. In aggregating operations, a windowing state store is used to collect the latest aggregation results per window. In join operations, a windowing state store is used to collect all of the records received within the defined window boundary.
- non-windowed aggregations and non-windowed KTables use TimestampedKeyValueStores
- time-windowed aggregations and KStream-KStream joins use TimestampedWindowStores
- session windowed aggregations use SessionStores (there is no timestamped session store as of now)
- State stores are fault-tolerant. In case of failure, Kafka Streams guarantees to fully restore all state stores prior to resuming the processing.

https://kafka.apache.org/documentation/streams/

Types of Stateful Transformations

Available stateful transformations in the DSL include:

- Aggregating
- Joining
- Windowing (as part of aggregations and joins)
- Applying custom processors and transformers, which may be stateful, for Processor API integration

https://kafka.apache.org/documentation/streams/

Aggregating

- After records are grouped by key via groupByKey or groupBy – and thus represented as either a KGroupedStream or a KGroupedTable, they can be aggregated via an operation such as reduce. Aggregations are key-based operations, which means that they always operate over records (notably record values) of the same key. You can perform aggregations on windowed or non-windowed data.
- Types of windows:

Window name	Behavior	Short description
Hopping time window	Time-based	Fixed-size, overlapping windows
Tumbling time window	Time-based	Fixed-size, non-overlapping, gap-less windows
Sliding time window	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps
Session window	Session-based	Dynamically-sized, non-overlapping, data-driven windows

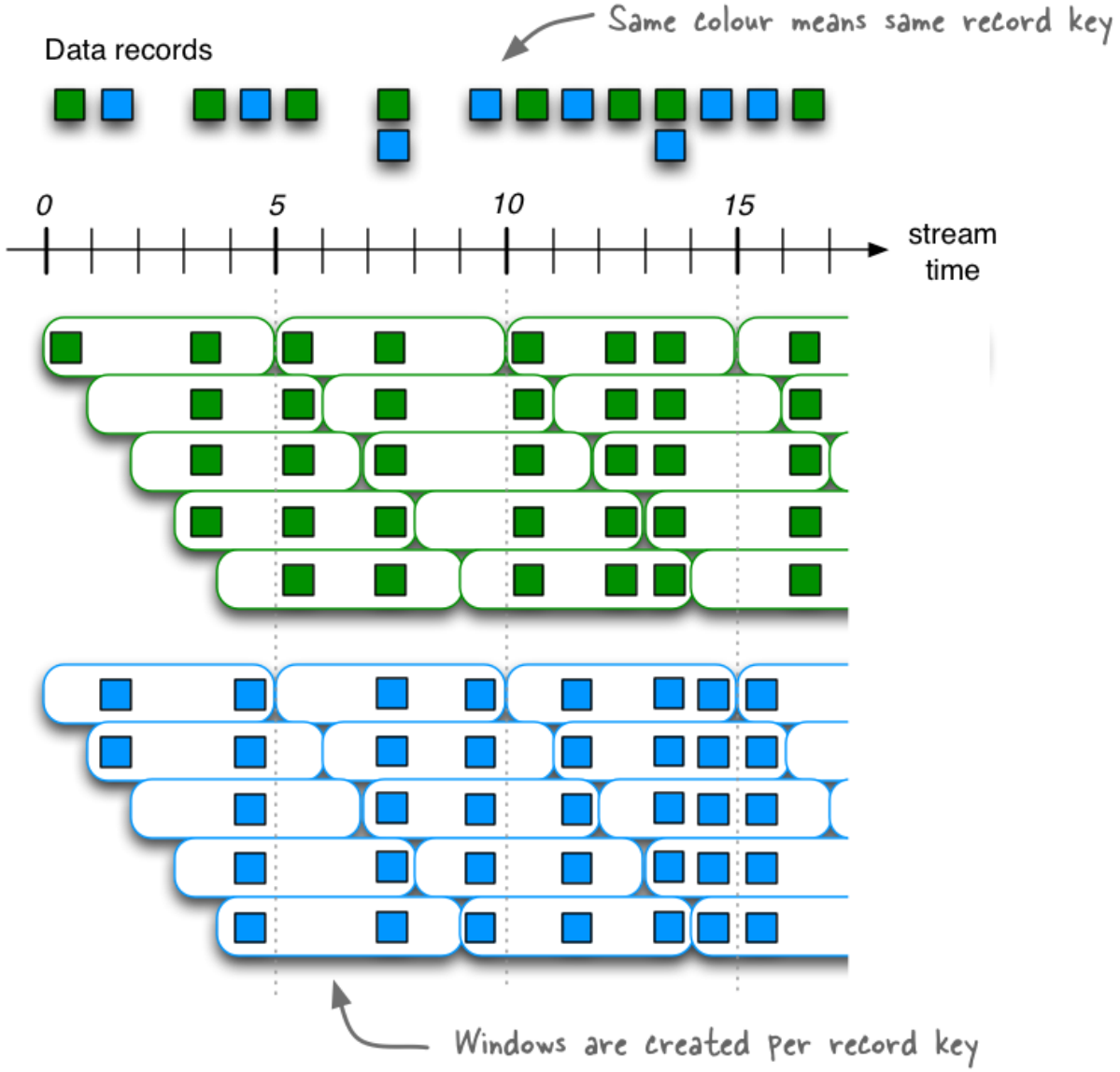
https://kafka.apache.org/documentation/streams/

Hopping Windows

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

    // A hopping time window with a size of 5 minutes and an advance interval of 1 min.
    // The window's name -- the string parameter -- is used to e.g. name the backing state store.
    Duration windowSize = Duration.ofMinutes(5);
    Duration advance = Duration.ofMinutes(1);
    TimeWindows.ofSizeWithNoGrace(windowSize).advanceBy(advance);
```

https://kafka.apache.org/documentation/streams/

A 5-min Hopping Window with a 1-min "hop"

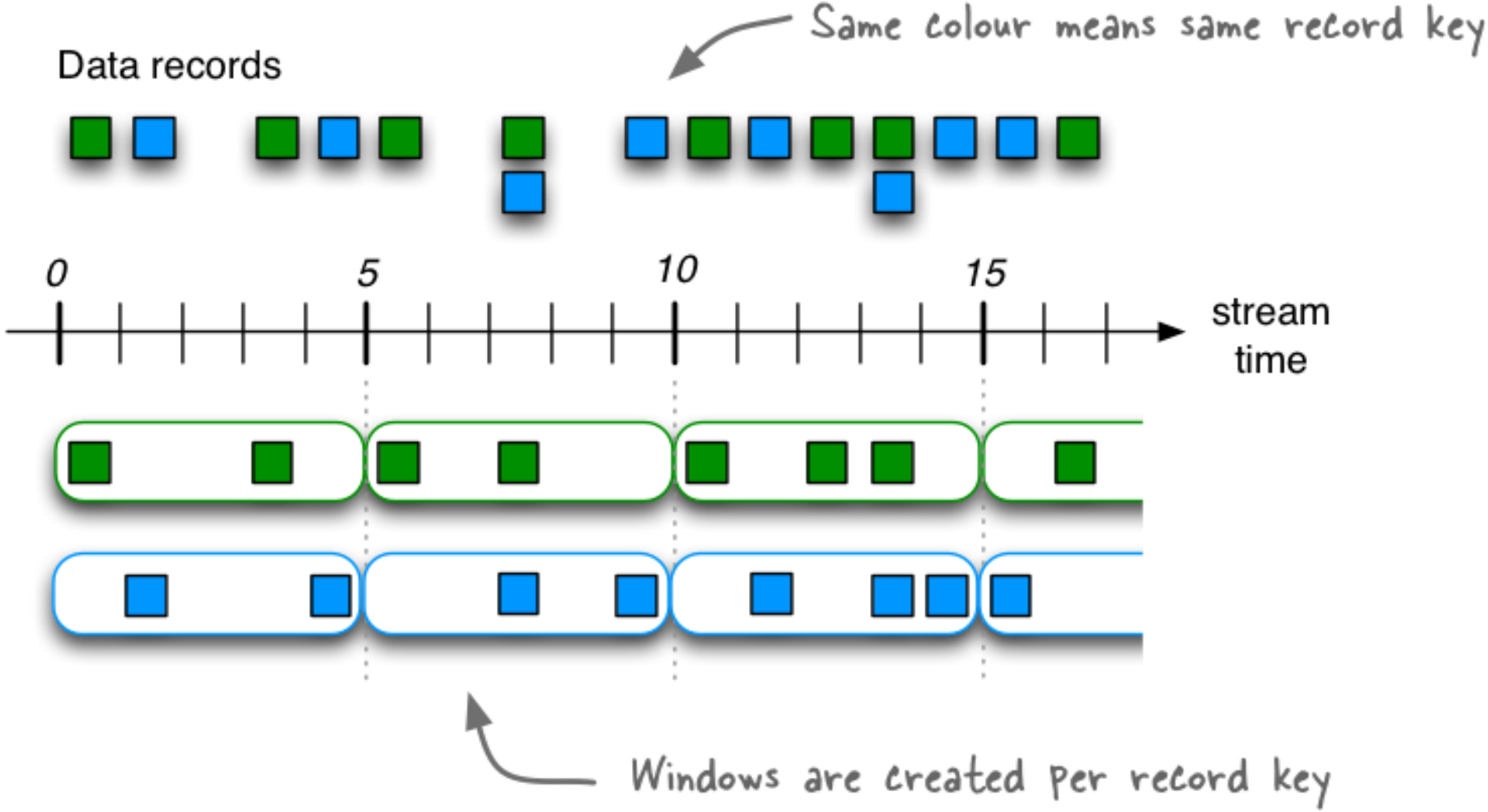


https://kafka.apache.org/documentation/streams/

Tumbling Time Windows

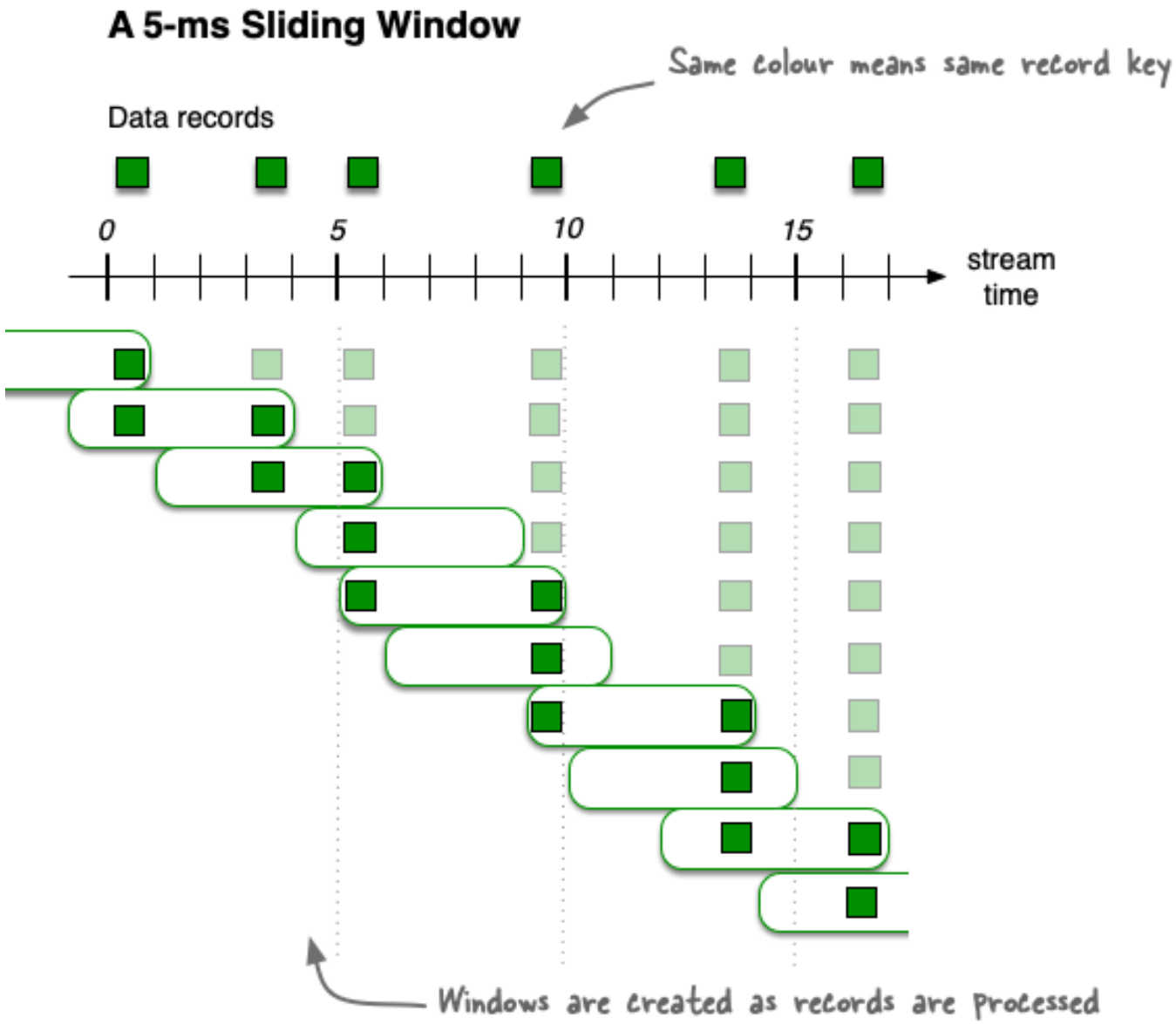
```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

// A tumbling time window with a size of 5 minutes (and, by definition, an implicit
// advance interval of 5 minutes), and grace period of 1 minute.
Duration windowSize = Duration.ofMinutes(5);
Duration gracePeriod = Duration.ofMinutes(1);
TimeWindows.ofSizeAndGrace(windowSize, gracePeriod);

// The above is equivalent to the following code:
TimeWindows.ofSizeAndGrace(windowSize, gracePeriod).advanceBy(windowSize);
```

A 5-min Tumbling Window



Sliding Time Windows

```
import org.apache.kafka.streams.kstream.SlidingWindows;

// A sliding time window with a time difference of 10 minutes and grace period of 30 minutes
Duration timeDifference = Duration.ofMinutes(10);
Duration gracePeriod = Duration.ofMinutes(30);
SlidingWindows.ofTimeDifferenceAndGrace(timeDifference, gracePeriod);
```



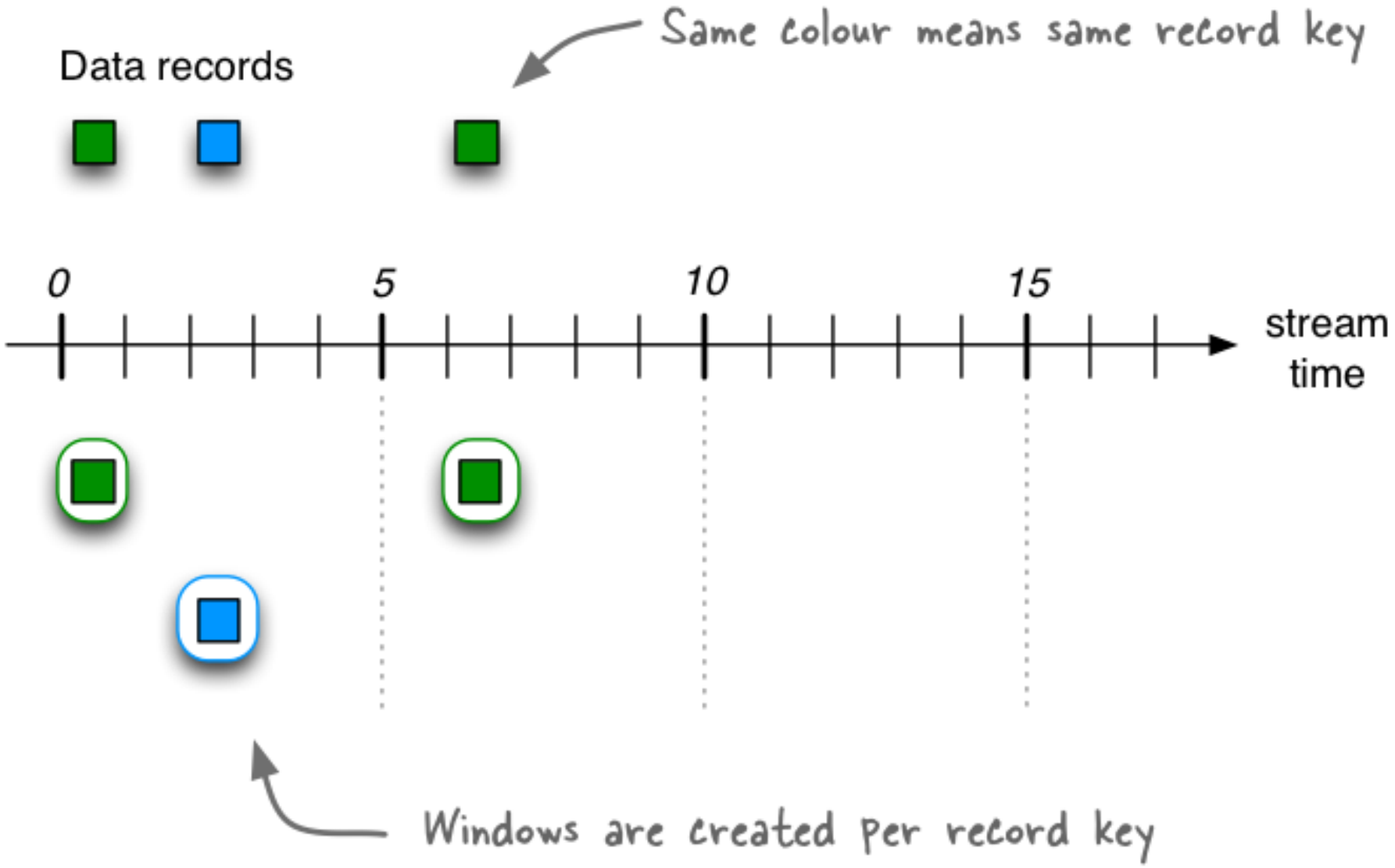
https://kafka.apache.org/documentation/streams/

Session Windows

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.SessionWindows;

// A session window with an inactivity gap of 5 minutes.
SessionWindows.ofInactivityGapWithNoGrace(Duration.ofMinutes(5));
```

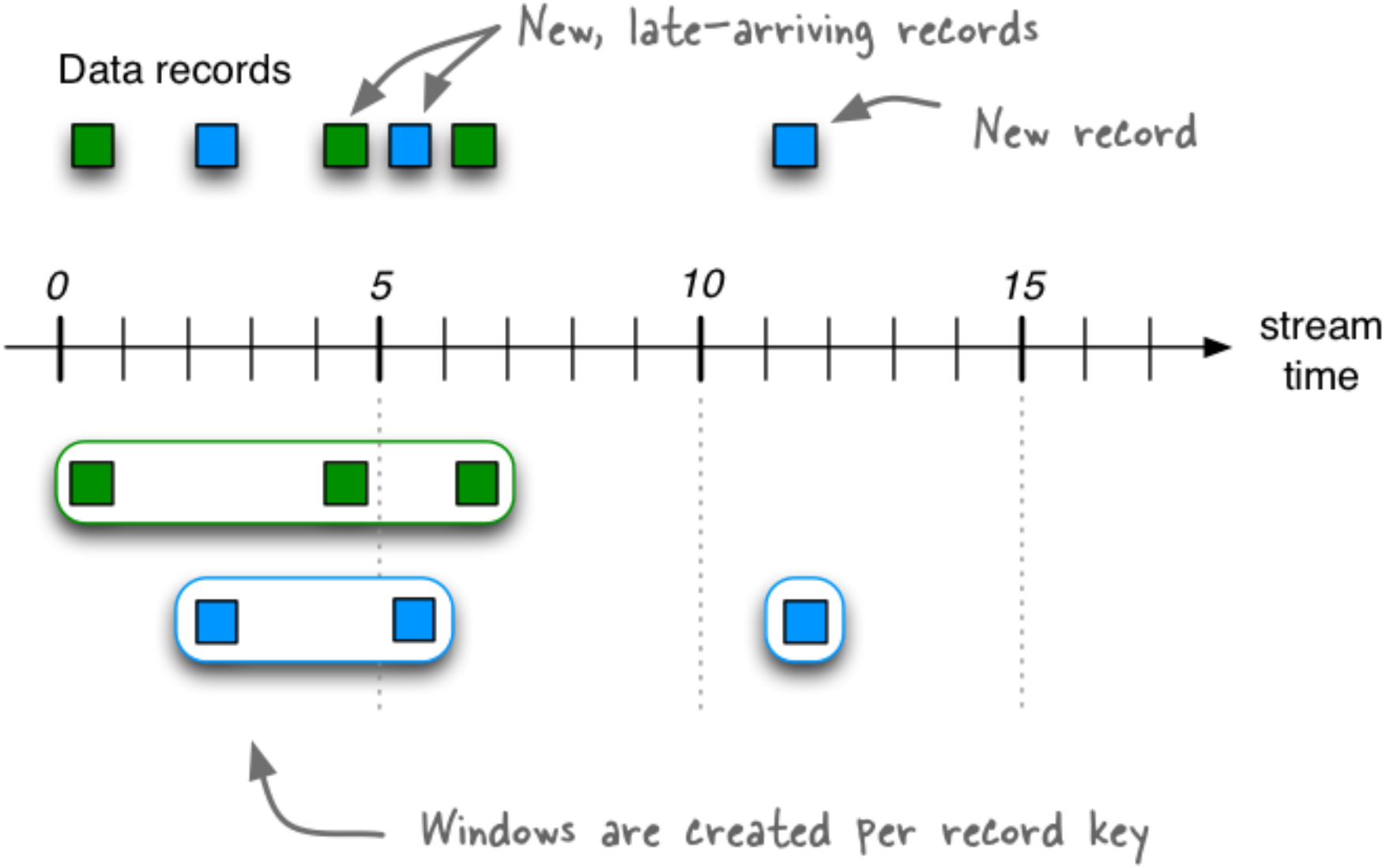
https://kafka.apache.org/documentation/streams/

A Session Window with a 5-min inactivity gap



https://kafka.apache.org/documentation/streams/

A Session Window with a 5-min inactivity gap



Joining

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
KStream-to-KStream	Windowed	Supported	Supported	Supported
KTable-to-KTable	Non-windowed	Supported	Supported	Supported
KTable-to-KTable Foreign-Key Join	Non-windowed	Supported	Supported	Not Supported
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported

https://kafka.apache.org/documentation/streams/

Join Co-partitioning Requirements

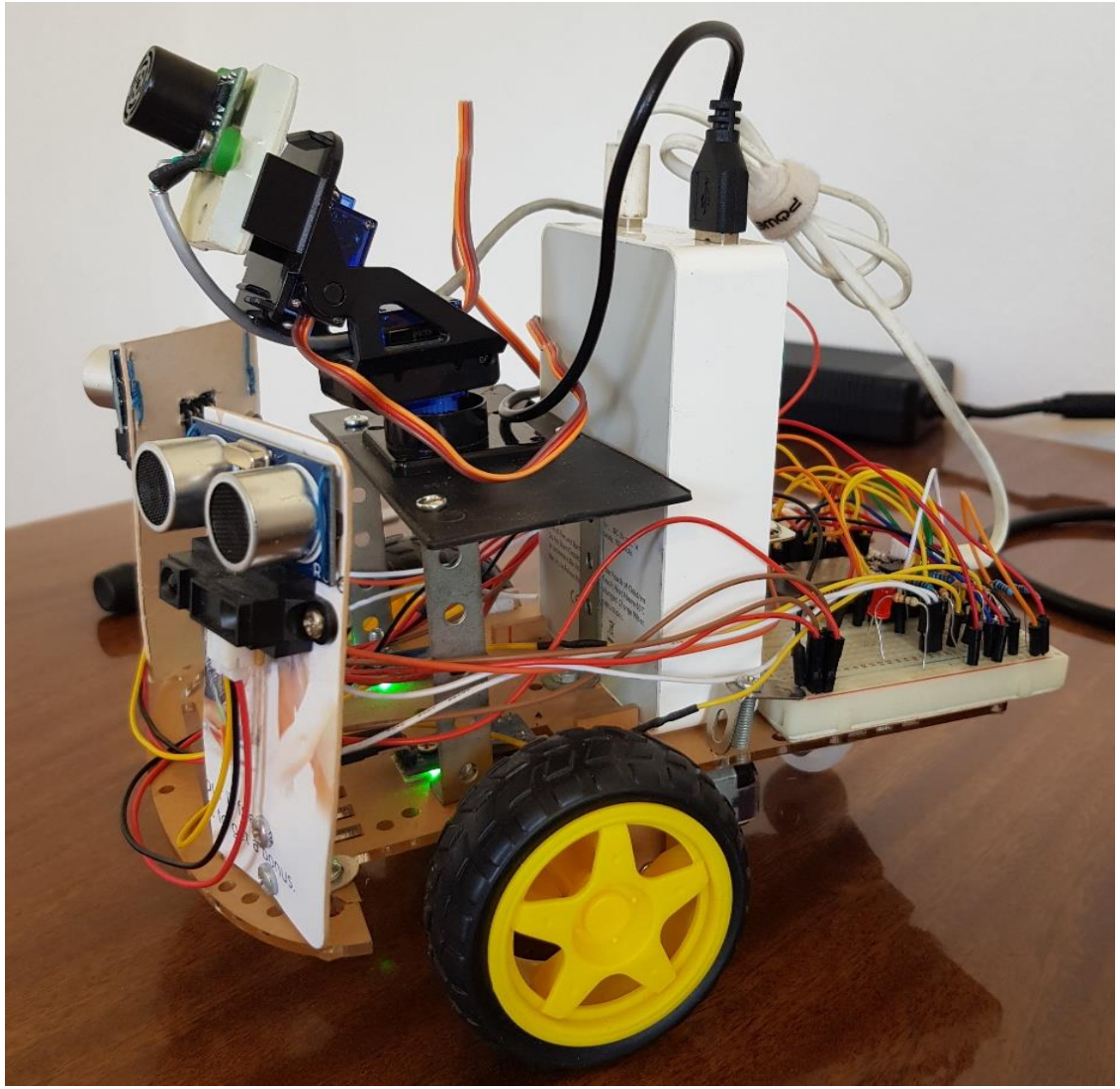
- For equi-joins, input data must be co-partitioned when joining. This ensures that input records with the same key from both sides of the join, are delivered to the same stream task during processing.
- Co-partitioning is not required when performing KTable-KTable Foreign-Key joins and Global KTable joins.
- The input topics of the join (left side and right side) must have the same number of partitions.
- All applications that write to the input topics must have the same partitioning strategy so that records with the same key are delivered to same partition number. In other words, the keyspace of the input data must be distributed across partitions in the same manner.

Join Co-partitioning Requirements - II

- Why is data co-partitioning required? Because KStream-KStream, KTable-KTable, and KStream-KTable joins are performed based on the keys of records (e.g., leftRecord.key == rightRecord.key), it is required that the input streams/tables of a join are co-partitioned by key.
- There are two exceptions where co-partitioning is not required. For KStream-GlobalKTable joins, co-partitioning is not required because all partitions of the GlobalKTable's underlying changelog stream are made available to each KafkaStreams instance. That is, each instance has a full copy of the changelog stream. Further, a KeyValueMapper allows for non-key based joins from the KStream to the GlobalKTable. KTable-KTable Foreign-Key joins also do not require co-partitioning. Kafka Streams internally ensures co-partitioning for Foreign-Key joins.

https://kafka.apache.org/documentation/streams/

Demos



Available @ Github:

https://github.com/iproduct/kafka-streams-javaland

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