THE STREAM EVENT PROCESSING (EP) SYSTEM PROJECT

A STARTER GUIDE

StreamEPS

eps PLATFORM

ARCHITECTURE GUIDE

v1.5.8

Author: Frank Appiah

Project: StreamEPS

Version: v1.0

© 2011

Table of Contents

Preface....................3
1. 1.Architectural Overview....................5
2. 2.Engine Architecture....................6
    2.1.Engine Core 6
    2.2.Engine Context : org.streameps.context....................6
        a)Segment Context....................6
        b)Temporal Context 7
        c)State Context 8
        d)Spatial Context 8
    2.3.Engine Processing : org.streameps.engine....................8
        a)Threading Support : org.streameps.thread....................8
        b)Logging Support 9
        c)HistoryStore....................9
3. 3.Core Functionality....................10
    3.1.Event Aggregation....................10
    3.2.Event Pattern Matching....................10
    3.3.Event Filtering....................11
        a)Extending the filter processor....................12
    3.4.Event Assertions....................12
        a)Classic Assertions....................12
        b)Modal Assertions....................12
        c)Trend Assertions....................13

Preface

This is an open source event stream processing system or platform that provides an engine for processing segment-oriented, temporal-oriented, state-oriented and spatial-oriented event stream. It provides support to other processing systems via processing element adapter, PE. It supports distributed, scalable, partially fault-tolerant properties and allow developers to easily develop applications for processing continuous unbounded streams of data. This event processing project is modelled on an event processing model, which has a core base supporting some common features in event processing engines.
The platform currently supports the following processing pipelines that is provided by the engine. These processing include the evaluation of event assertion functions; the basic operators, logic operators, modal operators and trend assertions:- increasing, decreasing, non-increasing, non-decreasing and mixed trends; pattern match detects; event filtering :- comparison filter, range filter and in-not value set filter.

Every basic event atom show minimally implement or extend the EventObject. The event object is an event with just a payload content with Payload class type and it also extends the header in order to inherit the properties/attributes of the header object. The header type is made up of a 9-tuple structure <*identifier*:String, *isComposable*:boolean, *chronon*: ChrononType, *occurrenceTime*:Date, *eventCertainty*:Float, *eventAnnotation*:String, *detectionTime*:Date, *eventSource*:String, *eventIdentity*:String>. The other events types in the StreamEPS project includes the following interface classes IStreamEvent, IDatabaseEvent, IPrimitiveEvent, an IEncrpytedEvent and a special event called ICurrentTimeEvent which carries along its payload the current time of the user system. The listeners used in the guide are the TestPatternMatchListener and TestUnPatternMatchListener which will explained in the subsequent sections. These are used to observe the events that are matched and unmatched with latter using the TestUnPatternMatchListener. The matched and unmatched events are stored in a IMatchEventMap and IUnMatchEventMap respectively.

This document guide is **not** fully complete and will deem it necessary that any errors should be forwarded to the author for immediate correction. It also does not adhere to any publishing house and it is solely the authors own work.

1. Architectural Overview

This section provides the architectural overview of the StreamEPS by a layer view approach as shown below. It depicts the building blocks of StreamEPS more importantly. The major architectural decisions taken will not be exhausted here. Please refer to the other document on the conceptual view of StreamEPS.

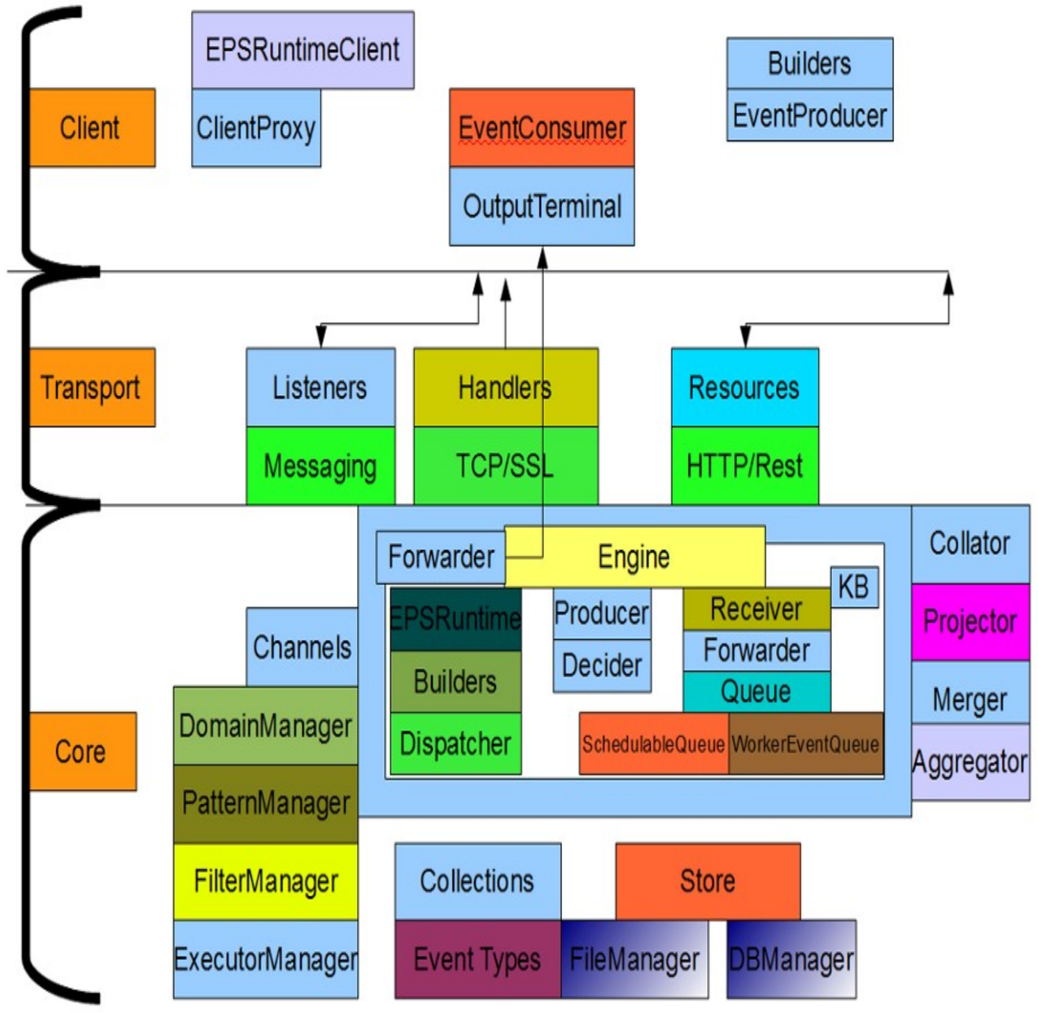

Figure 1: The building blocks of StreamEPS

2. Engine Architecture

This section describes into details the overall functionality of the engine in StreamEPS. It will discuss the engine core, contextual framework of the engine, engine processing , threading support, logging support, stores etc.

2.1. Engine Core

The bare core of the engine is made up of 8 major components namely the engine processor, receiver, EPS producer, forwarder, decider, event producer, event channels and the store. Conversely, a software system is said to be event based if its parts interact primarily using event notification to actuate events from external sources to detect patterns in the event stream. A system designed with StreamEPS provides easy build, test and maintainable code structure for any project. An event based system reduces the overall complexity of a system. An event channel is a proxy to the event producer of the client and client output terminal which comprises of the a list of channel input terminals and channel output terminals.

2.2. Engine Context : org.streameps.context

The core contextual framework are namely temporal-based, state-based, segment-based and spatial-based. Currently, only two contexts are supported which are the segment and temporal contexts. The main classes in the package (*org.streameps.context*) are:

- IContextDetail
- IContextDimType
- IContextPartition
- IPartitionWindow
- IPredicateTerm
- IContextInitiatorPolicy

A context initiator policy defines the behaviour required when a window has been opened and a subsequent initiator event is detected. The semantics are supported as defined in the specific policies which includes *add*, *ignore*, *refresh* and *extend*. The extend policy defines a time-out processing where the expiration event count or expiration time offset is reset to start with the new initiator event. Each context in the engine context extends the IContextDetail and the IContextParam. The context parameter interface is a parametrized class which the takes the specific parameter for a specified context.

a) Segment Context

A segmentation-oriented context assigns events to context partitions (defined by *org.streameps.context.IContextPartition*) based on the values of one or more event attributes, either using the value of these attribute(s) to pick a partition directly, or using predicate expressions to define context partition association. The interface *ISegmentParam* provides the container to keep all the event attributes and the event predicate expressions used in defining the context partition for a specific segment. The main classes in the package (org.streameps.context.segment) are:

- *ISegmentContext*
- *SegmentContext*
- *ISegmentParam*
- *SegmentParam*

b) Temporal Context

There are four main temporal contexts namely the event interval, sliding event interval, fixed interval and sliding fixed interval.

- IEventIntervalContext: The context specification includes an initiator event list, and at least one of the following: terminator event list, expiration time offset, or expiration event count. If an expiration time offset is specified, the temporal ordering parameter must indicate ordering by time stamp.
  The main classes in the package (org.streameps.context.temporal) are:
  - *IEventIntervalContext*
  - *IEventIntervalParam*
  - *EventIntervalContext*
  - *EventIntervalParam*
- IFixedIntervalContext: The fixed interval context is used to represent either a single fixed-length time period, or a fixed-length time period that repeats in a regular fashion. In a fixed interval context each window is an interval that has a fixed time length.
  The main classes in the package (org.streameps.context.temporal) are:
  - *IFixedIntervalContext*
  - *FixedIntervalContext*
  - *IFixedIntervalContextParam*
  - *FixedIntervalContextParam*
- ISlidingFixedIntervalContext: Each partition window in the sliding fixed interval context has a fixed temporal size or event count. New windows are opened at regular time intervals relative to each other.
  The main classes in the package (org.streameps.context.temporal) are:
  - *SlidingFixedIntervalContext*
  - *ISlidingFixedIntervalContext*
  - *SlidingFixedIntervalParam*
  - *ISlidingFixedIntervalParam*
- ISlidingEventIntervalContext: In a sliding event interval context the opening of each new window, and its duration, is determined by counting the number of events received by the event processing agent.
  The main classes in the package (org.streameps.context.temporal) are:
  - *ISlidingEventIntervalContext*
  - *SlidingEventIntervalContext*
  - *SlidingEventIntervalParam*
  - *ISlidingEventIntervalParam*

c) State Context

d) Spatial Context

2.3. Engine Processing : org.streameps.engine

The engine is designed based on an information flow processing (IFP) approach that provides a tool capable of timely processing large amount of information as it flows from the peripheral to the centre of the system. The engine receives such events as input and processes them, as soon as they are available, according to a set of processing rules or standing queries, which specify how to filter, combine and aggregate different streams of event, event by event, to generate new subset of events, which represent the output of the engine for the Client Terminal. The engine for the event processing system supports queued events, asynchronous and synchronous dispatch of events via an indicator set by the developer. A specified two-callback constructs used in the design are the *onContextReceive* and *pushContextReceived*. A callback is a function that is called when ever or where ever a new event is received from an event producer.
When an event is send to the engine and it is asynchronous, then the event is immediately added to the queue and the context partition is built. If the *saveOnReceive* flag is set during the engine building process, then it is added to the audit store for the participant events. On the other hand, if it is not asynchronous, it is rather added to the queue of the receiver. At the receiver side, it's *onReceive* method populates the worker event queue and if the sequence size is less than 1 after each decrement on event received, then the context partition is built. At the worker queue side, the event is added to the internal queue and the count down latch is count down until less than 1. If it is less than 1, then the count down latch is re-initialized and the events are dispatched by registering a new dispatch to the dispatcher service.

```
getDispatcherService().registerDispatcher (new Dispatchable() {
            public void dispatch() {
               accumulatorRef.get().getMap().remove(lastEventID);
               IEPSReceiver receiver = receiverRef;
               receiver.buildContextPartition(receiver.getReceiverContext(), tempEvents);
               receiver.pushContextPartition(receiver.getContextPartitions());
            }
 });
```

The context partitions are then pushed to the back to the receiver and the *onContextReceive* of the decider is called to forward the partitions for further processing like pattern matching. The partitions are set to the decider pair and the decider then decides on the decider pair created by the *decideOnContext* method call.
In the *decideOnContext* method, the chain of pattern detectors are retrieved from the decider pair and iteratively decide on the context partition. The internal pattern match and un-match listeners are used to receive the pattern match. The match decider context is then sent to the producer and the

knowledge base. The un-match decider context is immediately saved to the decider store location on the file system. If the *saveOnDecide* flag is set to true then the match decider context will be successfully saved to the match context location.
On match decider context received at the producer side:

- It sends the decider context to the registered decider context listener: IDeciderContextListener.
- It triggers the producer to create the filter context from the decider value and which is forwarded to the forwarder.
- If the aggregate flag is enabled, it then generates the aggregate which is published to the listeners.

The forwarder will receive the filter context which will then forward to the output terminals registered to listen for filter context values. The forwarder also forwards the filter context to the channel output terminal registered with this forwarder.

a) Threading Support : org.streameps.thread

Multi-threading helps improve the performance of event processing systems. In the engine, there is an executor thread manager provided by interface *org.streameps.thread.IEPSExecutorManager*. A worker callable can be submitted or scheduled to be executed with the following additional details: the *initial-delay* for the execution, the *next execution period* details and the *time unit* (microseconds, seconds, milliseconds, nanoseconds, minutes, hours and days) to describe the time values. Consequently, a worker callable can also execute either at fixed rate or fixed delay. A fixed rate execution creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initial-delay then *initial-delay + period,* then *initial-delay + 2 * period*, and so on.
A fixed delay execution of a worker callable process creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next. A task in the executor manager will only terminate via cancellation or termination of the executor. The manager also provides a facility to shut-down all the worker callable(s) in the queue, next for execution. There is an adapter which wraps around the worker callable (*org.streameps.thread.IWorkerCallable*) and the future result queue (*org.streameps.thread.IFutureResultQueue*).
The actual execution object passed to the *EPSExecutorManager* is the CallableAdapter which implements the Runnable class. It functions as both a concrete implementation of the execution and also adds the result unit to the queue. The FutureResultQueue is just a container to keep track of all worker callable processes and helps to implement a functionality where a task has complete before the next set of worker callable(s) can be dispatcher to the executor manager.

b) Logging Support

c) History Store

3. Core Functionality

3.1. Event Aggregation

The implementation of an event stream aggregation provided by StreamEPS supports the following aggregation functions like count, minimum, maximum, mode, etc. Unimplemented functions can be implemented by extending the *IAggregation* interface. The aggregation interface is a parametrized class of two parameter types;

- <*T*> : The accumulator for the aggregation process
- <*S*>*:* The return type after the aggregation process.

There are two main process phases of the aggregation process that includes *process* callback which is basically consistent of a sub-task of two other processes: the *executeInitiator* process call and *executeIterator* process call and the *output* callback which just involves the *executeExpirator* process call. All these processes takes place in a processor called *EventAggregatorPE* which implements a policy called an AggregatePolicy. The aggregate policy is a special user-defined policy that is evaluated at each point in the three sub-processes namely the *execute** processes to serve as a further separation of concern to the user of this processor. At the *executeInitiator* point of execution, the specified *property name* of event is used to extract the *property value* of the event from the stream source and return it to the next executor. For the *executeIterator,* the return value extracted from the event in the earlier *execute* process is then populated into the *accumulator* which is sorted. Finally, the accumulated value map returns a list of only the value set, which is iteratively processed by the aggregation instance defined in the processor. After the aggregation process evaluation, the aggregate is then passed to a listener set to propagate the aggregate result to the client via the set listener. This listener is called the *AggregatorListener* that has only two methods with the most important called *onAggregate* callback. The second is a mutator for setting the aggregate context with the interface IAggregateContext. The populate-sorted accumulator is cleared and the *executeExpirator* of the aggregate policy then invoked. It is recommended that any further extension of the aggregation processing element follows the same functional breakdown of processes.

3.2. Event Pattern Matching

The pattern matching processor supports an enumerated list of patterns defined by the PatternType including highest subset pattern defined by the class, HighestSubsetPE; lowest subset pattern defined by the class, LowestSubsetPE, a threshold average pattern created by the class ThresholdAveragePE (this is part of the collection of aggregation pattern detect processors), a pattern detect for a trend with a thorough description in the earlier sections, a logical and modal patterns also described above. There is a special pattern processor called a function pattern, which provides an interface, IFunctor for the developers to create functions not defined in the platform. This function pattern processor is one of the other kind to the aggregation pattern which lets the

developers to decide and make a pick to suit its need.
On the functional point, there is a so-called null pattern which is used to serve events from the stream source without any processing or so ever. This is defined in the org.streameps.processor.pattern.NullPattern class to provide this special functionality. This is really needed in the engine processing pipeline and you can find further information in the programmer guide, which provides more detail and important info on these pattern compositions. Each of these patterns extend the IBasePattern which provides the listeners (IPatternMatchListener and IPatternUnMatchListener) for clients to observe both the pattern matched and pattern unmatched activities, adding pattern parameters, getting access to the participant events/ matched events by a pull process, a define list of pattern policies and a user defined pre-post aware but defaulted to the class DefaultPrePostProcess. The pattern policies include the cardinality, consumption, evaluation, order and repeated policy.

There are two processor entries in the pattern matching detects for getting the functionality as specified by the pattern processor. The pattern processor with the interface, IPatternProcessor provides the interface/contract for specifying pattern matching processes defined by the contract method, *process*. There are two process methods one to iteratively add events to the participant event set provided by the class *IParticipantEventSet* and the other with the instance of the *SortedAccumulator*. These collections are provided in the org.streameps.aggregation.collection package. The second parameter of the *process* method for each process takes the implementation of the IBasePattern. The flexibility of this type of processor is that it allows the developer to set the listeners which is not possible in the IPatternManager. On the contrary, the IPatternManager interface provides the same functionality with additional pattern chain pipe but the matched event and unmatched events are to be pulled by the developer if not listeners are added to the list of pattern listeners.
@see The package org.streameps.test.HighestPatternTest for sample code in the project page.

3.3. Event Filtering

The core of event filtering in StreamEPS is made up of these three components namely the filter expression, the value set and the filter processor. The filter expression component is provided by the following interface classes IEventContentFilterExprn, IEventHeaderFilterExprn and IEventTypeFilterExprn with each extending an abstract core interface IPredicateExpr defined in the package org.streameps.filter. There are three value sets which implements the interface IValueSet to provide the unique identifier. The comparison value set provided by the interface IComparisonValueSet consists of a partition window of events sorted in an accumulator. The range value set with has an interface class defined by the IRangeValueSet with similar structure content as the comparison value set. The third value set is the in-not value set with its interface class InNotValueSet. The architectural decision is to place each value set into its bounded-context latter explained. Lets move on to the filter processing of events in the platform, StreamEPS. A filter in StreamEPS has a common *filter* method for the filtering process of the context value set provided. The filter process is provided by the interface IEPSFilter which has an abstract implementation called AbstractEPSFilter. It provides the developer an opportunity to implement custom filters with ease. It is recommend for the developer to extend the abstract filter class, AbstractEPSFilter instead of the IEPSFilter interface. The *filter* method takes a special context value set parameter called the IExprEvaluatorContext. The context value set has a filter type for determining the abstract filter type to use, a filter operator that is used in the evaluation of events , the context entry which provides the predicate expression and terms and most importantly, the parametrized event container which is defaulted to the sorted accumulator provided by the interface ISortedAccumulator. The filter type is an enumeration of string group values ; comparison, range and in-not values and the filter operator is made up of similar enumerated string values that includes the comparison group; *less than*(lt), *greater than*(gt), *equal*(eq), *greater than or equal* (geq) and on and on, the range group; *range close*([a,b]), *range open*((a,b)), *range half open* and *half closed*, *in list values* and finally the in-not value group that provides *not range*(half-open, half-closed, close, open).

a) Extending the filter processor

In order to implement new filters for the platform, the developer has to look at these

- The filter expression component to use.
- The specific value set as the context holder with its event container which the platform has provided more. A referent to the context holder draws the developer's attention to the IExprEvaluatorContext.
- Extending the AbstractEPSFilter for your custom filter.

3.4. Event Assertions

a) Classic Assertions

Classic event operators that involves conjunction, disjunction, negation and a compound mix-in of any of the event operators:
A conjunction event operator simply asserts the logic of an event that satisfies a list of pattern parameters after the assertion valuations are performed on the event.

b) Modal Assertions

Modal event evaluations involve the assertion of *always* and *sometimes* operators. The *always* assertion must satisfy all the pattern parameter list on the event to be evaluated on. In evaluation terms, an always assertion operator is equivalent to the classic event conjuncture. Conversely, the sometimes assertion operator will evaluate the event with pattern parameter list to determine whether one or more of the parameter's is satisfied. Similarly, there are three modal compound event assertions namely the *andCompounModal*, *notCompoundModal* and the *orCoumpoundModal*. The compound modal is a mix-in of *always* and *sometimes* modal operators with a logic operators.

c) Trend Assertions

Trend Assertions provide the evaluation of events from a stream source to assert whether or not it is increasing, decreasing, non-increasing, non-decreasing or mixed which each will require that is the trend object is provided. The trend object interface, IT*rendObject* consists of an *objectID*: Time-stamp, *attribute:* String, a *trend list:* List which is the schema property of the set of participating events from the stream source extracted from the property attribute defined in the trend object.