CS 446 Architecture Styles: Event-based Architecture

Team #16

Jean-Baptiste Beau (jalbeau)
Karen Lu (kz2lu)
Jay Hyunjae Park (hj8park)

In a nutshell

- In this architecture, some components are generating changes in a system, and others are listening to those changes.
- These changes can be input from a user or a program, changes in data, etc.
- When a component generates a change in the system, it will send a message about the change to a channel.
- This channel will then broadcast this message to other components in the system. The components that are listening for that message can then act accordingly.

Vocabulary: components and connectors

Components:

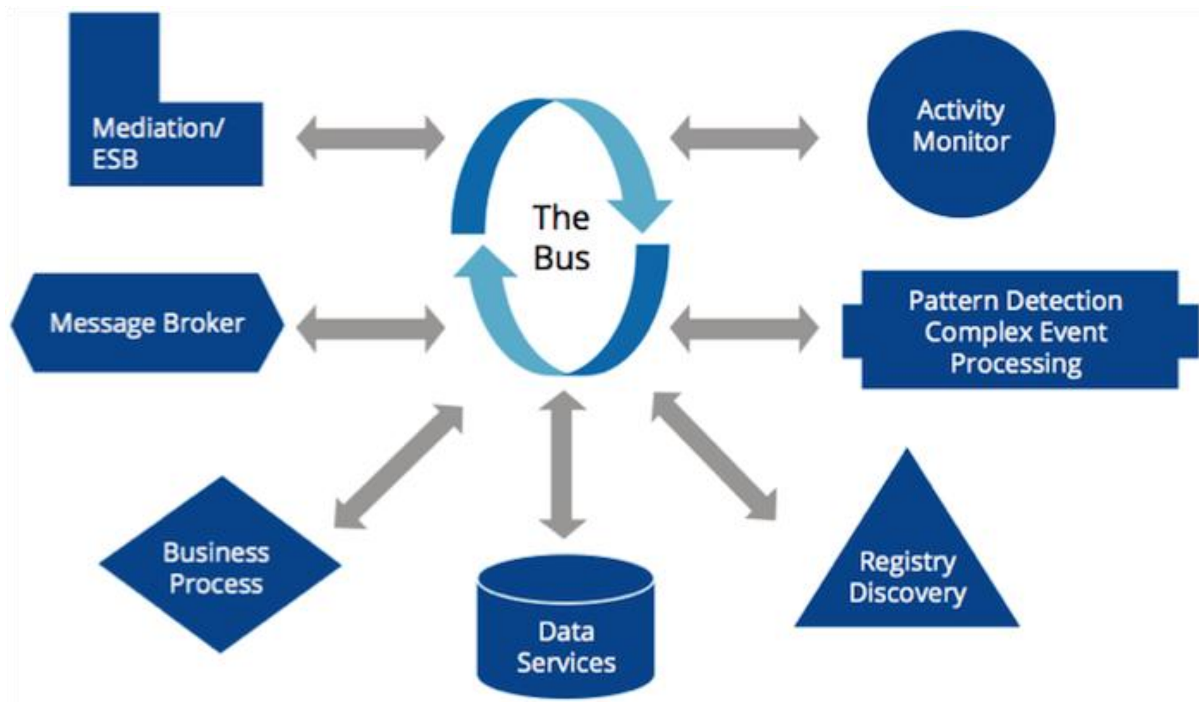
- An **event** is a change in the system.
- An **event notification or message** contains information about the event.
- An **event generator** is the component that generates events and sends event messages.
- An **event consumer** is the component that receives event messages and acts accordingly.

Connector:

- The **event bus or manager** is a channel that receives the *event messages* from *event generators* and broadcast the *event messages* to *event consumers*.

Topological constraints

- The consumers are **listening** for certain event messages from the bus.
- The generators **send event messages** to the bus.
- When the bus receives an event message, it tells the **consumers** about it.
- Implicit invocation: The generators and the consumers **do not know each other and don’t communicate directly**.
- A component can be both a generator and a consumer.



Topology of an event-based system. Every component can be a generator, a consumer or both. They all communicates through the event bus.

Specific kind of problems

- **User Interface:** This architecture is particularly useful for user interface programming. Essentially, it allows the computer to get notified whenever there is a user input, and run without blocking itself. Without the event-based, the program halts until there is an input from a user.

- **Non-Blocking I/O**: In a situation where data read is required either from database, or hard drive, a sequential program will remain pending until the read request is finished and data is returned. With an event-based architecture, however, this can be avoided since the caller can perform other tasks until the event is finished and gets notified.
- **Distributed System**: In a distributed system, this architecture is useful because it allows to easily add and remove components. The different components don't have to know each other and they communicate through the event bus only.

Change resilience (Advantages)

- **Loose coupling**: generators and consumers do not communicate with each other and can be dynamically added or removed.
- **Highly distributed**: any component of the system can be an event generator, an event consumer or both.

Negative behaviors (Disadvantages)

- **Single point of failure**: the event bus is a bottleneck.
- **Semantic coupling**: if the generators change the format of the event messages they send, the consumers must change how they handle the message.
- **No feedback**: cannot guarantee that an event message has been received and processed.
- **Indirection**: there's no direct link between two parts of code. The consumers are not known to the generators and the function calls are implicit, which makes the system harder to debug, update or maintain.
- **Race condition**: the order of the events might not be conserved.

Non-Functional Properties

***Support*:**

- **Scalability**: It's easy to add new consumers and generators. The other consumers and generators don't have to know about it, and don't have to change anything.

- **Adaptability**: If we change how a generator or consumer behaves internally, the other generators and consumers won't be affected.
- **Efficiency**: Cumulative workload is distributed and handled concurrently by many components; therefore, tasks can be completed more efficiently.
- **Reusability**: The components can be reused in several communications: a consumer can handle messages from different generators, and a generator can send messages to different consumers.
- **Complexity**: The components do not need to know about each other.

***Inhibit*:**

- **Complexity**: The asynchronous control flow due to the event-based architecture makes it extremely hard to follow the code, to debug, update and maintain the system.
- **Efficiency**: Due to the additional layer, the event bus, there are more operations than in a simple direct communication. The event bus has to look for consumers and notify them, which might take some time.
- **Adaptability**: If a generator changes the format of the event messages it sends, all the consumers registered for these events have to be updated.
- **Security**: Since the generators and consumers don't communicate directly: the generator can't verify where the message is sent to, and the consumers can't verify where the message is arriving from. Also, if the event bus is corrupted, data integrity is also not guaranteed.
- **Dependability**: Since components depend on the single connector, there exists a high dependency on the event bus. This also means the event bus can become a bottleneck.

Presented example

- The goal of this example is to show the interactions in an *event-based system*.
- Some people are the *event generators*. They will perform some activities and pass *event messages* to the bus.
- One person is the *event bus*. This person will receive messages from the generators and notify the consumers.
- Some people are the *event consumers*. They will receive messages from the bus and process these messages in different ways.

- The generators and the consumers don’t communicate with each other. All messages are passed through the event bus.

Scenario 1. Single Generator & Single Consumer:

- This is a basic scenario of how an event message is created, transmitted and processed.
- A generator shoots a basketball to score a goal. Every time he scores, he tells the event bus.
- The event bus notifies the consumer.
- A consumer writes down the current score.

Scenario 2. Single Generator & Multiple Consumers:

- This scenario is a bit more complex and involves multiple consumers. This is to show that several consumers can listen to the same event.
- A generator shoots a basketball to score a goal. Every time he scores, he tells the event bus.
- The event bus notifies the consumers.
- There are three consumers, each with a written sentence
    - Two of them are listening for the generator to score a goal, and update the score in the sentence.
    - The third consumer, however, is not listening for that particular event so he is not doing anything.

Scenario 3. Multiple Generators & Multiple Consumers:

- This scenario involves multiple generators and multiple consumers. This is to show that one consumer can listen to several events. Also consumers and generators and be added and removed at any time (loose coupling)
- A generator shoots a basketball to score a goal. Every time he scores, he tells the event bus.
- A second generator is getting names from the audience and pass them the event bus.
- The event bus notifies the consumers.
- There are three consumers, each with a written sentence
    - One consumer is listening for the first generator to score a goal, and update the score in his sentence.
    - Another consumer is listening for the first generator to score a goal and the second generator to get a name and will update the score and name in his sentence.

    - The last consumer is listening for the second generator to get a name and will update the name in his sentence.

Scenario 4. Highly Distributed:

- This scenario demonstrates that a generator can be a consumer and vice versa.
- Building on Scenario 1 where there was one consumer and one generator but now their roles are switched
- The generator (who was previously the consumer) writes down a random score and passes it to the event bus.
- The event bus notifies the consumer.
- The consumer (who was previously the generator) shoots a basketball until it achieves the given score.

Scenario 5. Single Point of Failure:

- This scenario shows one of the downsides of the event-based architecture when the event bus is down.
- A generator shoots a basketball to score a goal. Every time he scores, he tells the event bus.
- However, the event bus is down so the consumer is never notified.

Scenario 6. Semantic Coupling:

- This scenario shows another downside when there is a change in the format of the event data.
- A generator shoots a basketball to score a goal. Every time he scores, he tells the event bus. However unlike before, it now sends a ball (different format) instead of a paper.
- The event bus notifies the consumer.
- The consumer is confused and doesn’t know what to do with the ball given.

Sources

Complexity Labs. (2017, July 07). Event-Driven Architecture. Retrieved February 01, 2018, from http://complexitylabs.io/event-driven-architecture/

StackExchange. (n.d.). When should I use event-based programming? Retrieved February 01, 2018, from https://softwareengineering.stackexchange.com/questions/267752/when-should-i-use-event-based-programming

University of Alberta. (n.d.). 3.2.8 – Event Based - Architectural Styles. Retrieved February 01, 2018, from https://www.coursera.org/learn/software-architecture/lecture/BCG96/3-2-8-event-based

Wikipedia. (n.d.). Event-driven architecture. Retrieved February 01, 2018, from https://en.wikipedia.org/wiki/Event-driven_architecture