Event Aggregator

You are building applications using an Event Driven Architecture. You have applications in your architecture that are interested in different types of events. For instance, you may have applications only interested in business events, and other applications that are only interested in high-level infrastructure events. Finally, you have applications that generate many types of events, ranging from the very high level to the very low-level.

In many situations, you have a need to have events of different formats and from multiple sources be fed into the channel for processing. The issue is that often there are disparate categories of events that are processed on the bus. In particular, quite often you see that events can be partitioned into “low” level and “high” level events that are of interest to a disjoint set of applications on the bus.

Consider the following example. In an environment that processes credit card, checking and savings account transactions one of the more interesting issues is how to look for fraud. What you certainly don’t want to do is to send every potential transaction event through the entire fraud-processing application. The transaction volume (in the millions of transactions per day) would overwhelm all other processing. Instead, you want to segment off certain types of events, like purchases over a certain limit, or checks made out to “cash” and then scan them for further processing. Likewise, a sequence of small transactions to the same merchant in a short time period might be an indicator of fraud. So, limiting the total number of events evaluated by some set of criteria is an important consideration, as is scanning for particular patterns across multiple events.

How do we prevent flooding an Event Backbone with many, many events that are not of a type of interest to most other applications connected to that Event Backbone? In other words, how do we keep the events flowing on an Event Backbone pertinent to the applications connected to that bus?

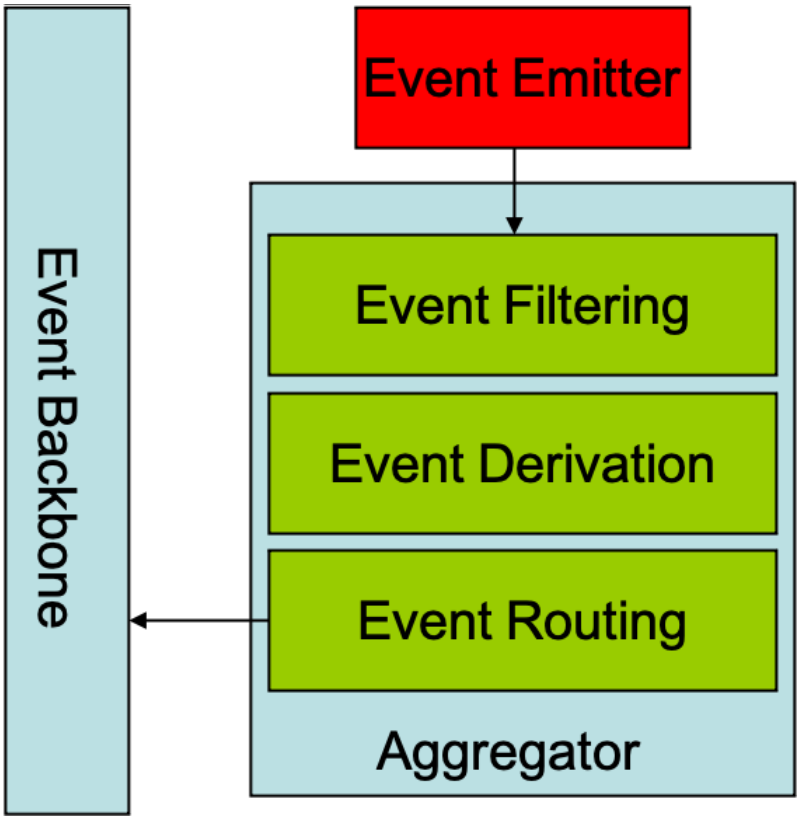
There are many small events which are not of interest to most applications. Specific sequences of these small events are often of interest to business applications.

Therefore,

Define a micro architecture that splits the main Event Backbone into at least two distinct component parts. In the first piece, called the Aggregator, you define a series of processing steps that do the following actions:

1. Capture many disparate, small events from Event Emitters on one topic
2. Perform Event filtering to remove any extraneous or repeated events.
3. Use an Event Deriver to generate new events based on specific patterns of event occurrence and other information.
4. Route the events onto a second topic (for instance Business Events)

This combination of steps is shown in the picture below:



The types of events that are routed to the “main” Event Backbone are of limited and distinct types that are meaningful to applications connected to the main Event Backbone. This division of the Event Backbone will prevent flooding of the “main” Event Backbone by events that are not relevant to other applications.

In effect, applying an Event Aggregator is a form of Federation of Event Backbones. By splitting the backbone into at least two parts - one of which (the Aggregator) is a one-way gate into the other(s), we introduce all the issues that federation involves into the solution. For instance, we must decide how to physically connect the buses, and must address issues of latency and Quality of Service.

Also, if the Aggregator becomes disconnected from the main Event Backbone(s) then we must address the issue of how to detect that and determine how to re-introduce lost events into the main backbone(s). In addition, we are in effect partitioning the events into two different classes - this means that any new applications that depend directly on the low-level events that are filtered and transformed by the Event Aggregator should connect to the Event Aggregator, and not the main Event Backbone.

It is important to note that the filtering and aggregation steps can be implemented in many different programming styles. For instance, while this approach can be implemented algorithmically, it can equally be implemented using a Machine Learning model. The combination of an available historic stream of events (from Event Sourcing) for training and the ability to execute the results of running the Machine Learning model on the real-time stream from the Event Backbone is very powerful for many domains such as bank fraud detection.