Concursus: Event Sourcing for the Internet of Things

OpenCredo Ltd* Dominic Fox† Tareq Abedrabbo‡

May 10, 2016

Abstract

Keywords: event sourcing, CQRS, stream processing, microservices, internet of things, Java 8, Cassandra, RabbitMQ, Kafka.

We present Concursus, a framework for developing distributed applications using CQRS and event sourcing patterns within a modern, Java 8-centric, programming model. Following a high-level survey of the trends leading towards adoption of these patterns, we show how Concursus simplifies the task of programming event sourcing applications by providing a concise, intuitive API to systems composed of event processing middleware. We provide a brief account of a distributed, microservice-based architecture which we successfully implemented using these techniques. We then discuss the scalability, reliability and fault-tolerance characteristics an event system should have, and how Concursus supports building systems with these characteristics. Finally we indicate some future directions in event sourcing and stream processing technology, and suggest how Concursus can be integrated with emerging technologies such as Apache Kafka.

1 From the Internet of Users to the Internet of Things

Services for the "internet of users" (or the "world-wide web") are typically characterised by request/response patterns of interaction, serialised and contextualised within a "session". Somebody sits down at a computer, opens a web browser, logs in to a service and performs a series of operations, waiting for one operation to complete successfully before beginning the next. A paradigmatic application of this kind is filling a shopping trolley and completing an order. There is a processing context, the state of the trolley and/or the order, that is modified by each interaction, and carried forward from one interaction to the next. The major architectural challenge in implementing this kind of system is maintaining the links between users, sessions and session data in a scalable and

*OpenCredo Ltd, 5-11 Lavington St, London SE1 0NZ
†dominic.fox@opencredo.com
‡tareq.abedrabbo@opencredo.com

reliable way, so as to uphold the user's illusion that they are engaged in a series of transactions with a single respondent who remembers who they are and what they've done so far, rather than a load-balanced cluster of virtual machines any of which might be shut down without notice at any moment.

1.1 Asynchronous and Message-Driven Architectures

Some more modern web applications incorporate mechanisms for push-notification, so that the logged-in user can receive alerts about events that take place within a shared context: a chat room, or a network of users publishing and subscribing to each other's updates. The request/response interaction pattern no longer predominates in this environment. I upload a photo to an image-sharing site, and expect that my followers will be able to see it sooner or later, but I do not have to wait for notification that every one of my followers has been notified that it exists. I observe their "likes" and comments on my photo intermittently as they occur. Although their underlying means of interaction with the system is still HTTP request/response pairs, users of social media sites are behaving more like participants in a message queue-based architecture, where decoupled, asynchronous messaging is the norm.

1.2 One universe, many worlds

We are now starting to see a new style of application, often (although not always) associated with the slogan "The Internet of Things" (IoT). This style of application is characterised by a much higher number of participants, and much more extreme decoupling, to the point where the metaphor of a "shared context" starts to break down. The "things" are not conceptualised as being in a "room" together, or even as participating in a common "social network". They transmit information about their status, and receive notifications telling them how to behave, but the co-ordinating mechanisms which connect things to other things, and compose coherent stories about their interactions, are hidden from them.

In an IoT-style application there is a separation between the mechanisms of communication and the mechanisms of co-ordination. A message queue-based architecture decouples message producers from message consumers, but exposes the co-ordinating abstractions - queues, topics and exchanges - within the communication layer. Often these abstractions provide the metaphors in terms of which the whole system is defined and understood: there is one "world" - one topology - to which everything belongs. In an IoT scenario, communication is often brutally simplified: a network-enabled lightbulb broadcasts telemetry data and receives instructions on hue and brightness; a motion sensor transmits a binary flag indicating whether it thinks there is anybody in the room. The management of a household's mood lighting and power consumption is the responsibility of a separate co-ordinating service that must consume data from multiple sources and make decisions about what is to be done. One such service switches the lights off when a room has been empty for a short while; another, responsible for household security, switches on a camera and sends an SMS to the homeowner when a room that is supposed to be empty appears not to be (see

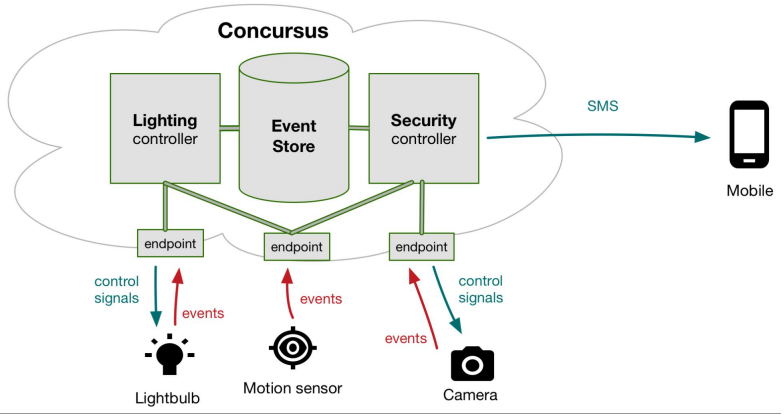

Figure 1: Example IoT scenario

Figure 1). These two services co-ordinate different sets of devices in different ways: they compose different "worlds" out of the same atoms.

2 An Architecture for the Internet of Things

The architecture of IoT-style applications is not *radically* different from that of distributed systems co-ordinated via message queues, but it is one in which the message queue "topic exchange" metaphor is no longer appropriate to describe the overall way the system is organised (even if message queues are still used pervasively as a mechanism within it). The layer of the system that is concerned with collecting data, and dispatching updates to devices, becomes increasingly decoupled from the layer which is concerned with analysing data, composing coherent views of the world, and making decisions. When we want to query the system to find out something about its state, we will often end up addressing our queries to a particular subsystem's view of its own domain. A truly global view of the system - its total state at any given time - may not be immediately available, and might be laborious to calculate. We will often have to make do with approximations.

2.1 Microservices and Domain Driven Design

There are two existing architectural trends that feed into the IoT model. The first is the shift towards "microservices" and "domain-driven design" (NOTE: Eric Evans (2004). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.), which are design philosophies that hold that different areas of business functionality should be separated out into functionally-decoupled "bounded contexts" which manage their own data. In a microservices architecture there is no canonical global data model (such as might be represented by a single very large relational database schema), but rather a collection of models each of which represents common entities (such as users, or inventory items) to itself in its own distinctive way. Here there is a distinction

between the global concept "user" as it is expressed in the "shared language" of the system, and the concept "user-for-X" as it is expressed in the domain of each bounded context "X". Two contexts wishing to communicate with each other about some particular user may have only that user's *identity* in common: their internal representations of what a user is *for them* may be totally different. For example, in the access control domain a user is someone who has credentials which must be verified and permissions which may be granted or withheld; in the profile management domain, a user is someone with a nickname, a profile picture, biographical information and a list of culinary preferences. If we wanted to compose a "complete" view of the user, we would need to consult both of these domains and "glue" their representations of the user together somehow.

2.2 CQRS

The second trend is the shift towards Command/Query Responsibility Segregation (CQRS) patterns, in which responsibility for updating the state of the system (or a particular bounded context) is separated from responsibility for providing a queryable view of that state. The key insight behind CQRS patterns is that reads (queries) and writes (commands) often have different scalability, consistency and reliability requirements. For example, it is often acceptable to provide a fast, cached view of frequently-queried entities in the system, optimised for the most common query patterns, which is not immediately updated when a write is performed. We want writes to be reliable, with dependable processing guarantees; we want reads to be fast, and to provide a consistent-enough view of the data for the client's purposes. At a deeper level, CQRS patterns decouple the semantics of state *changes* (represented by commands) from internal state *representation*: unlike ORM-mapped database CRUD operations, where we retrieve a representation of an entity from the "persistence layer", modify it and then save it back again, a "command" is something more akin to a database stored procedure, which may have as its outcome the modification of *multiple* query-optimised views of the state it addresses.

Both microservices architecture and CQRS patterns encourage an "ontological pluralism"[1] in which there is no globally transparent model of the entire system, but rather a range of overlapping *projections* of system state, which may have varying consistency requirements (e.g. "eventual consistency") but are typically not immediately synchronisable into a coherent global view. In the IoT model, we add to this plurality of representations a *stream processing* element, in which the business logic of the system is applied to aggregates of data from many sources, downstream from where that data was collected. What was first separated, distributed and partitioned for scalability, flows back together in stream processing.

In Concursus we have drawn together some of these threads into an opinionated framework for building distributed applications that use CQRS and event sourcing patterns, building on research by Google[2] and others[3], and our own practical experience in delivering systems that must scale to handle large volumes of events (of the order of millions per day) from many sources.

3 From Relational Modelling to Event Sourcing

Concursus uses an *event sourcing*[4] data model, in which append-only event histories replace mutable records as the fundamental structure of persistent data. This has a number of consequences, which we will briefly survey here. In a traditional RDBMS-backed application, the current state of the system is represented by a collection of "entities" whose relationships to each other are managed through referential integrity constraints enforced by the database. There is a single, global model of the application's data, defined through database schemas and integrity constraints, which prescribes all of the states it is possible for it to be in, and all of the state transitions which are permitted to occur. Transactions are expected to arrive in order and to move the system from one consistent overall state to another, and it should not be possible to commit any transaction that leaves the system in an inconsistent state.

3.1 Bounded contexts and distributed state

As systems become more distributed, so their "current state" becomes less immediately available, as we might have to query multiple data stores in order to ascertain it, and the mechanisms for ensuring consistency in global state changes become more complex. In a highly decoupled message queue-based system, "distributed transactions" are both difficult to implement, and introduce a synchronisation overhead which hampers scalability. It may then become expedient to relax some of the consistency guarantees that were provided by the RDBMS-centric approach. Local bounded contexts may still enforce referential integrity constraints within their own boundaries, but these mechanisms are no longer expected to be globally applicable. It is sometimes necessary to perform reconciliation activities to ensure that remote parts of the system have not fallen into conflicting states.

3.2 The event log as source of truth

The event sourcing approach brings this movement away from global consistency to its extreme limit. In an event sourcing system, the primary "source of truth" is a log of events that have occurred within the system, similar to the transaction log of an RDBMS. Whereas it is common to add "audit tables" to RDBMS systems to track the history of changes to key business entities, in an event sourcing system the history of changes *is* the content of the core persistence layer. Every entity (or "aggregate root", in DDD terminology) has its own recorded event history, and in order to consult the state of an entity at a particular moment in time we must "roll up" the entity's event history up to that moment into a representation of its state at the end of the rolled-up sequence of events. We may cache a "snapshot" of this computed state in order to save repeatedly recalculating it, but in general we will always have to do *some* work to bring our cached representation fully "up-to-date" by replaying any subsequent transactions against it.

3.3 Correctness and Consistency

How, given this approach, do we ensure that a given transaction is valid - that it obeys referential integrity constraints, and does not result in an incoherent representation? The short answer is that an event sourcing persistence layer provides *no* mechanism for doing this, since the only way to know about the contemporary state of related entities at the point where a transaction is submitted is to compute it (or to retrieve, and bring up-to-date, a cached representation). Worse still, there may be no global ordering of events available: while each entity has its own totally ordered event history, it will not in general be possible to determine which of two events occurring to two different entities with the same timestamp happened "first" (especially as event timestamps may themselves be issued by multiple sources which may not be perfectly synchronised).

Suppose for example that a user is created and simultaneously added to a user-group. The creation of the user is recorded in the user's event history, and the addition of the user to the group is recorded in the group's event history. When we come to replay the events for these two entities, if the "user added" event is replayed before the "user created" event, we will be adding to the group a user that does not "yet" exist. Once both histories have been replayed in their entirety, we can check to see whether everything is consistent; but this is a reconciliation activity carried out after the fact, similar to correcting accounting entries, rather than a validation step in which we decide whether or not to accept the "user added" event to begin with.

This situation mirrors the "Internet of Things" scenario discussed above: from the point of view of an event sourcing system, each entity exists in its own silo, rather than being situated in a model in which it is explicitly related to other things through foreign keys and other constraints. The task of co-ordinating entities, and building up a consistent picture of the state of a group of entities gathered together within the same domain, is carried out at a higher level - or, from a stream-processing perspective, *downstream* from the collection and persistence of the system's primary data, its event logs.

3.4 Write First, Reason Later

What are the advantages of this approach? It is especially well-suited to a scenario in which data enters the system from a very large number of discrete sources, such as IoT devices, and we cannot afford the overhead of linearisation of a global data model. It enables us to "write first, reason later", in a highly scalable fashion: since each write is to the append-only log associated with a particular entity, writes can readily be partitioned by entity id and distributed among a cluster of practically unlimited size. If we need to construct local models for reasoning, such as an RDBMS containing a view of the current state of all of the entities in a particular domain, we can treat these models as transient, since they can always be rebuilt from scratch by replaying stored events into them, and we can replicate updates into multiple copies using standard log replication techniques. Finally, we can replay event histories into stream processing systems to generate both real-time and bulk analytics, treating the event store as a universal *buffer* for stream processing.

There is thus a strong affinity between the event sourcing model and the IoT universe discussed earlier: its apparent weaknesses, from an RDBMS perspective, turn out to be strengths when applied to a situation in which it is no longer feasible to manage everything that happens within the system in a linear fashion, governed by a single global model. Moving to an event sourcing approach unlocks precisely the architectural patterns that are needed to build highly-scalable systems.

4 The Concursus Programming Model

4.1 Emitting Events

The first thing we need, if we are to use event sourcing patterns in our applications, is a way to emit events. Concursus defines an event as a combination of metadata, which is used to index and organise events into discrete event histories, and event data, which captures the details of what happened. For the metadata, we need the following:

- An event timestamp, which states when the event occurred.
- A globally unique aggregate id, which states which entity the event occurred to.
- An event type, which states what kind of event it was.

By following a set of conventions, we can provide all of this information, together with the event data, in a single Java method call. The timestamp and aggregate id are provided as the first two parameters of the method call, the event type is derived from the method name (or can be overridden by an annotation on the method), and the event data is derived from any remaining method parameters. The following interface thus defines a collection of events that can occur to lightbulbs:

```
@HandlesEventsFor(lightbulb)
public interface LightbulbEvents {
  @Initial
  void created(StreamTimestamp timestamp, String id, int wattage);
  void screwedIn(StreamTimestamp timestamp, String id, String
      location);
  void switchedOn(StreamTimestamp timestamp, String id);
  void switchedOff(StreamTimestamp timestamp, String id);
  void unscrewed(StreamTimestamp timestamp, String id);
  @Terminal
  void blown(StreamTimestamp timestamp, String id);
}
```

A `StreamTimestamp` is a combination of a millisecond-resolution timestamp (a Java 8 `Instant`) and a stream id which is provided in case multiple events affecting the same aggregate occur within the same millisecond time interval, in which case they are conceptualised as occurring within separate streams of

events within the same history. This interface thus defines a lightbulb as something which can be created with an initial specification of wattage, screwed in and unscrewed from various locations, switched on and off, and finally blown.

Concursus can now create a Java dynamic proxy which implements this interface, generates `Event`s on method calls, and passes them on to an event handler of some kind. Let's start by simply writing a `String` representation of each `Event` to the console:

```
LightbulbEvents events = EventEmittingProxy.proxying(System.out::
    println, LightbulbEvents.class);
String lightbulbId = UUID.randomUUID().toString();
StreamTimestamp start = StreamTimestamp.now("stream_a");
events.created(start, lightbulbId, 60);
events.screwedIn(start.plus(1, MINUTES), lightbulbId, hallway);
events.switchedOn(start.plus(2, MINUTES), lightbulbId);
```

This will output the following sequence of event representations:

```
lightbulb:254ddc61-abcc-49aa-9837-b3995e888979 created_0
at 2016-04-06T14:02:25.191Z/stream_a
with lightbulb/created_0{wattage=60}
lightbulb:254ddc61-abcc-49aa-9837-b3995e888979 screwedIn_0
at 2016-04-06T14:03:25.191Z/stream_a
with lightbulb/screwedIn_0{location=hallway}
lightbulb:254ddc61-abcc-49aa-9837-b3995e888979 switchedOn_0
at 2016-04-06T14:04:25.191Z/stream_a
with lightbulb/switchedOn_0{}
```

The first line contains the aggregate type ("lightbulb") and id ("254ddc61-abcc-49aa-9837-b3995e888979", the String UUID we supplied as the second method parameter), followed by the event name. Event names are versioned in Concursus, and begin at version 0, hence the method created emits an event with the name "created_0". The second line contains the stream timestamp we supplied as the second method parameter, and the final line contains a "named tuple" containing the event data for each event.

4.2 Replaying Events

Suppose we have a collection of Events we would like to replay to a handler implementing the `LightbulbEvents` interface. This can be done using a `DispatchingEventOutChannel`:

```
public void replayToHandler(List collectedEvents, LightbulbEvents
    handler) {
  Consumer eventConsumer = DispatchingEventOutChannel.toHandler(
      LightbulbEvents.class, handler);
  collectedEvents.forEach(eventConsumer);
}
```

Between them, the `EventEmittingProxy` and `DispatchingEventOutChannel` convert method calls into `Event` objects and `Event` objects back into method calls. From the point of view of the client programmer, this is nearly all there is to Concursus: we use method calls on proxy objects to play events into the system, and replay stored events to event handlers implementing the same interfaces.

4.3 Event-handling Middleware

Almost everything else is the responsibility of *event-handling middleware*, which takes care of such things as:

- Writing batches of events into a persistent event log, e.g. in Redis or Cassandra.
- Filtering event batches to remove events with duplicate aggregate id/event timestamp combinations, to ensure idempotency.
- Publishing events out to message queues once they have been persistently logged.
- Building indexes linking aggregate ids to event data, for more flexible querying.
- Serialising events to JSON and sending them via HTTP to a remote endpoint, or writing them to a Kafka topic for downstream processing.

In many cases, all a class needs to do in order to function as event-handling middleware is to implement `Consumer<Event>`. Concursus includes a range of components providing the functionality listed above, along with Spring Bean definitions that make it easy to wire together an event-sourcing application using Spring's dependency injection (see Figure 2).

4.4 Command Processing

Concursus also provides support for distributed command processing, using a similar mechanism. `Command`s are defined via interfaces in a similar fashion to `Event`s; the major difference is that a `Command` method may have a return value, and may fail (barring failure of a middleware component such as an event log, emitting an event will always succeed):

```
@HandlesCommandsFor("person")
public interface PersonCommands {
  Person create(StreamTimestamp ts, String personId, String name,
      LocalDate dob);
  Person changeName(StreamTimestamp ts, String personId, String
      newName);
  Person moveToAddress(StreamTimestamp ts, String personId, String
      addressId);
  void delete(StreamTimestamp ts, String personId);
}
```

A client will *issue* a command, which is routed to a command processor which checks it for validity and, if successful, *emits* events representing the outcome. Semantically, a command is an imperative, a "do this!", while an event is an assertion that something has been done; by convention, commands will have names like "create" and "delete" while events will have names like "created" and "deleted".

An important feature of Concursus's command processing is the ability to route commands to separate processors based on the ids of the aggregates to which

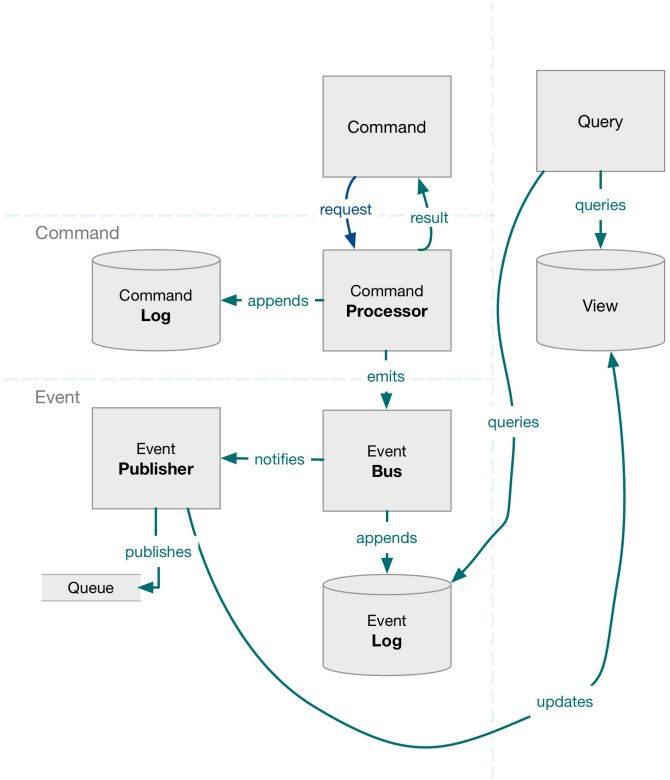

Figure 2: Illustration of a CQRS architecture using the Concursus Command Processor, Command Log, Event Bus, Event Log and Event Publisher.

they are addressed, effectively sharding execution. If each processor runs single-threaded, then this is a cheap way of ensuring that no two commands addressed to the same aggregate will ever be executed simultaneously. A Hazelcast implementation of Concursus's `CommandExecutor` interface enables this behaviour to be distributed among a cluster of processors.

4.5 State modelling

Noticeably absent from the core Concursus programming model is domain classes representing aggregates. We can process a command or emit an event without having an object in hand representing the affected aggregate, which means that we can process requests that arrive out-of-order: for example, the deletion of an item followed by its creation. Provided the *event timestamps* on the events place them in the correct order, or we can apply a causal ordering to the event history which re-orders them sensibly, the processing order is not necessarily significant. If we insist on retrieving a representation of the aggregate and testing that it is in a valid state for a command to be executed against it before emitting events, then we will impose the linearity of method execution in object-oriented programming (i.e. you must call the constructor on a class before you can call a method on the resulting instance) on our processing model. In some cases this

is desirable, but the problem with tying command and event processing into domain classes is that it makes it mandatory.

In some cases, however, we will want to enforce a correct linear sequence of actions at the command level, such that commands will fail if they would emit events that would be inconsistent with the currently-recorded event history for an aggregate. In the `PersonCommands` example above, some of the command methods return a `Person` - that is, if successful, they return a representation of the state of the aggregate following command execution. It is useful in these cases to have an easy way to "roll up" the event history of an aggregate into such a representation. In Concursus this is done by providing a "state class" with static factory methods mapped to "initial" events (annotated with `@Initial` in the event-defining interface) and instance methods mapped to all subsequent events. Here is an example state class for a lightbulb:

```
@HandlesEventsFor("lightbulb")
public static final class LightbulbState {

  @HandlesEvent
  public static LightbulbState created(String id, int wattage) {
    return new LightbulbState(id, wattage);
  }

  private final String id;
  private final int wattage;
  private Optional screwedInLocation = Optional.empty();
  private boolean switchedOn = false;
  public LightbulbState(String id, int wattage) {
    this.id = id;
    this.wattage = wattage;
  }

  @HandlesEvent
  public void screwedIn(String location) {
    screwedInLocation = Optional.of(location);
  }

  @HandlesEvent
  public void unscrewed() {
    screwedInLocation = Optional.empty();
  }

  @HandlesEvent
  public void switchedOn() {
    switchedOn = true;
  }

  @HandlesEvent
  public void switchedOff() {
    switchedOn = false;
  }

  // getters
}
```

Events can be replayed to this class, creating an instance with the factory method and then mutating it with the instance methods until its state represents the "current" state of the lightbulb, based on the event history supplied to it. Multiple state classes can be created for the same aggregate type, rep-

resenting different aspects of its changing state over time that we might be interested in.

A `StateRepository` class is provided which supplies an API for retrieving the event history for an aggregate and replaying it into a state class. Here is an example of it in use, in a method implementing the `switchOn` command for a lightbulb:

```
public LightbulbState switchOn(StreamTimestamp ts, String
    lightbulbId) {
  LightbulbState lightbulb = lightbulbStateRepository
        .getState(lightbulbId)
    .orElseThrow(NoSuchLightbulbException::new);
  eventBus.updating(lightbulb, bus -> {
    bus.dispatch(LightbulbEvents.class, e ->
      e.switchedOn(ts, lightbulbId));
  });
  return lightbulb;
}
```

The eventBus is a component which enables batches of events to be generated and dispatched collectively. Rather than directly calling methods on the retrieved `LightbulbState` instance to modify it, we issue events to the event bus, instructing it to route those events to the state class instance, updating it, as well as to downstream processing (e.g. a persistent event log). We then return the updated instance to the caller.

By modelling the event history of an aggregate as a series of transitions in a state machine, and providing a mechanism for replaying events into a class representing that state machine, Concursus models the *behaviour* of aggregates rather than simply collecting the most recent values for properties (as in the Kafka Streams table model). A state class is not merely a "bean-like" POJO, but a means of checking the validity of a sequence of events against a model of permitted transitions and their side-effects.

5 Building a Distributed Concursus System

Concursus is based on our experience of building practical, performant applications to process data at scale within a microservices architecture. In situations where the volume of data was greater than a traditional architecture could deal with, we found ourselves gravitating towards microservices supported by a distributed CQRS, event-sourced domain model. There were several components that we found fitted well together when building solutions of this kind.

The first was Spring Boot, as a standard and convenient platform for microservice development and deployment. This led us to make Spring integration (via the concursus-spring module) a priority, as Spring's Java configuration and dependency injection can greatly simplify the task of wiring together the cooperating pieces of command and event-handling middleware that make up a complete CQRS system. For example, filters can be introduced which observe or intercept events being written to the event log, simply by annotating classes with @Filter and ensuring they are visible to Spring's component scanning.

The second was RabbitMQ, as a transport for "integration events" broadcast from one service to other services in the system. Once an event has been written to the event log, an event publisher pushes it to in-process event handlers which take further action such as updating a cache or writing a message into a queue. This then enables out-of-process subscribers, such as other microservices subscribed to the queue, to respond. We considered Kafka as an alternative notification mechanism, and found that it enabled several other interesting patterns, such as combining durable event logging and publication in a single mechanism, with maintaining a queryable event store and other views configured as downstream processing tasks.

Finally, we found that Cassandra was ideal as a reliable and scalable event store, partitioning events by aggregate id and clustering and ordering them by timestamp so that the persistent data model resembled a "wide and shallow" collection of ordered event histories. The choice of Cassandra involves some trade-offs. On the down-side, querying is limited to retrieving the event histories for one or more aggregates by id, and any further indexing requires additional code and tables to implement. On the up-side, both reads and writes are highly scalable, and a single standard table definition suffices for storage of events of all kinds across the system.

6 Scalability, Reliability and Fault-Tolerance

A Concursus application can be seen simply as a collection of microservices communicating through messaging middleware. Therefore, the same good scalability practices that apply to microservices can also be applied to Concursus services. Most importantly, each service should minimise local state as much as possible and should avoid strong locking around shared resources as much as possible. This is not always easy to achieve, as there are common situations where some state need to be readily available, for example to validate incoming requests synchronously against a stateful view, or where locking is need to maintain consistency, such as ordering multiple events on the same entity.

We found that using an in-memory grid, such as Hazelcast, helps us solve these problems in a coherent and elegant way. Using distributed collections enables us to maintain any number of shared views in a way that is close to the code, thus minimising the overhead of a fully-fledged database. The ability to leverage Hazelcast's natural partitioning capabilities and to dispatch computation to the data using Entry Processors has been very useful to minimise locking and avoid bottlenecks.

A Concursus application naturally inherits the processing guarantees of the underlying messaging middleware. For example, using RabbitMQ or Kafka, at-least-once processing can be achieved because the underlying middleware will redeliver a message until it is acknowledged by a consumer. At-least-once processing can be good enough in situations where we accepted that a computed result can be slightly off, or can be corrected at later point.

Often though, exactly-once processing semantics are highly desirable or even required. One example is writing to the event log, which must only maintain exactly one copy of each event to guarantee consistency of the event sourced model.

This can be achieved by combining at-least-once processing with idempotent writes to the event log datastore (Cassandra). Because events are immutable and have stable keys, rewriting the same event is an idempotent operation.

In some situations, exactly-once processing semantics are required but the underlying computation is not naturally idempotent. In these cases, Concursus offers a distributed and time-based *idempotent filter* that can detect and drop duplicate messages to achieve exactly-once processing semantics.

7 Future Directions

We think there are natural affinities between Concursus and Kafka Streams, which we are currently exploring. Simply put, Kafka Streams can provide a low-overhead streaming abstraction on top of Concursus event sourcing and distributed CQRS. The integration mechanisms provided by Concursus for this purpose are the `JsonEventsOutChannel` and `JsonEventsInChannel`, which serialise events from event emitters so that they can be published, and deserialise subscribed events so that they can be dispatched to suitable handlers.

We see Concursus as an enabler for different architectural patterns, as opposed to one rigid system, and therefore we are exploring how a more integrated use of Kafka as a durable storage for Concursus streams would allow us to move to an architectural style where the event log is updated asynchronously and would therefore play a less central role in the system.

8 Further Reading

Source code `http://github.com/opencredo/concursus`

Web site `https://opencredo.com/concursus`

Notes

[1]Alain Badiou, Albert Toscano (tr.) (2009). Logics of Worlds: Being and Event II. Continuum.

[2]Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. FernÃ¡ndez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt & Sam Whittle (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. Proceedings of the VLDB Endowment, 8, 1792-1803. `http://research.google.com/pubs/pub43864.html`

[3]Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker & Ion Stoica (2013). Discretized streams: fault-tolerant streaming computation at scale. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13), 423-438. `http://dx.doi.org/10.1145/2517349.2522737`

[4]A useful introduction is given by Martin Fowler: `http://martinfowler.com/eaaDev/EventSourcing.html`