

Actors or Not

Async Event Architectures

demonware

Yaroslav Tkachenko

Senior Software Engineer at Demonware (Activision)

Background

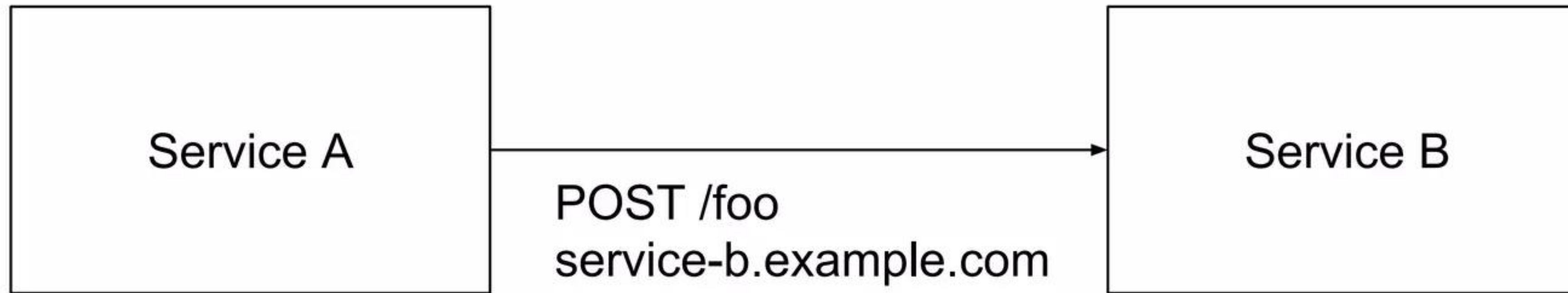
- 10 years in the industry
- ~1 year at Demonware/Activision, 5 years at Bench Accounting
- Mostly web, back-end, platform, infrastructure and data things
- @sap1ens / sap1ens.com
- Talk to me about data pipelines, stream processing and the Premier League ;-)

Two stories

demonware

Bench^π

Context: sync vs async communication



“Easy” way – HTTP (RPC) API

Context: sync vs async communication

- **Destination** – where to send request?
 - Service discovery
 - Tight coupling
- **Time** – expect reply right away?
- **Failure** – always expect success?
 - Retries
 - Back-pressure
 - Circuit breakers

You cannot make
synchronous requests
over the network
behave like local ones

Context: async communication styles

- **Point-to-Point Channel**
 - One sender
 - One receiver
- **Publish-Subscribe Channel (Broadcast)**
 - One publisher
 - Multiple subscribers

Context: Events vs Commands

- **Event**
 - Simply a notification that something happened in the past
- **Command**
 - Request to invoke some functionality (“RPC over messaging”)

demonŵare

ACTIVISION®

BILZARD®

ENTERTAINMENT

King



WWII

CALL OF DUTY
WWII

Demonware by the numbers

- 469+ million gamers
- 3.2+ million concurrent online gamers
- 100+ games
- 300,000 requests per second at peak
- Average query response time of <.02 second
- 630,000+ metrics a minute
- 132 billion+ API calls per month

Demonware Back-end Services

- Core game services including:
 - Auth
 - Matchmaking
 - Leaderboards
 - Marketplace
 - Loot & Rewards
 - Storage
 - Etc.
- **Erlang** for networking layer, **Python** for application layer
- Still have a big application monolith, but slowly migrating to independent services (**SOA**)

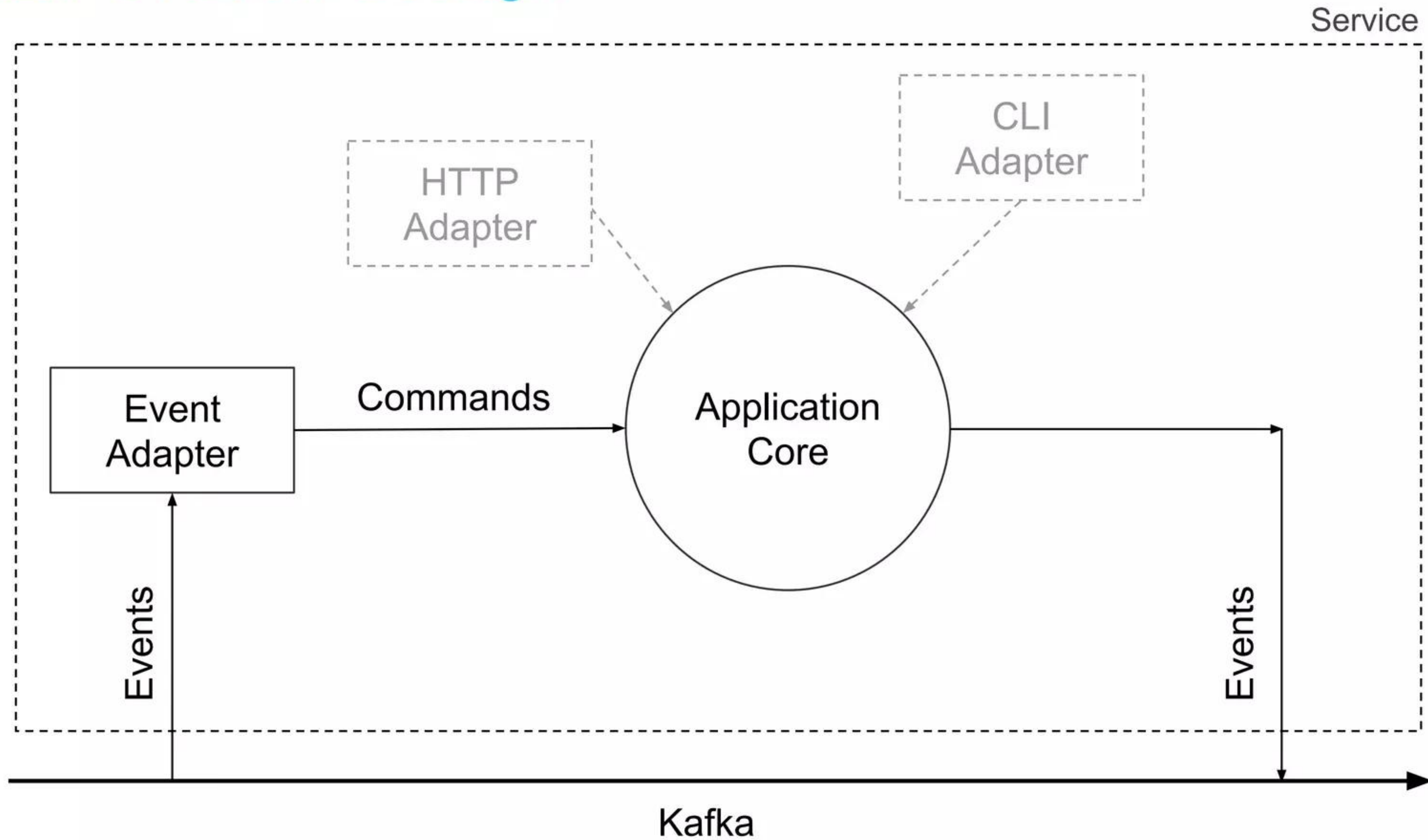
DW Services: Synchronous communication

- Lots of synchronous **request/response** communication between the monolith and the services using:
 - **HTTP**
 - **RPC**
- The requesting process:
 - conceptually **knows which service it wants to call into**
 - is **aware of the action that it is requesting**, and its effects
 - generally **needs to be notified of the request's completion** and any associated information before proceeding with its business logic

DW Services: Asynchronous communication*

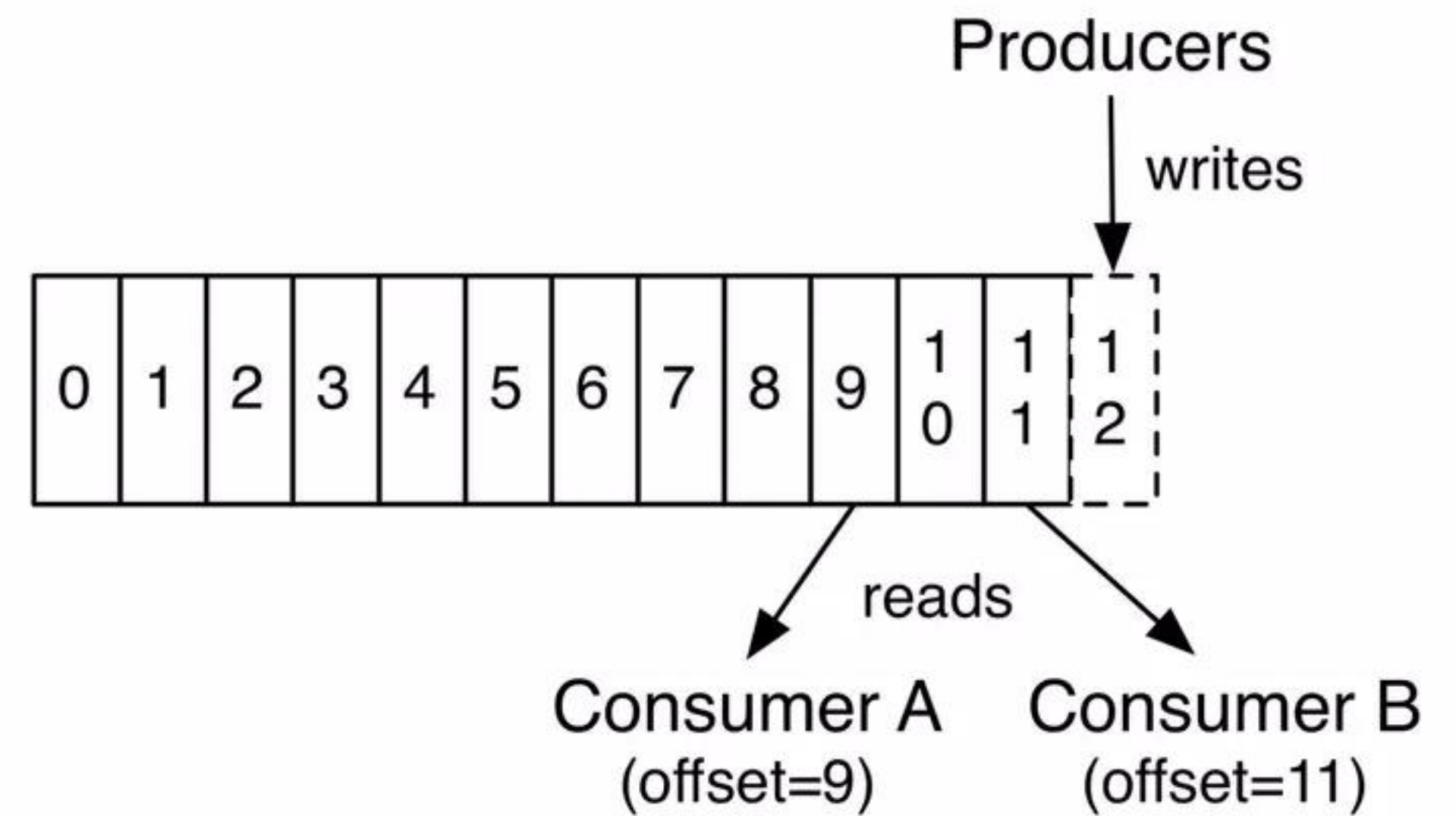
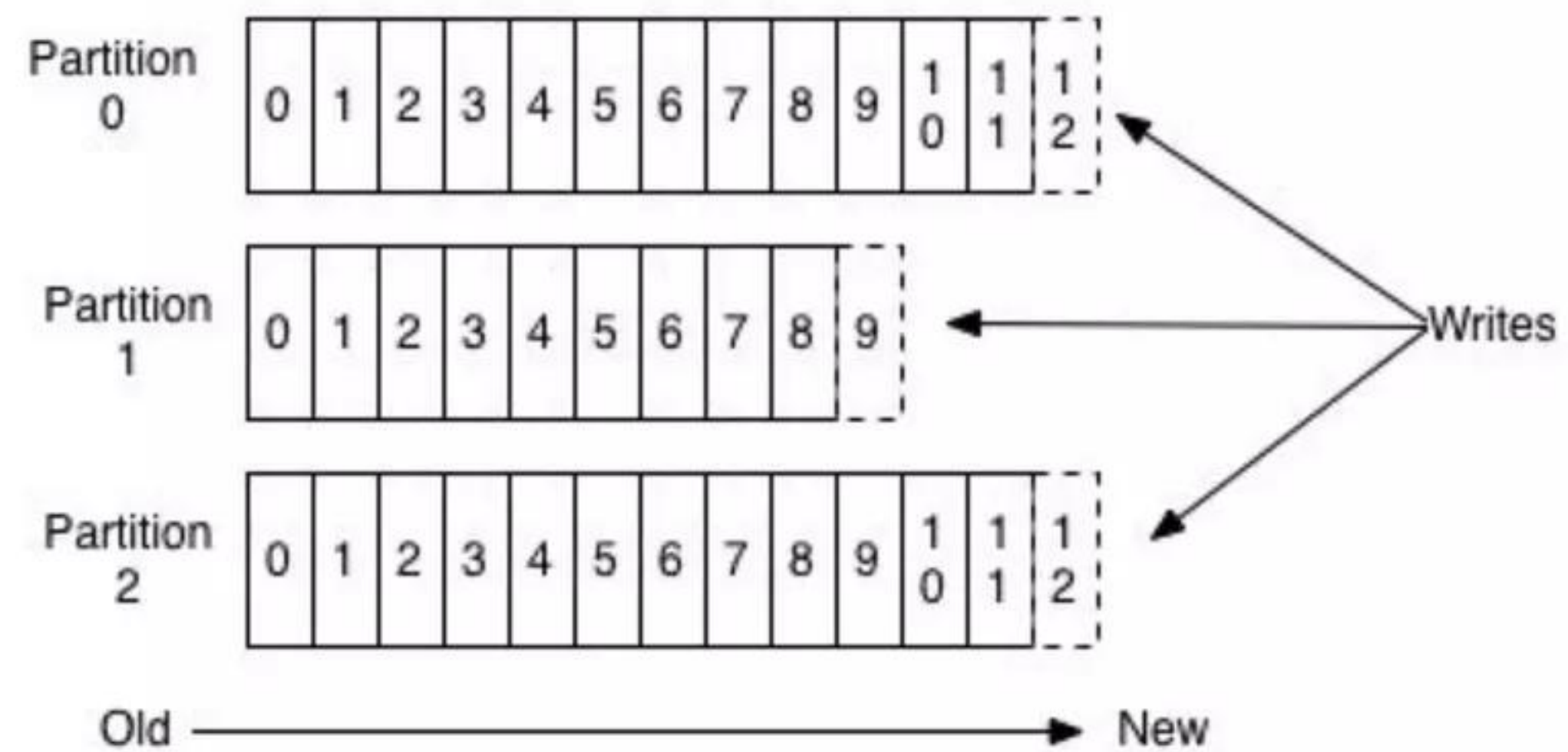
- Using **Domain Events**
- Communication model assumes the following:
 - The event may need to be handled by **zero or more service processes**, each with different use cases; the process that generates the event does not need to be aware of them
 - The process that generates the **event does not need to be aware of what actions will be triggered**, and what their effects might be
 - The process that generates the **event does not need to be notified of the handlers' completion** before proceeding with its business logic
- Seamless integration with the Data Pipeline / Warehouse

Domain Driven Design

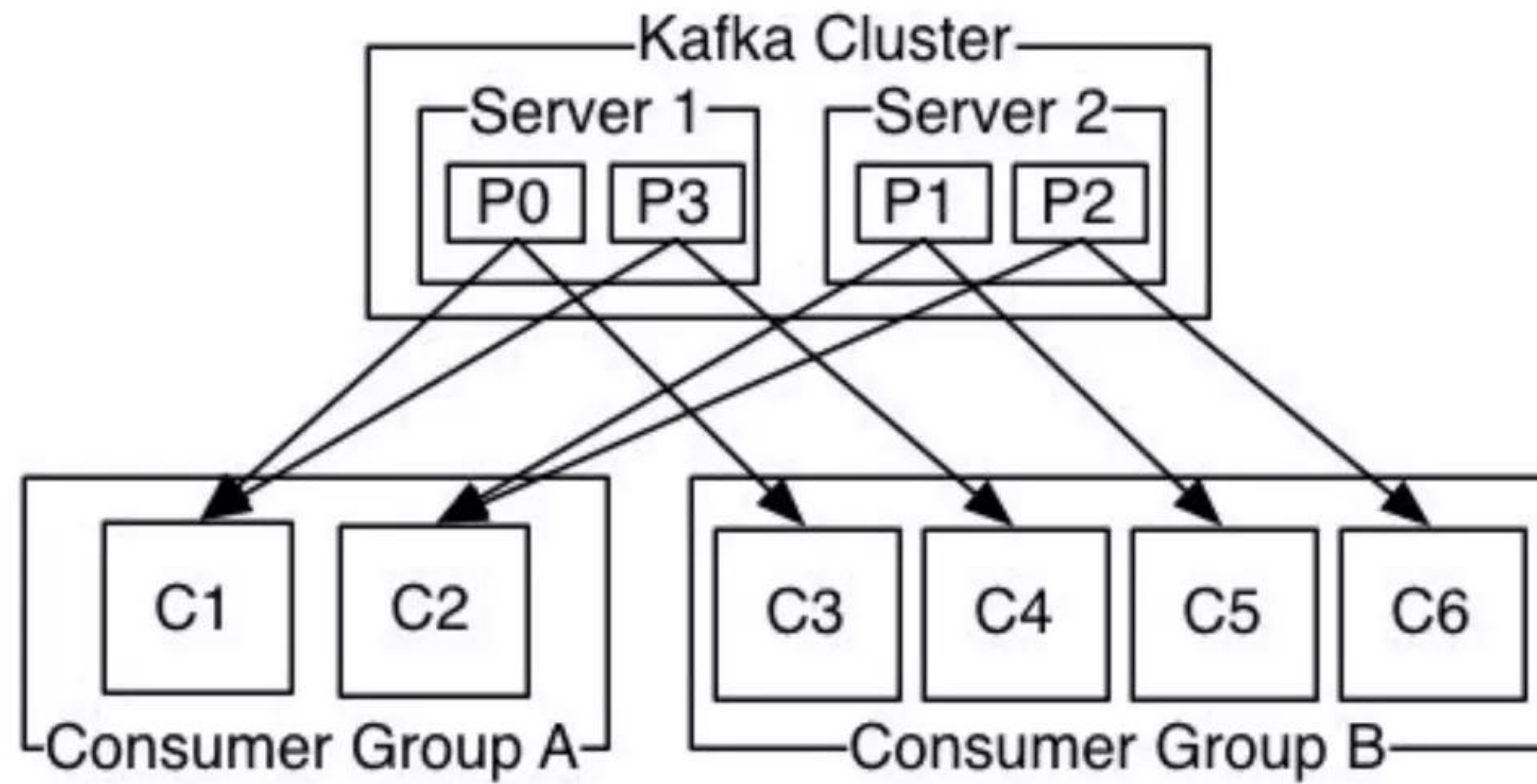


Kafka

Anatomy of a Topic



Kafka

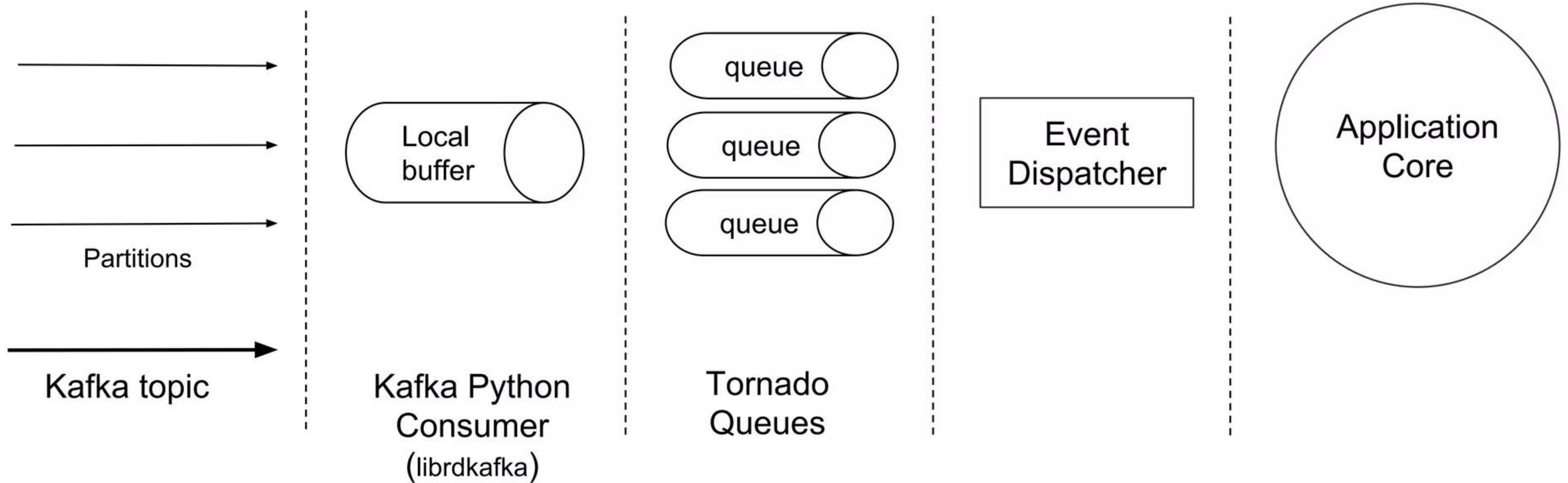


Publish-Subscribe OR **Point-to-Point** is a decision made by consumers

Kafka

- Service name is used as a topic name in Kafka
- Services have to explicitly subscribe to interested topics on startup (some extra filtering is also supported)
- All messages are typically partitioned by a user ID to preserve order

Event Dispatcher



Event Dispatcher

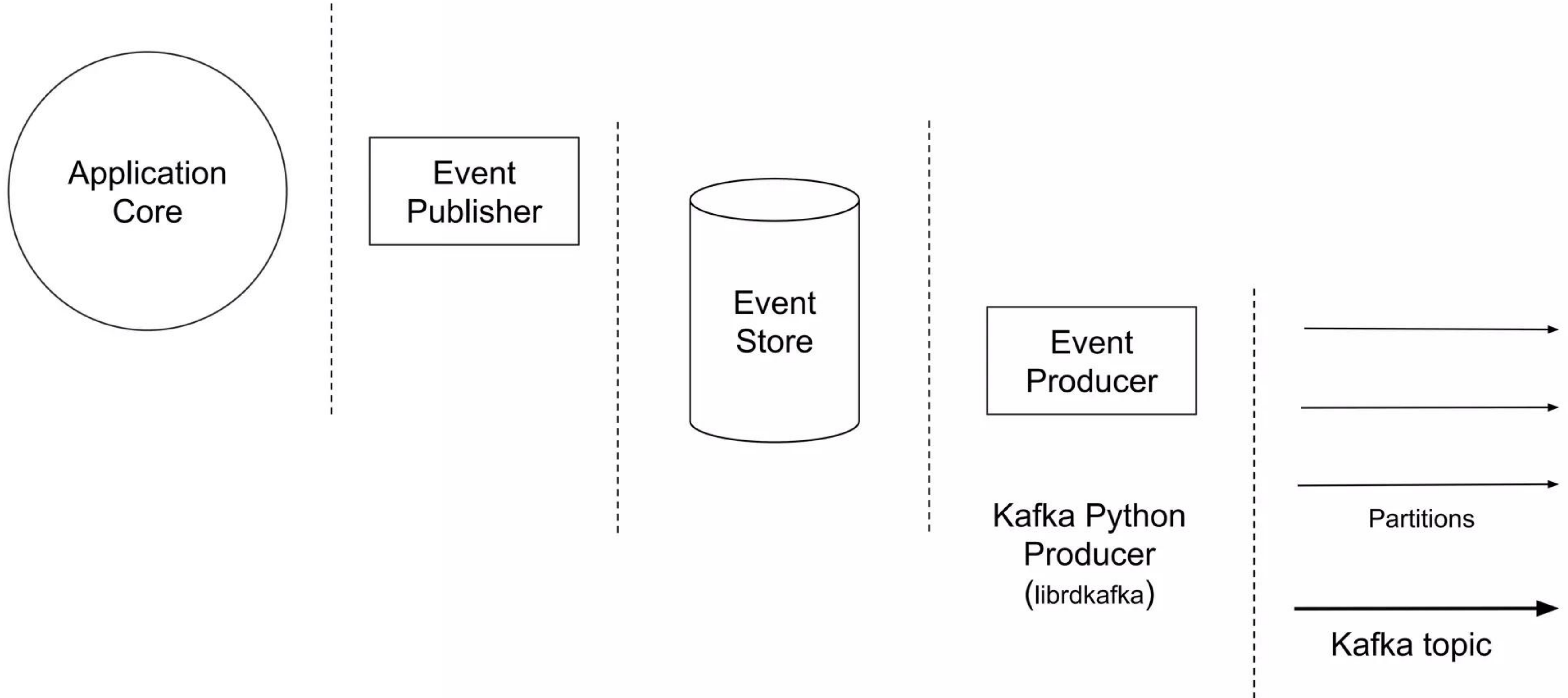
```
1  @demonata.event.source (
2      name='events_from_service_a'
3  )
4  class ServiceAEventsDispatcher (object):
5      def __init__(self, my_app_service):
6          self._app = my_app_service
7
8      @demonata.event.schema (
9          name='service.UserUpdated' ,
10         ge_version='1.2.3' ,
11         event_dto=UserUpdated
12     )
13     def on_user_updated (self, message, event):
14         assert isinstance (message, DwPublishedEvent)
15         # ...
```

Publishing Events

The following reliability modes are supported:

- **Fire and forget**, relying on Kafka producer (acks = 0, 1, all)
- **At least once (guaranteed)**, using remote EventStore backed by a DB
- **At least once (intermediate)**, using local EventStore

Event Publisher



Publishing Events

```
1 @demonata.coroutine
2 def handle_event_atomically(self, event_to_process):
3     entity_key = self.determine_entity_key(event_to_process)
4     entity = self.db.read(entity_key)
5
6     some_data = yield self.perform_some_async_io_read()
7     new_entity, new_event = self.apply_business_logic(
8         entity, event_to_process, some_data
9     )
10
11     # single-shard MySQL transaction:
12     with self.db.trans(shard_key=entity_key):
13         db.save(new_entity)
14         self.publisher.publish(new_event)
15         commit()
```

Event Framework in Demonware

- Decorator-driven consumers using callbacks
- Reliable producers
- Non-blocking IO using Tornado
- Apache Kafka as a transport

But still...

Can we do better?

Event Dispatcher

**This is just
a boilerplate**

```
1 @demonata.event.source (  
2     name='events_from_service_a'  
3 )  
4 class ServiceAEventsDispatcher (object):  
5     def __init__(self, my_app_service):  
6         self._app = my_app_service  
7  
8     @demonata.event.schema (  
9         name='service.UserUpdated' ,  
10        ge_version='1.2.3' ,  
11        event_dto=UserUpdated  
12    )  
13    def on_user_updated (self, message, event):  
14        assert isinstance (message, DwPublishedEvent)  
15        # ...
```

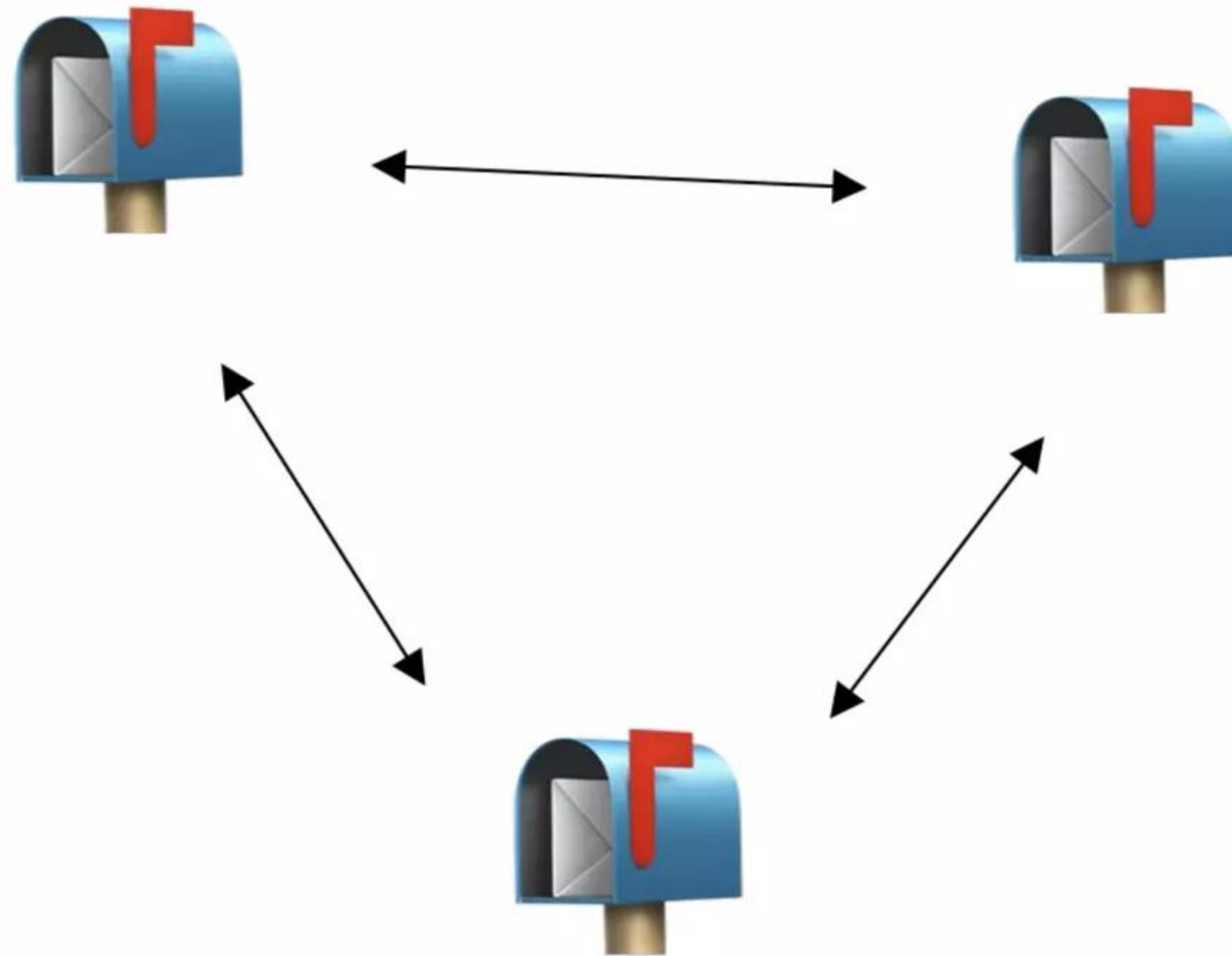
**Callback that
should pass
an event to
the actual
application**

Can we create
producers and
consumers that support
message-passing
natively?

Actors

- Communicate with **asynchronous messages** instead of method invocations
- Manage their **own state**
- When responding to a message, can:
 - Create other (child) actors
 - Send messages to other actors
 - Stop (child) actors or themselves

Actors



Actors: Erlang

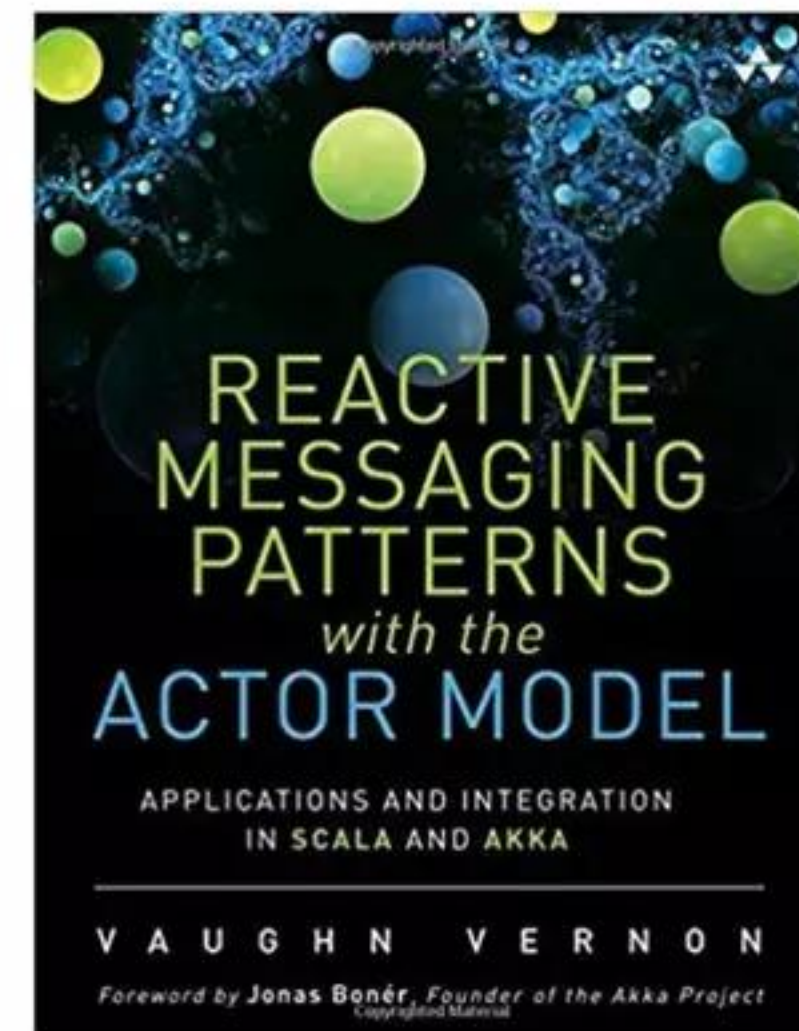
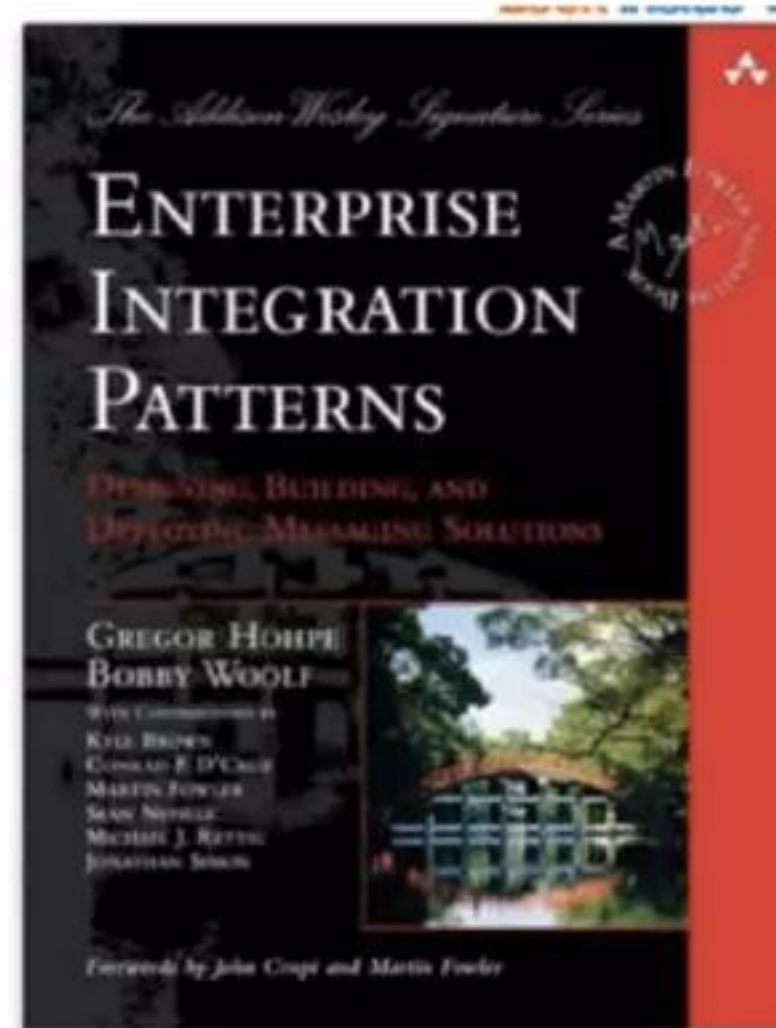
```
1 loop() ->
2     receive
3         {From, Msg} ->
4             io:format("received ~p~n", [Msg]),
5
6             From ! "got it";
7     end.
```

Actors: Akka

```
1 class MyActor extends Actor with ActorLogging {  
2   def receive = {  
3     case msg => {  
4       log.info(s"received $msg")  
5  
6       sender() ! "got it"  
7     }  
8   }  
9 }
```

Actor-to-Actor communication

- **Asynchronous and non-blocking message-passing**
- Doesn't mean senders must wait indefinitely – timeouts can be used
- Location transparency
- Enterprise Integration Patterns!



Bench^π



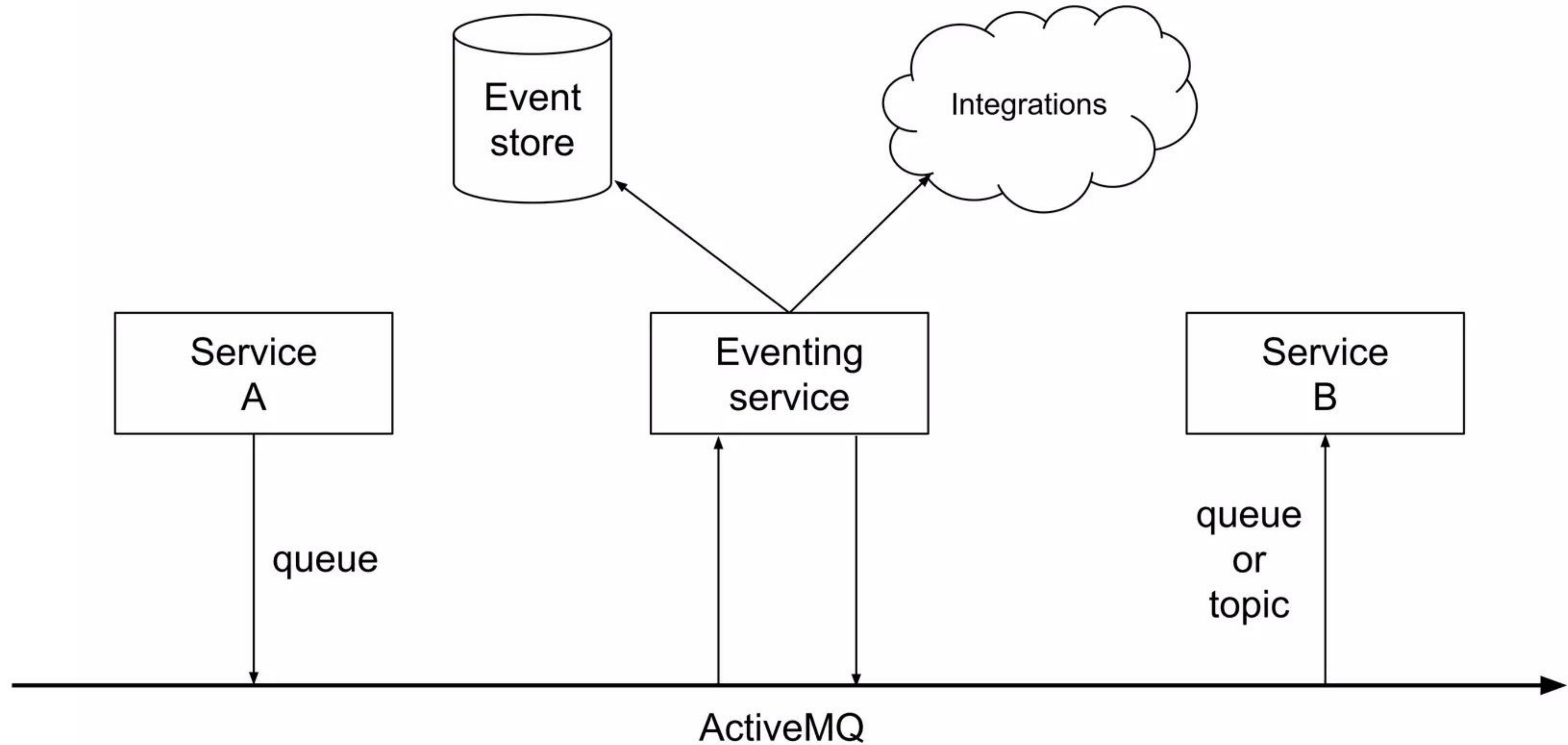
Bench Accounting Online Services

- Classic SAAS application used by the customers and internal bookkeepers:
 - Double-entry bookkeeping with sophisticated reconciliation engine and reporting [no external software]
 - Receipt collection and OCR
 - Integrations with banks, statement providers, Stripe, Shopify, etc.
- Enterprise **Java** monolith transitioning to **Scala** microservices (with **Akka**)
- Legacy event-based system built for notifications

Bench Accounting Legacy Eventing

- Multiple issues:
 - Designed for a few specific use-cases, **schema is not extendable**
 - Wasn't built for **microservices**
 - Tight **coupling**
- New requirements:
 - Introduce **real-time** messaging (web & mobile)
 - Add a framework for producing and consuming **Domain Events** and **Commands** (both point-to-point and broadcasts)
 - Otherwise very similar to the Demonware's async communication model

Bench Accounting Eventing System



ActiveMQ

Message in



FIFO queue



Message out

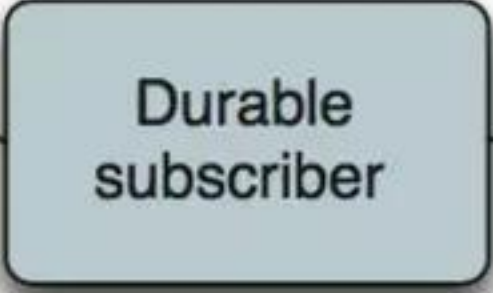
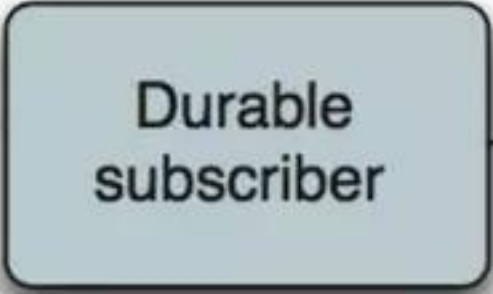
Point-to-Point

Message in



Topic

Messages out



Publish-Subscribe

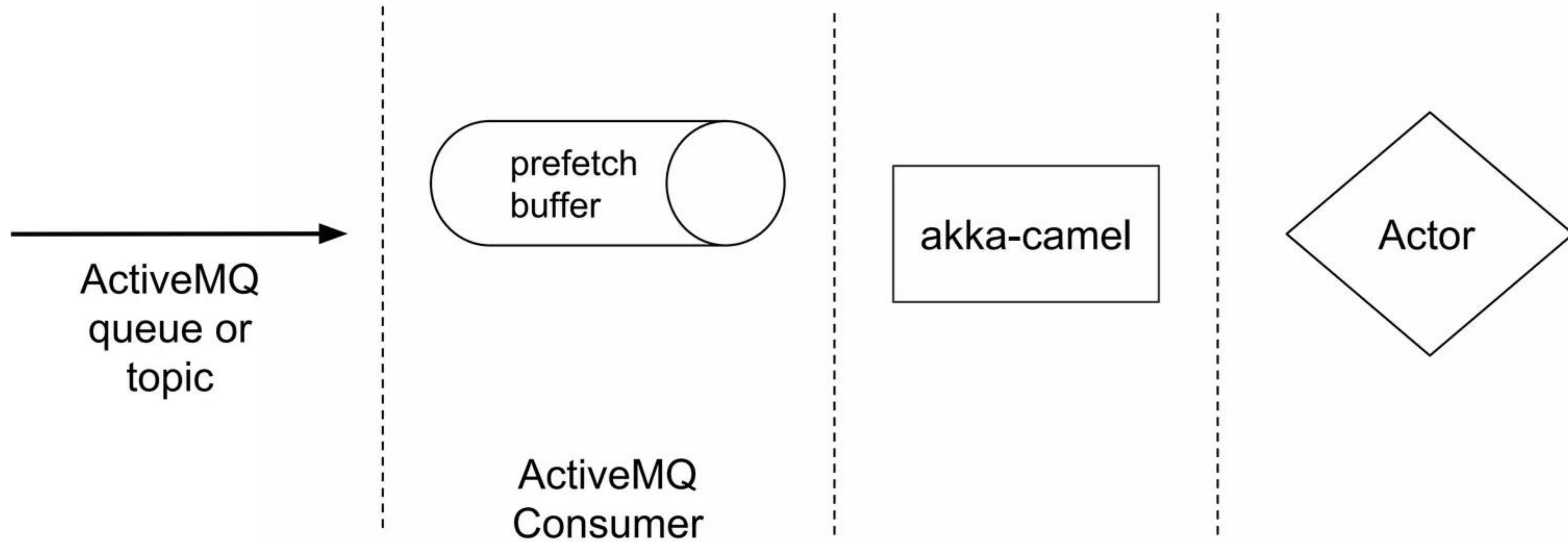
ActiveMQ

- Service name is used as a queue or topic name in ActiveMQ, but there is also a topic for global events
- Services can subscribe to interested queues or topics any time a new actor is created
- Supports 3 modes of operations:
 - **Point-to-Point** channel using a queue (perfect for **Commands**)
 - **Publish-Subscribe** channel with guaranteed delivery using a Virtual topic
 - Global **Publish-Subscribe** channel with guaranteed delivery using a Virtual topic

Secret sauce: Apache Camel

- Integration framework that implements Enterprise Integration Patterns
- akka-camel is an official Akka library (now deprecated, Alpakka is a modern alternative)
- Can be used with any JVM language
- “The most unknown coolest library out there”: JM (c)

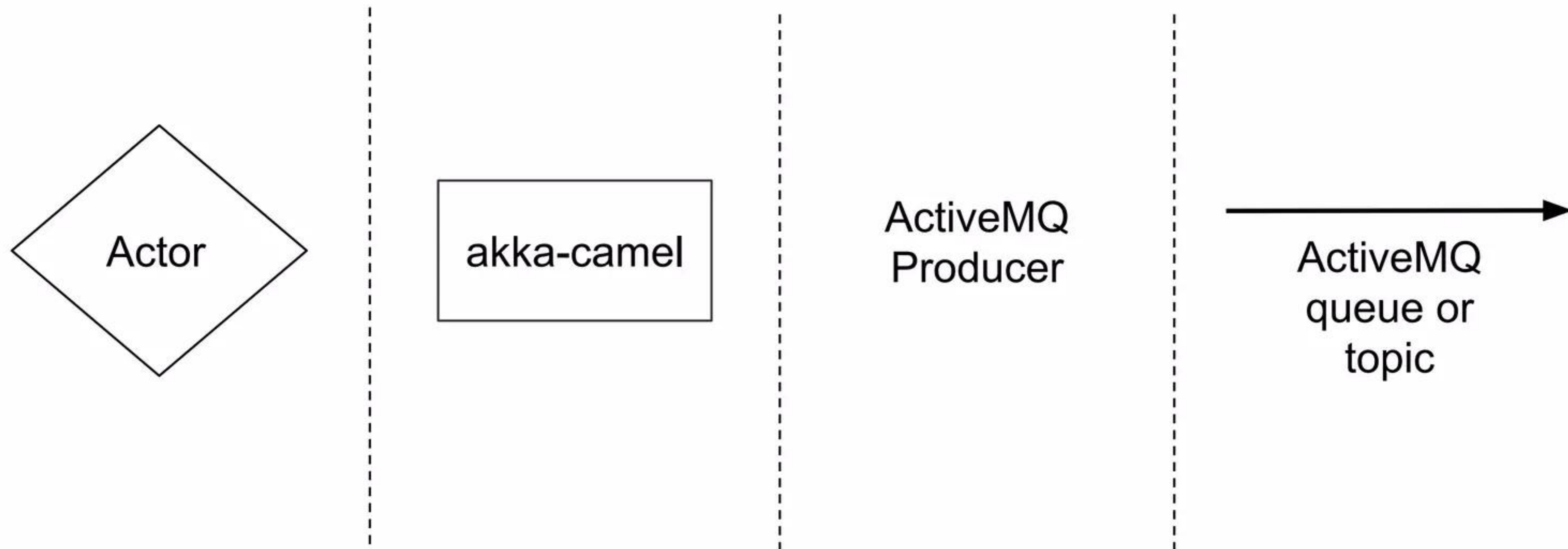
Event Listener



Event Listener

```
1 class CustomerService extends EventingConsumer {
2   def endpointUri = "activemq:Consumer.CustomerService.VirtualTopic.events"
3
4   def receive = {
5     case e: CamelMessage if e.isEvent && e.name == "some.event.name" => {
6       self ! DeleteAccount (e.clientId, sender())
7     }
8
9     case DeleteAccount (clientId, originalSender) => {
10      // ...
11    }
12  }
13 }
```

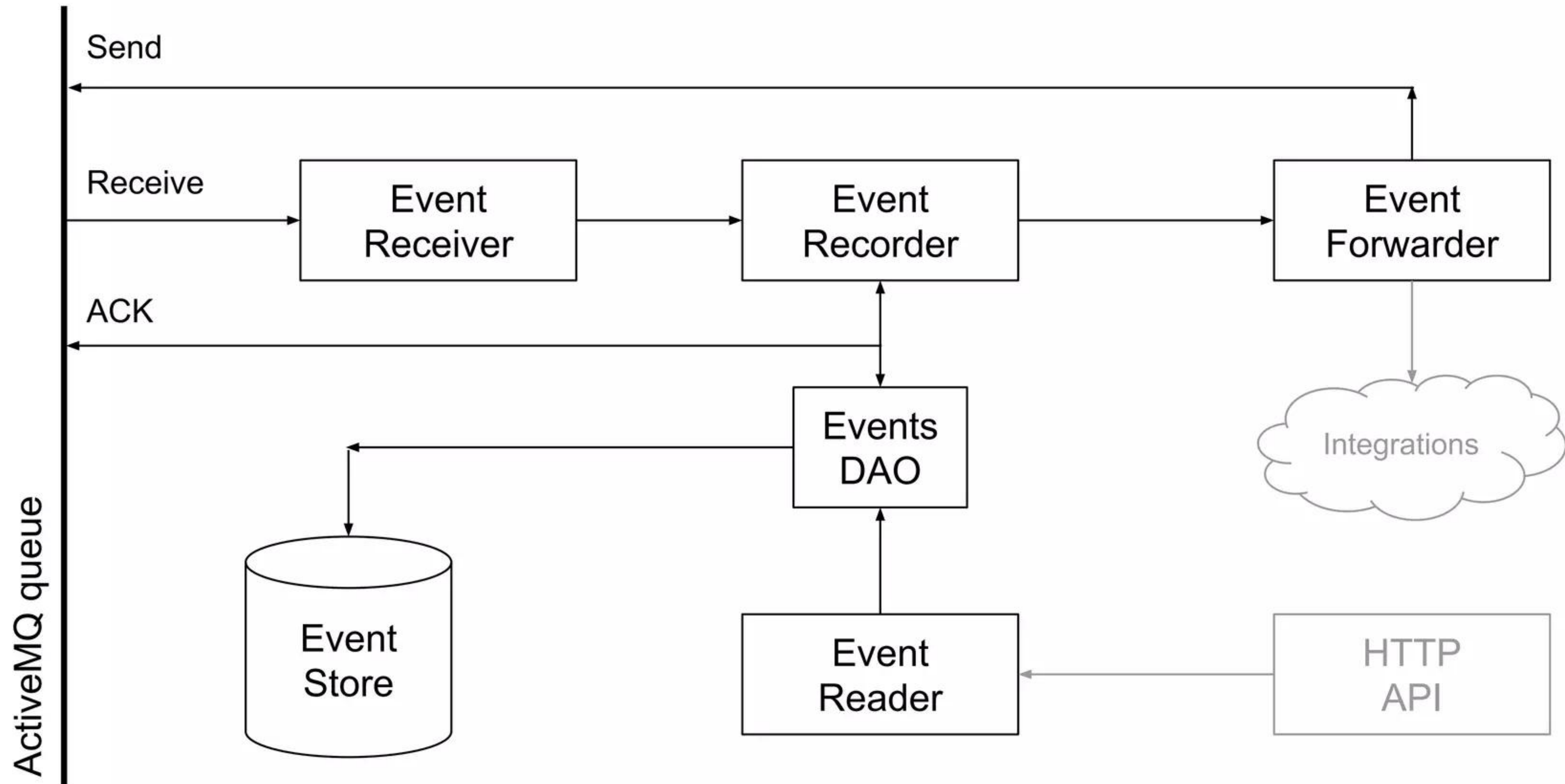
Event Sender



Event Sender

```
1 // Broadcast
2 EventingClient
3     .buildSystemEvent (Event.BankError, userId, Component.ServiceA)
4     .send(true)
5
6 // Direct
7 EventingClient
8     .buildSystemEventWithAsset (Event.BankError, userId, Component.ServiceB)
9     .buildUrlAsset ("http://example.com" )
10    .sendDirect ("reporting" )
```

Eventing Service



Eventing Service

So, do we need this “router” service?

- Routing is handled in one place
- Lightweight consumers and producers
- The same Event Store is used for all services

Event framework in Bench Accounting

- Actor-based consumers and producers using Apache Camel
- Producer with ACKs
- Non-blocking IO
- Apache ActiveMQ as a transport

Lessons learned

So, Actors

- Semantics is important! Natural message-passing in Actors is a huge advantage
- Asynchronous communication and location transparency by default makes it easy to move actors between service boundaries
- We could also talk about supervision hierarchies and “Let it crash” philosophy, excellent concurrency, networking features, etc... next time! You can start with basics

Recommendations

- Domain Driven Design and Enterprise Integration Patterns are great!
- Understand your Domain space and choose the concepts you need to support: Events, Commands, Documents or all of them
- Explicitly handle all possible failures. They will happen eventually
- Event Stores can be used for so many things! Tracing and debugging, auditing, data analytics, etc.
- Actors or not? It really depends. It's possible to build asynchronous, non-blocking event frameworks in Java, Python, Node.js or a lot of the other languages, but actors are asynchronous and message-based by default

Recommendations

- Carefully choose the transport layer. Apache Kafka can handle an impressive scale, but many messaging features are missing / support just introduced
- Understand what you need to optimize: latency or throughput. You might need to introduce multiple channels with different characteristics
- Do you really need exactly-once semantics?
- Message formats and schemas are extremely important! Choose binary formats (Protobuf, Avro) AND/OR make sure to use a schema registry and design a schema evolution strategy
- Consider splitting your messages into an envelope (metadata) and a payload. Events and Commands could use the same envelope

Challenges

- We're too attached to the synchronous request/response paradigm. It's everywhere – in the libraries, frameworks, standards. It takes time to learn how to live in the asynchronous world
- High coupling will kill you. Routing is not a problem when you have a handful of services (producers/consumers), but things get really complicated with 10+ services. Try to avoid coupling by using Events as much as possible and stay away from Commands unless you really need them
- Managing a properly partitioned, replicated and monitored message broker cluster is still a non-trivial problem. Consider using managed services if your Ops resources are limited

Challenges

- It's very straightforward to implement event-based communication for writes, but harder for reads. You'll probably end up with some sort of DB denormalization, in-memory hash join tables, caching or all of the above
- When you have dozens of producers and consumer scattered across the service it becomes challenging to see the full picture. State and sequence diagrams can help with capturing business use-cases, distributed tracing becomes almost a must-have
- When things break you won't notice them immediately without a proper monitoring and alerting. Considering covering all critical business use-cases first

That signup page...

Thanks

Davide Romani (Demonware)
Pavel Rodionov (Bench Accounting)

Questions?

@sap1ens | sap1ens.com