Event Pipeline Pattern

Amisha Thakkar & Bina Ramamurthy
(athakkar@cse.buffalo.edu,bina@cse.buffalo.edu)

State University of New York at Buffalo
Amherst, New York

1. Event Pipeline

The event pipeline pattern describes how to dynamically construct a stack of event managers, each providing a unique communication capability as desired by the client or service.

2. Introduction

Events are designed to communicate state changes in any object to those entities that are interested in the state change in the object. The base characteristic of events is that they are asynchronous. Mail roles in an event-oriented system are:

Event Generators: Are objects that generate notifications. Entities (clients) that are interested in receiving these notifications subscribe to this event generator.

Remote Event Listener: Is an object on the client side that can receive notifications.

Remote Events: Is the object that is passed by value to the remote event listener through notification.

Event Manager (Third Party Agent): Is positioned between an event generator and event receiver for processing the event and providing the desired properties.

In a distributed system [1] events are notified to interested entities just as in a local event model. The main difference, however, is that the interested entities may be at remote locations. There are other differences too that have to considered when designing a distributed event model:
In the local case events can be easily delivered in the order they are generated in, this is not so easy in a distributed environment. Partial failures in a network can have a devastating effect on event delivery in a distributed system. But this is not an issue in a stand-alone system. The amount of work involved in sending a local event is small as compared to work done in handling the event. But the situation reverses itself in a distributed scenario. In the local case guaranteed delivery is not a problem but in the distributed environment the receiving entity itself may have crashed. Detection of a disconnected receiver is difficult. To determine whether the disconnection is temporary or permanent is a bigger challenge. In this work we are considering Jini [4] networked distributed systems. Jini event model is quite narrow [5] in that it provides very few classes and methods to manage events.

3. Examples

Consider the following examples to evaluate the properties that an event-receiving client may be interested in. We look at a stockbroker's office.

a. The stockbroker comes to know that stocks of Harry's chocolates just fell and after a few minutes is notified that the stock rose. But actually the reverse had occurred. He/she maybe very unhappy considering the fact that he/she had based his/her decision *on the order in which the events happened*.
b. The stockbroker is not being paid well by his client Jenny. So he/she has decided to not spend too much time on Jenny's portfolio. He/she wants all the information about the stocks jenny is interested just once a day *all bundled into one* document. He/she does not want information for her stocks instantaneously.
c. The stockbroker subscribes to some FlyQuick air travel agency. He/she wants the information from FlyQuick to be out in his/her *mailbox*. He/she would pick it up from there when he wants to or tells the postbox to send it to him periodically. Also even if he/she is out of town he/she still wants all the FlyQuick information *collected and kept safe.* He/she wants all this when he comes back
d. The stockbroker has some friends Joe and Michelle who give him important tips. He/she needs assurance that their *information is always delivered* to him/her.
e. The stockbroker may also be receiving alerts form his banks to keep him posted on his/her current account status or the latest transactions. He/she is *not interested in transactions lesser than $1000*.
f. The stockbroker also refers to some analysts. When each of their trend predictions *falls into a pattern* (which in the past has been very useful) he/she wants to be notified.

4. Problem

Many of the ad-hoc, spontaneous networks used in wireless and mobile applications do not provide a powerful event model such as the one available with Java Swing and AWT for local event handling [4]. But remote event and distributed event handling are equally if not more important in a networked or ad-hoc networking environment. Events maybe more prevalent and perhaps more applicable to a networked environment; the delivery and the handling requirements are so varied and application dependent that it is impossible to provide a set of standard interfaces and classes associated with events. Jini, E-speak and other such spontaneous-networking frameworks do not provide extensive support for event processing. Many applications have to use their own ad-hoc methods and application specific code to satisfy such requirements as store and forward, in-order delivery and efficient delivery. We need a systematic approach to managing event in a lightweight networks.

5. Solution

Keith Edward in his Core Jini book suggests an event pipeline. As a remote event propagates from its source to destination it may be required to satisfy some fundamental needs such in-order delivery and reliable delivery and to apply some application specific operations such as event grouping. We need a general structure that allows selective plug and play of one or more of the event processing requirements.
Some types of event management used often are:
a. In-order delivery

In example *3a* (Section 3) this is what the stockbroker needs. He needs all the events to come in order.
b. Efficient delivery

This is what the stockbroker needs in example *3b*, all events packaged into one event. Another example for use of the efficiency property would be the Mars pathfinder. The cost information transmission is so high that it would be preferable that in some situations the pathfinder could bundle all information in one package and then send it.
c. Store and forward

This property would be very useful for the stockbroker in example *3c*. It would be a really useful service when the client is unavailable (just like the answering machine which stores information when the client is unavailable).
d. Reliable delivery

Reliable delivery could be very important in some applications where the delivery has to be guaranteed. In this case we would some acknowledgement from the recipient about the delivery. Reliable delivery of information is what the stockbroker would want in example *3d*.
e. Filtering of events

The capability of providing semantic filtering is what the stockbroker would want in example *3e*. He/she could use this kind of filtering for any semantic they need-for instance on some day he/she wants to look at updates on only steel stocks and not on internet company stocks.
g. Grouping of events

Many events can be semantically replaced by one event. This what the stockbroker would like in example *3f*. This to forms an efficient form of delivery but the difference is in the fact that semantically meaningful events are grouped together and not all.

Our solution is an event-pipeline with one or more stages, each stage managing one of the above listed functionality. Pipeline pattern allows for dynamically altering the constituent stages and the order of the stages. This may not be possible in a non-pipelined solution. We present this solution as a pattern [2] since we see many more uses for this approach in the other networked systems.

6. Pattern

Jini provides a composable event model that one event listener can feed into the other. A tandem of event listeners makes up the event pipeline. Figure 1 shows an event pipeline placed between the source and the target of events. (There is a similarity with the chain of responsibility pattern).

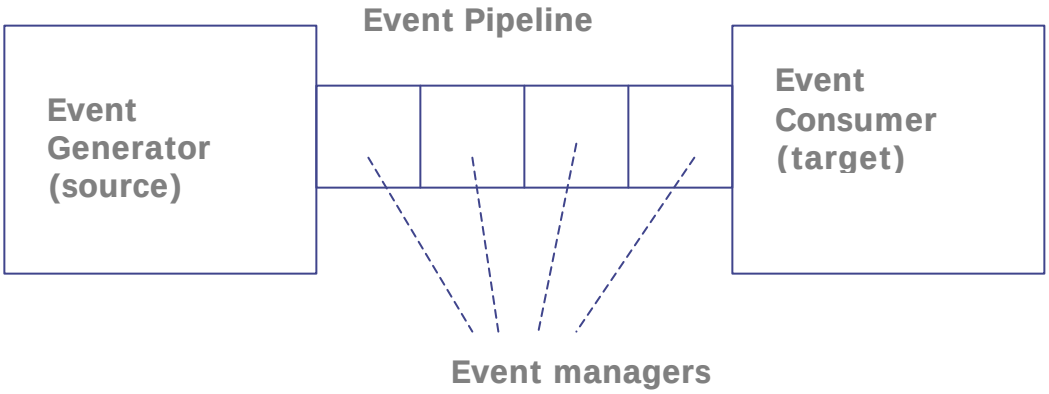

Figure 1. Event Pipeline

Event pipeline just like an instruction pipeline has pluggable stages one feeding into the next. We consider only a linear pipeline pattern. Non-linear patterns can be considered for complex event-oriented applications.

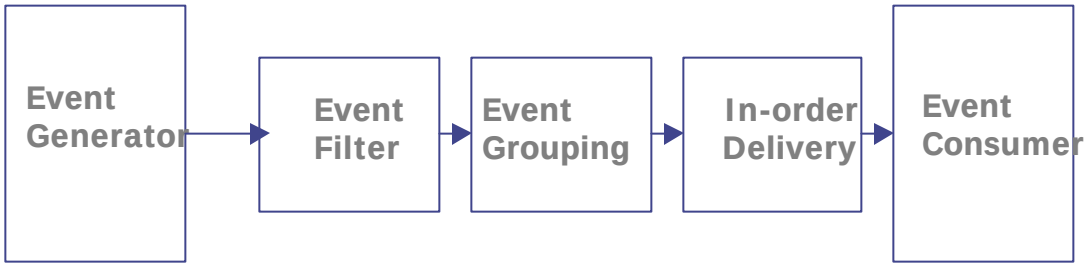

Figure 2. Stages in a Sample Event Pipeline

Figure 2 shows a pipeline with three stages; Event Filtering, Event grouping and In-Order delivery. Code snippets associated with the design and implementation of these stages and usage are given in the next sections.

7. Code Snippets

Code snippets illustrating the concept of the event pipeline are shown in Figure 3 below.

```
Class FilterStage  extends UniCastRemoteObject
                implements RemoteEventListener, Runnable
{
    public FilterStage( ) throws RemoteException
    { }

    public void notify(RemoteEvent e)
    {
            /*Events get notified  through this method */
            ……..
    }

    public void DeliverToNextStage(….. RemoteEventListener nextStage)
                            throws RemoteException
            {
                    try {
                            nextstage.notify((RemoteEvent)(filteredevent));

                            :
                            :
                        }
                    catch…….
            }
    ……..
    }
```

Figure 3. Definition of a pipeline stage

8. Usage

The stages can be added to the service side or client side. For example, StoreAndForwardStage would be on the client side whereas EfficiencyStage can be on the server side. We will assume that lookup service is the event generator and that events have to be delivered filtered, grouped and in-order to the client (as shown in Figure 2). Client code snippet is shown Figure 4.

```
Class EventConsumerClient implements RemoteEventListener
{
        //instantiate the stage closest to the client

        InOrderDeliveryStage iods = new InOrderDeliveryStage(this);

        //next closest to consumer
        GroupingStage gs = new GroupingStage(iods);

        Filterstage  fs = new FilterStage(gs);

        //connect the event source to first stage so that events are delivered to the first
        //stage of the pipeline

        EventRegistration erg= lookup.register(……,fs);
}
```

Figure 4. Client Code Snippets

9. Uses

a. CORBA - In CORBA event service provides no support for filtering events or even specifying the delivery requirements. An event pipeline can be built to handle such requests. The properties for the communication channel can be provided this way.

b. Wireless Network – In a wireless network different kind of events needs to be handled and they can be handled using the event pipeline. This allows the user to incorporate the kind of properties desired.

c. Mobile Network – In a mobile network where the connectivity is provided on the fly, building in the properties provided by the application channel will enhance the reliability.

d. Device Network – In device networks when a printer is down or a scanner is mal-functioning it will be easier to notify the consumer using the event pipeline.

e. Appliances Network – In a non-computing environment where devices may lack the intelligence proxies on their behalf could generate and consume relevant events. The event pipeline in this case again serves to add the properties as required.

10. Future Work

A full implementation of event pipeline is planned. We also plan to study the efficiency of introducing stages with the raw event delivery of events. In an “eventful” system pipeline could mean parallel processing too!

11. References

1. G. Coulouris, J. Dollimore, T. Kindberg: Distributed Systems - Concepts and Design, Addison-Wesley, 1994.

2. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns - Elements Of Reusable Object-Oriented Software, Addison Wesley, 1995.

3. C. Hortmann, G. Cornell: Core Java, Sun Microsystems Press, 1999.

4. W. Keith Edwards: Core Jini, Sun Microsystems Press, 1999.

5. Jan Newmarch, Jan Newmarch's Guide to JINI Technology, 2000 http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml

HERE platform pipelines are designed to accommodate specific usage patterns. The available patterns are illustrated below, starting with the simplest pattern and progressing to more complex use cases. Additional information is provided throughout the Developer Guide. General Pattern. This is the general pattern for using a pipeline. Figure 1. Basic pipeline pattern. Note: Data sources and sinks. A specific catalog layer can serve as a source or a sink, but never both at once. Goal : Build a event pipeline that can accept arbitrary JSON messages and store it in S3. Photo by Scott Webb on Unsplash. Note: Check out our latest Serverless Smart Radio series.Â AWS API Gateway : Accepts events from different sources and forwards to the Kinesis. AWS Kinesis Where events sequenced and ready to be consumed (stored in disk). AWS Kinesis Firehose : Consumes the Kinesis stream and serialize into a destination in batches of N minutes or N megabytes. AWS S3: Store all the events in batches. Asynchronous pipelines. Testing event-driven apps. In a previous article for this series, I covered the topic of asynchronous methods: methods or functions that â€œreturnâ€ a value by passing it to a callback instead of using the return keyword.Â This isnâ€™t a pattern Iâ€™ve used an awful lot, but it solved a problem on Songkick really nicely last week. We have various buttons on the site that let users start tracking things, for example to start following an artist or say theyâ€™re going to a concert. These buttons typically are forms that submit using Ajax. Pipeline events are JavaScript entry points to perform user-defined actions. API Server calls the following types of pipeline events for every request: Request pipelines. API Server invokes these pipeline events. after. it receives the request but.