Most of the content of these slides is extracted from the following references:

Outline

1. Motivations and requirements
2. Definition of Event-Based Systems
3. Formal specification of simple event-based system
4. Notification filtering mechanisms
5. Lab on topic-based filtering with the AMQP standard
1 Motivations and requirements

1.1 E.g. Spotify
1.2 E.g. Autonomic computing—Loop MAPE-K
1.3 E.g. Data-driven inference cycle
1.4 Requirements
1.1 E.g. Spotify

- Routing, event-driven for high performance, scalability (number of events per minute, GB per second)

1.2 E.g. Autonomic computing—Loop MAPE-K

1.3 E.g. Data-driven inference cycle

1.4 Requirements

- Data production/consumption decoupling
  - Synchronization decoupling: asynchronous and anonymous communication
  - Space decoupling: unknown producers and consumers
  - Time decoupling: production and consumption at different times

- Scalability: in messages per minute, in data per second, in clients (producers and consumers)

- Data lifecycle management + filtering + aggregation

- Adaptation to mobile and heterogeneous environments

- One name for many “technologies”: distributed event-based systems, distributed publish-subscribe systems, distributed messaging services, message-oriented middleware, active databases, etc.

1.4.1 Example of unified architecture model: Context data distribution

- E.g. for the events from the Internet of Things
1.4.2 Classification of Data Dissemination

- Sensor direct access
  - Data flooding
  - Subscription flooding
  - System wide scope
  - Limited scope
  - Context-obliviou
  - Context-aware

- Flooding-based

- Selection-based

- Gossip-based

---

## Comparison I

<table>
<thead>
<tr>
<th>Category</th>
<th>Sub-category</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor Direct</td>
<td>-</td>
<td>• No state on mobile nodes</td>
<td>• Strong coupling between sources/sinks</td>
</tr>
<tr>
<td>Access</td>
<td></td>
<td>• Low network overhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Sink always receive interesting data</td>
<td></td>
</tr>
<tr>
<td>Flooding-based</td>
<td>Data flooding</td>
<td>• Low state on mobile nodes</td>
<td>• High network overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Loose coupling between sources/sinks</td>
<td>• Dissemination can reach not-interested nodes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Sink always receive interesting data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscription</td>
<td>• Loose coupling between sources/sinks</td>
<td>• High state on mobile nodes</td>
</tr>
<tr>
<td>flooding</td>
<td></td>
<td>• Sink always receive interesting data</td>
<td>• High network overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dissemination reaches only interested nodes</td>
<td></td>
</tr>
</tbody>
</table>

## Comparison II

<table>
<thead>
<tr>
<th>Method</th>
<th>System wide scope</th>
<th>Limited scope</th>
<th>Gossip-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection-based</td>
<td>● Loose coupling between sources/sinks&lt;br&gt;● Sink always receive interesting data&lt;br&gt;● Dissemination reaches only interested nodes</td>
<td>● Medium state on mobile nodes&lt;br&gt;● Medium network overhead</td>
<td>● Sink could miss interesting data&lt;br&gt;● Medium state on mobile nodes&lt;br&gt;● Low network overhead</td>
</tr>
<tr>
<td>Limited scope</td>
<td>● Loose coupling between sources/sinks&lt;br&gt;● Dissemination reaches only interested nodes</td>
<td>● Medium state on mobile nodes&lt;br&gt;● Medium network overhead</td>
<td>● Sink could miss interesting data&lt;br&gt;● Dissemination can reach not-interested nodes</td>
</tr>
<tr>
<td>Gossip-based</td>
<td>● Low state on mobile nodes&lt;br&gt;● Loose coupling between sources/sinks&lt;br&gt;● Low network overhead</td>
<td>● Medium state on mobile nodes&lt;br&gt;● Sink could miss interesting data&lt;br&gt;● Dissemination can reach not-interested nodes</td>
<td></td>
</tr>
</tbody>
</table>
1.4.4 Classification of Data Routing

### Comparison I

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Architecture</td>
<td>-</td>
<td>• Context data access is always ensured</td>
<td>• Limited scalability and reliability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No management overhead for routing overlay maintenance</td>
<td>• Locality principles difficult to apply</td>
</tr>
<tr>
<td>Decentralized Architecture</td>
<td>Flat distributed architecture</td>
<td>• Increased scalability and reliability</td>
<td>• Context data access could not be always ensured</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Locality principles easy to apply</td>
<td>• Additional management overhead for routing overlay maintenance</td>
</tr>
<tr>
<td></td>
<td>Hierarchical distributed architecture</td>
<td>• Increased scalability and reliability</td>
<td>• Context data access could not be always ensured</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Locality principles easy to apply</td>
<td>• Additional management overhead for routing overlay maintenance</td>
</tr>
</tbody>
</table>

---

2 Definition of Event-Based Systems

2.1 Models of interaction and EBS
2.2 Rough intuition of EBS service
2.3 Constituents of EBS
### 2.1 Models of interaction and EBS

<table>
<thead>
<tr>
<th>Adressee</th>
<th>Initiator: Consumer</th>
<th>Initiator: Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Request/Reply</td>
<td>Callback</td>
</tr>
<tr>
<td>Indirect</td>
<td>Anonymous Request/Reply</td>
<td>Event-Based</td>
</tr>
</tbody>
</table>

- **Initiator**: describes whether the consumer or the provider initiates the interaction.

- **Addressing**: indicates whether the addressee of the interaction is known or unknown.

- The tradeoff is between the simplicity of request/reply and the flexibility of event-based interaction.
2.2 Rough intuition of EBS service

- Some processes advertise future publication of information and publish information
- Some processes subscribe to information filters and consume information
- The notification service decouples the components:
  - Producers are unaware of any consumers
  - Consumers rely only on the information published, not on where or by whom it is published
- Event-based components are not designed to work with specific other components
  - Facilitates the separation of communication from computation
  - Potential for easy integration of autonomous and heterogenous components into systems that are easy to evolve and scale
2.3 Constituents of EBS

Event Consumer

F : Filter

Publish/Subscribe interface

N : Notification

1.a. advertise (F)

Producer

2. publish (N)

1.b. subscribe (F)

3. notify (N)

Notification Service Notification Service

Communication Implementation
2.3.1 Terminology I

- **Event**: any happening of interest that can be observed from within a computer
  
  - Event example: physical event, timer event, etc.

- **Notification**: an object that contains data describing the event

- **Producer**: a component that publishes notifications

- **Consumer**: a component that reacts to notifications delivered to them by the notification service

- **Subscription**: describes a set of notifications a consumer is interested in

- **Advertisement**: is issued by a producer to declare the notifications it is willing to send
2.3.1 Terminology II

**Event notification service:**

- Is the mediator in event-based systems that decouples producers from consumers
- Implements a publish/subscribe interface

**Operations:**

- `publish(n)`: a producer pushes notification \( n \) to the notification service
- `advertise(F)`: a producer advertises that it will send notifications which match the filter \( F \)
- `subscribe(F)`: a consumer subscribes to receive notifications which match the filter \( F \)
- `notify(n)`: the notification service delivers the notification \( n \) to those consumers that have a matching subscription

- Maintains published events and subscribers interests
3 Formal specification of simple event-based system

3.1 Formal background—Temporal logic
3.2 Changes of the state caused by interface operations (w/o advertisements)
3.3 Trace-based specification of simple event-based system (w/o advertisements)
3.4 Changes of the state caused by the adding advertisements
3.5 Safety specification of simple event system with advertisements
3.6 Liveness specification of simple event system with advertisements
3.1 Formal background—Temporal logic

- **Trace**: a sequence of states: $\sigma = s_0, s_1, s_2, \ldots$
  - **Subtrace**: $\sigma|_i$ is the trace $s_i, s_{i+1}, s_{i+2}, \ldots$

- **Atomic predicate** $P$ is true for every trace whose first state satisfies $P$

- **Formula $\Psi$**: with **quantifiers** ($\forall, \exists$) and **logical operators** ($\lor, \land, \Rightarrow, \neg$)

- **Temporal operators**:
  - $\blacksquare$ ("always")
    - $\blacksquare \Psi$ is true for traces $\sigma$ iff $\forall i \geq 0, \Psi$ is true for $\sigma|_i$
    - $\blacksquare P$ means $P$ always holds, *i.e.* for all subtraces
  - $\Diamond$ ("eventually")
    - $\Diamond \Psi$ is true for traces $\sigma$ iff $\exists i \geq 0 : \Psi$ is true for $\sigma|_i$
    - $\Diamond P$ means $P$ will hold eventually, *i.e.* there exists a subtrace for which $P$ holds
  - $\bigcirc$ ("next")
    - $\bigcirc \Psi$ is true for traces $\sigma$ iff $\Psi$ is true for $\sigma|_1$
    - $\bigcirc P$ means $P$ holds for the subtrace starting at the second place of the trace
3.1.1 Exercise

? ◻◻P

? ◻◻P

? ◻[P ⇒ ◻P]

? ◻[P ⇒ ◇Q]

? ◻[P ⇒ ○◻¬P]

? P ⇒ ◇◻Q

? ◻¬P ∨ ◻¬Q ≡ ¬(◇P ∧ ◇Q)
3.2 Changes of the state caused by interface operations (w/o advertisements)

- $X$: a component of a system (being a producer and/or a consumer)
- $C$: the set of all the components
- $S_X$: a set of active subscriptions for component $X$
- $P_X$: a set of published notifications by component $X$
- $D_X$: a set of delivered notifications to component $X$
- $N$: the set of all the notifications; $N \subseteq N'$: a set of notifications
  - $n \in N(S_X)$: $X$ has a subscription that matches $n \in N$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>State Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sub(X, F)$</td>
<td>Component $X$ subscribes to filter $F$</td>
<td>$S'_X = S_X \cup {F}$</td>
</tr>
<tr>
<td>$unsub(X, F)$</td>
<td>Component $X$ unsubscribes to filter $F$</td>
<td>$S'_X = S_X \setminus {F}$</td>
</tr>
<tr>
<td>$pub(X, n)$</td>
<td>Component $X$ publishes $n$</td>
<td>$P'_X = P_X \cup {n}$</td>
</tr>
<tr>
<td>$notify(X, n)$</td>
<td>Component $X$ is notified about $n$</td>
<td>$D'_X = D_X \cup {n}$</td>
</tr>
</tbody>
</table>

- « ′ » indicates the state of a variable after the execution of the interface operation
3.2.1 Exercise

? ◊ notify(X, n)

? □ ¬ unsub(X, F)

? □ [notify(X, n) ⇒ □ ◊ ¬ notify(X, n)]

? □ [notify(X, n) ⇒ n ∈ N(SX)]

? □ [notify(Y, n) ⇒ n ∈ ∪X∈C Px]
3.3 Trace-based specification of simple event-based system (w/o advertisements)

- A component receives
  (a) only notifications it is currently subscribed to
  (b) only notifications that have previously been published
  (c) a notification at most once
  (d) all future notifications matching one of its active subscriptions

- **Safety**: demands that “something irremediably bad” will never happen

\[
\square \left[ \text{notify} \left( Y, n \right) \implies \left[ n \in N \left( S_Y \right) \right] \right] \quad (=a)
\]

\[
\land \left[ n \in \bigcup_{X \in C} P_X \right] \quad (=b)
\]

\[
\land \left[ \Diamond \Box \neg \text{notify} \left( Y, n \right) \right] \quad (=c)
\]

- **Liveness**: requires that “something good” will eventually happen

\[
\square \left[ \square \left( F \in S_Y \right) \implies \Diamond \Box \left[ \text{pub} \left( X, n \right) \land n \in N \left( F \right) \implies \Diamond \text{notify} \left( Y, n \right) \right] \right] \quad (=d)
\]
3.4 Changes of the state caused by the adding advertisements

- $A_X$: set of all active advertisements of component $X$
- $n \in N(A_X)$: $X$ has an advertisement that matches $n \in \mathcal{N}$

<table>
<thead>
<tr>
<th>$adv(X, F)$</th>
<th>Component $X$ advertises filter $F$</th>
<th>$A'_X = A_X \cup {F}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$unadv(X, F)$</td>
<td>Component $X$ unadvertises filter $F$</td>
<td>$A'_X = A_X \setminus {F}$</td>
</tr>
</tbody>
</table>
3.5 Safety specification of simple event system with advertisements

(a) + (b) + (c) +

(e) If a notification is published that does not match any of the active advertisements of the publishing component, the notification should not be delivered to any component

$$\square \left[ \text{notify}(Y, n) \implies \bigcirc \square \neg \text{notify}(Y, n) \right] \quad (=c)$$

$$\land \left[ \text{notify}(Y, n) \implies n \in \bigcup_{X \in C} P_X \cap N(S_Y) \right] \quad (=b,a)$$

$$\land \left[ \text{pub}(X, n) \land n \notin N(A_X) \implies \square \neg \text{notify}(Y, n) \right] \quad (=e)$$
3.6 Liveness specification of simple event system with advertisements

f1) If a client $Y$ is always subscribed to $F$ and a client $X$ always advertises $G$

f2) then there exists a future time where a notification $n$ published by $X$ matches $F$ and $G$

f3) will lead to the delivery of $n$ to $Y$.

$$\square \left[ \square (F \in S_Y) \land \square (G \in A_X) \right] \quad (=f1)$$
$$\Rightarrow \left[ \lozenge \square (pub(X, n) \land n \in N(F) \cap N(G) \right) \quad (=f2)$$
$$\Rightarrow \lozenge notify(Y, n)) \right] \quad (=f3)$$
4 Notification filtering mechanisms

4.1 Channels
4.2 Subject-based filtering (also called topic-based filtering)
4.3 Type-based filtering
4.4 Content-based filtering
4.1 Channels

- Producers select a channel into which a notification is published
- Consumers select a channel and will get all notifications published therein
- Channel identifier is only the visible message part to the event-based service
- **Example of solution:** CORBA Event Service, CORBA Notification Service
4.2 Subject-based filtering (also called topic-based filtering)

- Uses string matching for notification selection
- Each notification and subscription is defined as a rooted path in a tree of subjects
- Example:
  - A stock exchange application publishes new quotations of FooBar under the subject: /Exchange/Europe/London/Technology/FooBar
  - Consumers subscribe for /Exchange/Europe/London/Technology/* to get all technologies quotations
- Example of solution: AMQP standard (Advanced Message Queuing Protocol), TIBCO Rendezvous, JMS (Java Message Queue), WebSphere MQ Publish/Subscribe (WMQPS), Apache Kafka, Apache Qpid, Spring RabbitMQ
4.3 Type-based filtering

- Uses subtype inclusion to select notifications
- If a consumer subscribes to the type `StockQuote`, it will receive Technology quotations and other notifications that are sub-types of `StockQuote`
### 4.4 Content-based filtering

- Filters are evaluated on the whole content of notifications.
- Example solutions: template matching, extensible filter expressions on name value pairs, XPath expressions on XML schemas, etc.
- Example:
  - \(m1\): \(((\text{company}, \text{“Telco”}), (\text{price}, 120))\)
  - \(m2\): \(((\text{company}, \text{“Telco”}), (\text{price}, 90))\)
  - Filter \(F\): \(((\text{company} = \text{“Telco”}), (\text{price} < 100))\)
5 Lab on topic-based filtering with the AMQP standard

Lots of implementations: RabbitMQ, Apache Kafka, Apache Qpid

- We propose to follow a tutorial on RabbitMQ
5.1 AMQP and RabbitMQ concepts

Excerpts from RabbitMQ Web pages

- The AMQP model consists of a set of components that route and store notifications within a broker. There are three main types of components, which are connected into processing chains in the broker to create the desired functionality:

  - The **exchange** receives notifications from publishers and routes them to “message queues” based on criteria.
  - The **message queue** stores notifications until they can be safely processed by a consumer (or multiple consumers).
  - The **binding** defines the relationship between a message queue and an exchange, and provides the notification routing criteria.

Using this model, it is possible to emulate the classic message-oriented middleware concepts of store-and-forward queues and topic-based DEBS. This is of course the latter service that we want.

- **Exchange.**
  An exchange accepts notifications from a producer and routes them to message queues according to pre-arranged criteria. These criteria are called “bindings”. Exchanges are matching and routing engines: They inspect notifications and using their binding tables, decide how to forward these notifications to message queues or other exchanges. Exchanges never store notifications.

AMQP defines a number of standard exchange types. Exchange instances are named so that applications can specify how to bind queues and publish notifications.
**Message Queue.**
A message queue stores notifications in memory or on disk, and delivers them in FIFO sequence to one or more consumers. Message queues are storage and distribution entities. They are created and maintained by the broker. A message queue possesses various properties: private or shared, durable or temporary or auto-deleted, client-named or broker-named, etc. Durable message queues last until they are deleted. Temporary message queues last until the broker shuts down. Auto-deleted message queues last until they are no longer used.

In the presence of multiple readers of the same queue, the queue may not exhibit FIFO characteristics. The only way to guarantee FIFO is to have just one consumer connected to the queue.

An acknowledgement is a message from the consumer to a message queue indicating that the consumer has successfully processed a notification. There are two possible acknowledgement models: (1) automatic, in which the broker removes a content from a message queue as soon as it delivers it to a consumer; (2) explicit, in which the consumer must send an Ack message for each notification, or batch of notifications, that it has processed. Most of the time, consumers rely on the automatic acknowledgement mechanism.

**Binding.**
A binding is a relationship between a message queue and an exchange. The lifespan of bindings depends on the message queues they are defined for — i.e., when a message queue is destroyed, its bindings are also destroyed.
Routing key and binding key.
An exchange examines a notification's properties —i.e., its header fields— and decides how to route the notification. In the majority of simple cases, the exchange examines a single key field, which is called the “routing key”. A routing key is a virtual address that the exchange may use to decide how to route notifications. For topic-based routing, the routing key is usually a topics hierarchy value (e.g., membership.group.toulouse.opera.fans).

Routing keys are added by producers to notifications' content to indicate the “address” of consumers. When binding a message queue to an exchange, another key is given, called the binding key. This latter key is the parameter used by the exchange to configure the routing protocol implemented by the exchange. Therefore, the routing algorithm uses the routing key of a notification and the binding key of the bound message queues to route notifications to consumers attached to message queues. Note that most of the time, the terms “routing key” and “binding key” are used interchangeably.
5.1 AMQP and RabbitMQ concepts IV

Exchange types.
The notification filtering mechanisms available are “channel-based filtering” and “topic-based filtering”. The notification filtering mechanisms are specified via the exchange types. AMQP specifies three main types of exchanges.

The fan-out exchange type implements channel-based filtering and works as follows:

- A message queue binds to the exchange with no arguments.
- A publisher sends to the exchange a notification.
- The notification is passed to the message queue unconditionally.

The direct exchange type implements a simplistic form of topic-based filtering and works as follows:

- A message queue binds to the exchange using a routing key K. Message queues can bind using any valid routing key value, but most often, message queues will bind using their own name as routing key.
- A publisher sends to the exchange a notification with the routing key R.
- The notification is passed to the message queue if K = R.
5.1 AMQP and RabbitMQ concepts

The topic exchange type works as follows:

- A message queue binds to the exchange using a binding key \( B \) as the routing pattern.
- A publisher sends to the exchange a notification with the routing key \( R \).
- The notification is passed to the message queue if \( R \) matches \( B \).

The routing key used for a topic exchange must consist of zero or more words delimited by dots. Each word may contain the letters [A--Z] and [a--z], and the digits [0--9]. The binding key follows the same rules as the routing key with the addition of * that matches a single word, and # that matches zero or more words. Thus, the binding key `membership.group.#` matches the routing keys `membership.group.tsp.bde` and `membership.group.toulouse.opera.fans` but not `numberusers.group`. This exchange type is stated to be optional in the AMQP specification, and is available for instance in the RabbitMQ implementation.

**Virtual host** A virtual host is a data partition within the broker. A virtual host comprises its own name space with exchanges and message queues. Each connection must be associated with a single virtual host. The protocol offers no mechanisms for creating or configuring virtual hosts — i.e., this is done in an undefined manner within the broker and is entirely implementation-dependent. The authentication scheme of the broker is shared between all virtual hosts on a broker.