jKool Streaming

Terms, Concepts & Architecture

Table of Contents

Introduction 1

jKool Streaming Architecture 1

jKool Data Model 2

jKool Data Types 3

- Activity, Events, Properties 3
  - jKool and Duration Times 3
- Snapshots 4
  - Reason 1 for Using Snapshots 4
  - Reason 2 for Using Snapshots 5
- Correlators 6
  - Stitching 6
  - Example 1 6
  - Example 2 7
  - Example 3 8

jKool Fields 11

jKool Streaming Ecosystem 14

- Streaming for Developers: 14
  - Restful API 14
  - Java Event Streaming Library (JESL) 14
  - Streaming Agent Framework (tnt4j-streams) 14
- Streaming for Specific Data Sources: 14
  - Stream Simulator 14
  - Log4j (tnt4j-log4j12) 14
  - Syslog (tnt4j-syslogd) 15
  - Java JMX (tnt4-stream-jmx) 15
  - Java Servlet Filter Tracking (tnt4j-servlet-filter) 15
  - Java Method & Transaction Tracing (tnt4j-streams-zorka) 15
  - Java GC (tnt4j-stream-gc) 15

Contact Us 15

Introduction

jKool is a Unified Application and Fast Data Analytics platform for analyzing machine data such as logs, metrics, performance, transactions – all things time series. jKool can be used to analyze time series machine data from sensors, software, mobile devices as well as complex multi-step transactions. Just stream your data and let jKool store, index and analyze your data. To stream, you'll have a choice of specific collectors as well as APIs that are described at the end of this document.

This document will familiarize you with the jKool streaming model. Its intent is to get you comfortable with some very basic concepts so that you understand how to structure your data for streaming.

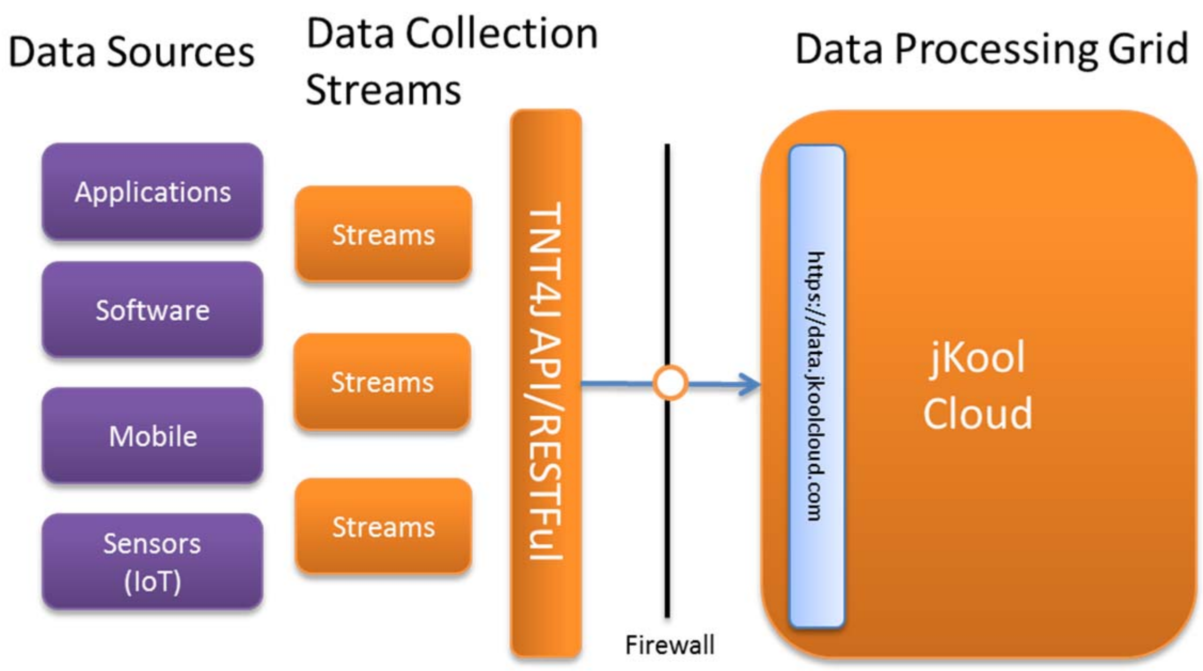
This document and its corresponding sample code uses simple examples in order to explain jKool concepts and methods. These methods can be combined in sophisticated ways to gain meaningful insight into your data, flows, performance, and trends.

jKool Streaming Architecture

The jKool streaming model consists of:

1. Data sources -- such as applications, devices, software, mobile, IoT sensors
2. Streams – software adapters that provide conduit between Data Sources and jKool
3. Parsers – data parsers that parse underlying streams and convert into jKool model (some streams have imbedded parsers for specific data sources)

A high-level view of jKool Streaming Architecture is shown below.



jKool Data Model

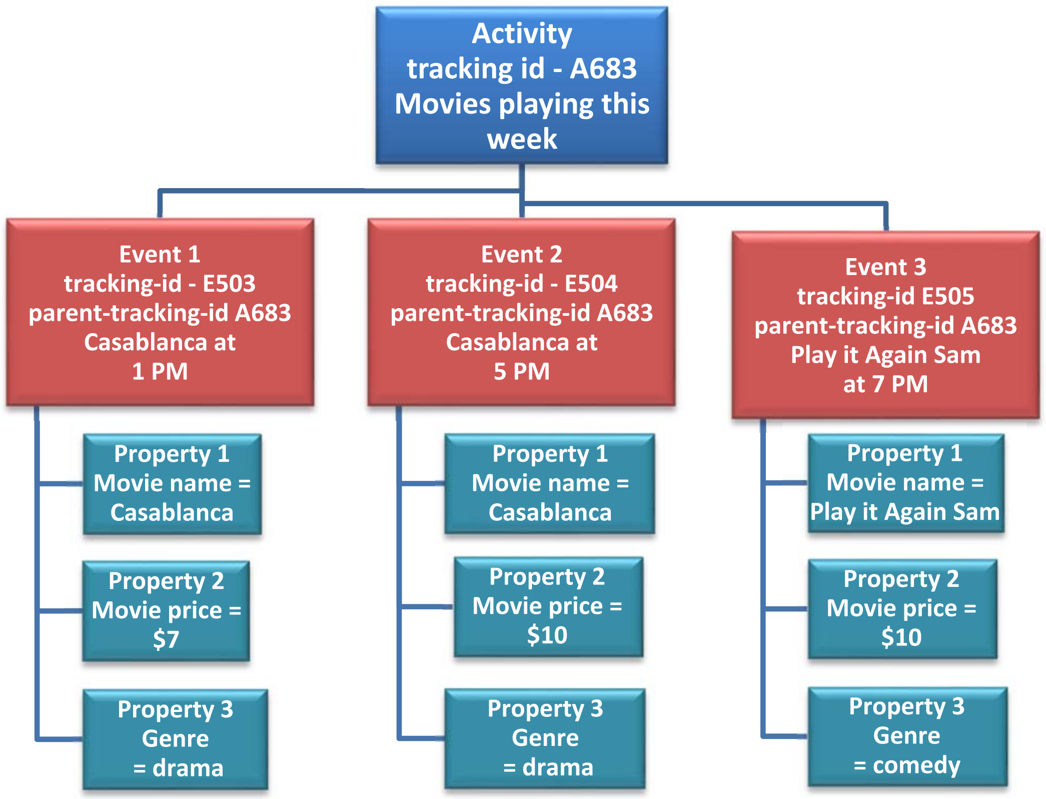
To understand the jKool data model, you must understand the following concepts:

- **Events** – Actions or occurrences recognized by software; something that happens. Each event has predefined and/or custom fields. Each event has a tracking-id. Tracking-ids can be shared among events if they are related to each other.
- **Activities** – A collection of related events. Each activity has a unique tracking-id. An activity contains related events that are streamed over a period of time. jKool also creates activities by relating data based on correlators (please see the section on Correlators).
- **Properties** – Used to define your "custom" event fields. Each property is defined by a name, a value, and a data type.
- **Snapshots** – Snapshots are a way of grouping or associating properties to data that occurred at a point-in-time and are outside the scope of the original data set. Snapshots attach to events or can be reported stand-alone. Properties, in addition to attaching directly to events, can also attach to snapshots.
- **Correlators** – IDs that allow you to connect ("stitch") events that flow from different data sources and are related (such as a business transaction and web transaction).

jKool Data Types

Activity, Events, Properties

- **Activity** = Movies playing this week (group of related events or other activities)
- **Events** = Specific movies playing at specific times (low level events)
- **Properties** = Movie name, movie price, and genre (name, value pairs)



Please notice the tracking-ids in the above example. Associate an event to an activity by putting the activity's tracking-id into the event's parent-tracking-id field.

jKool and Duration Times

Activities and events each can have durations. Although not depicted in the above diagram (for diagram simplicity), the duration of the activity is the week (i.e., 11/1/15 – 11/7/15) and the duration of the events are the playing times of the events (i.e., 2 hours if the movie played from 1:00 to 3:00). Events don't have to have durations; they can be events that occur at a point in time. In that case the duration of the event would be zero.

Having to deal with durations and tracking-ids can be a little tricky. For instance, how does one stream an activity, then stream the events that comprise that activity if we do not know when the activity will end? We know when the activity ends in the above movie example, but this is not always the case. For instance, if we are streaming an order being processed by a computer system, we have no idea when the event will actually end. To handle unknown activity end times in jKool, stream the events first, then stream the associated activity when its events are finished streaming. If however you know the end-time ahead of time, you can stream it prior to streaming its events. In either case, what is important is that you associate the event to the activity via the parent-tracking-id. So even if you have not yet streamed the activity, note what its tracking-id will be and store it in its events parent-tracking-id field. Then, when the activity ends and you know its end time and duration, stream the activity.

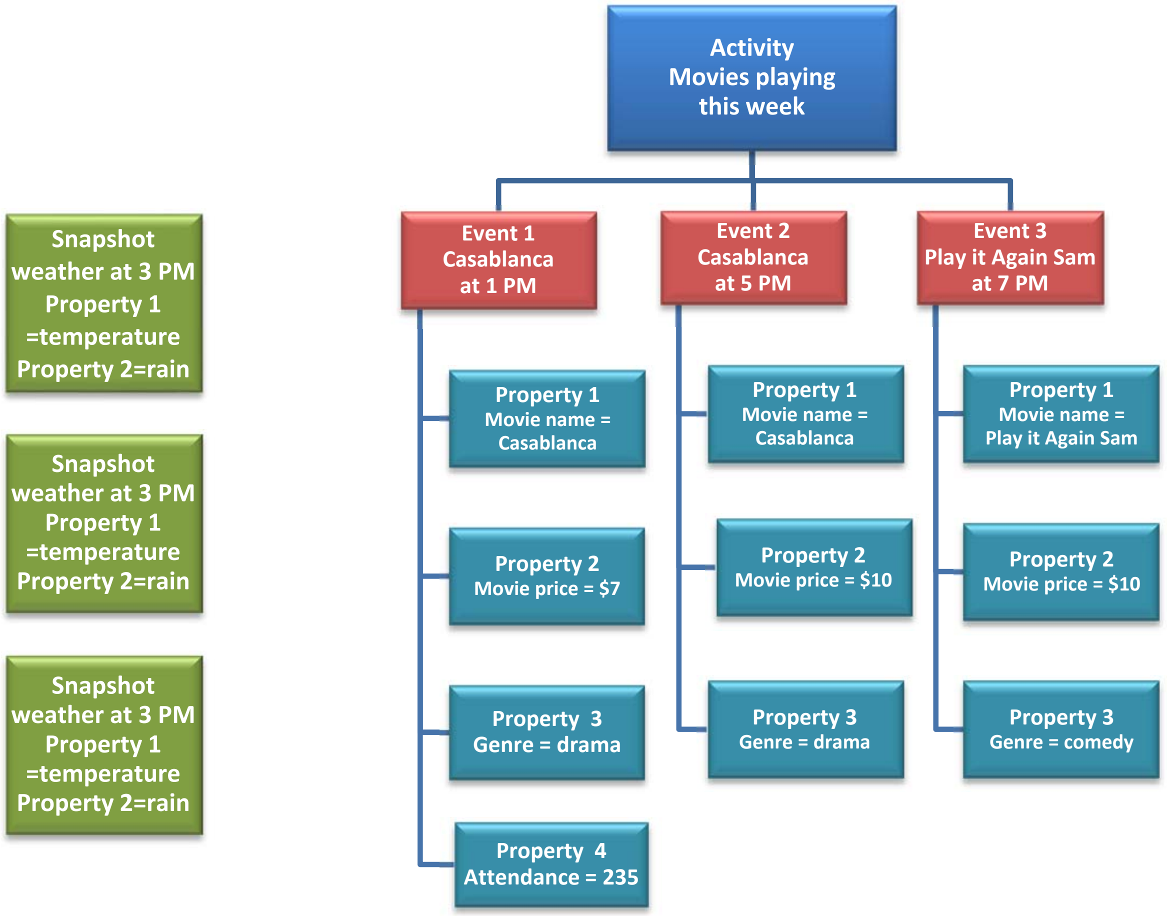
Be advised that you do not need to stream activities. jKool will work without activities. Use Activities when relating multiple events into logical groups – examples transactions. If not using activities, simply set the event's parent-tracking-id to null, unless you are building some type of "event hierarchy" and wish to give it another "event's" tracking-id as the parent-id.

Snapshots

Think of a snapshot as a sample of contextual data at a "point-in-time." It is used for one of **two** reasons.

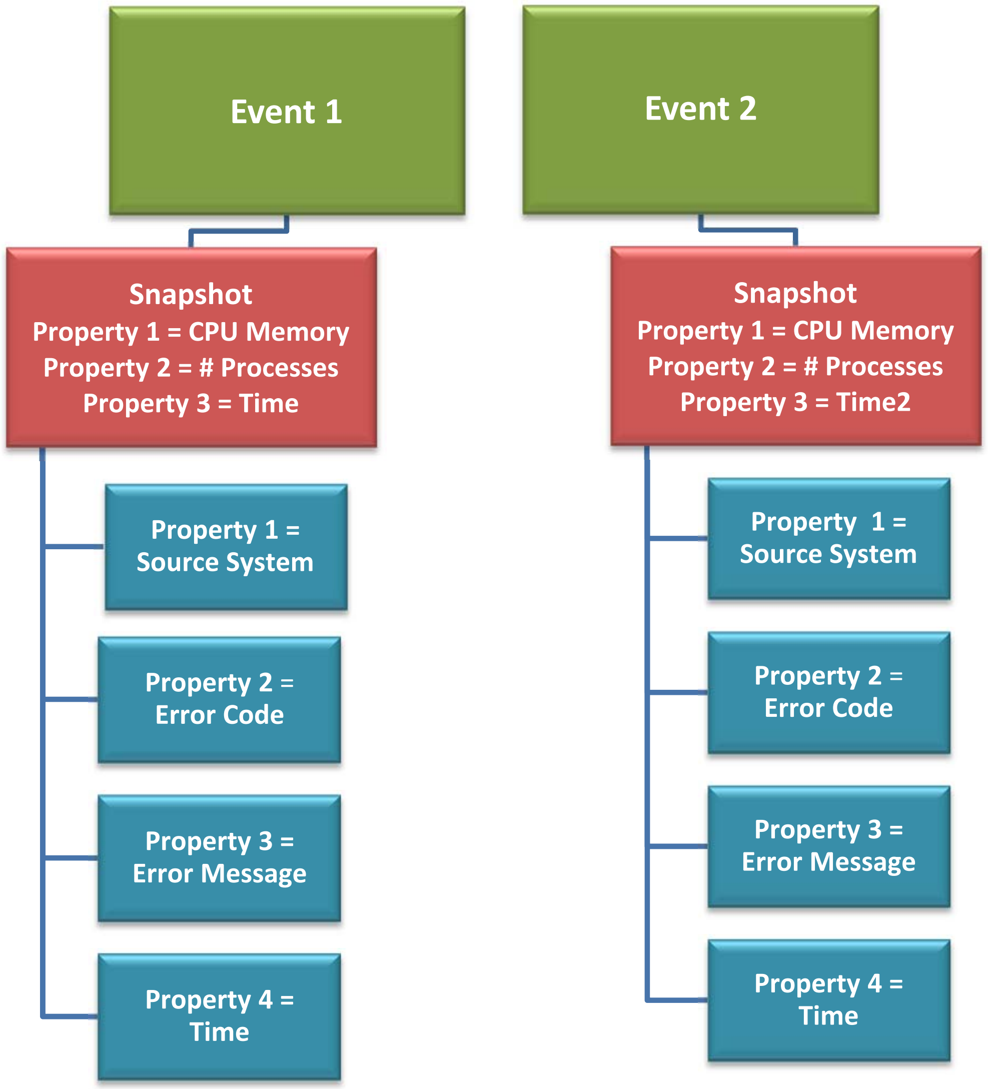
Reason 1 for Using Snapshots

Associate the snapshot to data in the events. In our movie example, we may take a snapshot of the weather every half hour. We can use these snapshots to see for example, does movie attendance spike when it is raining. In this example, the snapshot is a stand-alone snapshot. Associations between the snapshot and the movie events will be made during data analysis.

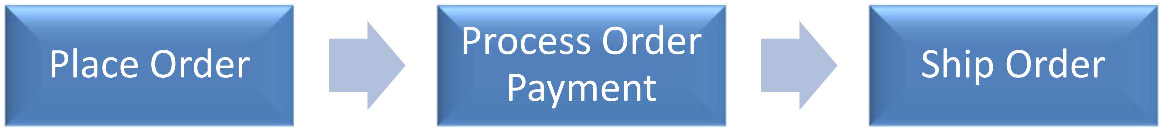

Activity Movies playing this week
Snapshot weather at 3 PM Property 1 =temperature Property 2=rain
Snapshot weather at 3 PM Property 1 =temperature Property 2=rain
Snapshot weather at 3 PM Property 1 =temperature Property 2=rain
Event 1 Casablanca at 1 PM
Event 2 Casablanca at 5 PM
Event 3 Play it Again Sam at 7 PM
Property 1 Movie name = Casablanca
Property 2 Movie price = $7
Property 3 Genre = drama
Property 4 Attendance = 235
Property 1 Movie name = Casablanca
Property 2 Movie price = $10
Property 3 Genre = drama
Property 1 Movie name = Play it Again Sam
Property 2 Movie price = $10
Property 3 Genre = comedy

Reason 2 for Using Snapshots

To help gather information as to “why/context” any event occurred. We will use a computer system example to explain this. Let’s say, we had a failure in our system. The code entered a “catch block.” In this catch block we wish to see what was happening with the operating system at the time of the error. For instance, we may wish to report memory utilization at the time the error condition was caught to see if running low on memory created the error. In this situation, we would “attach” the snapshot to the event. The event would contain properties pertaining to the error. The snapshots would contain properties pertaining to the operating system memory.



Correlators



Data can be in a "flow" as seen in the above diagram. Quite often, when data is flowing, it is flowing from disparate data sources. It flows via SEND and RECEIVE messages as is done with conventional messaging queues that are used in today's computer system. Correlators are a way of connecting this data that is flowing from different data sources.

The purpose of correlators is to allow the user to dive deeper into the meaning of their data. For instance, correlators can be used to diagnose the root cause of problems and gain insights into data that are not obvious.

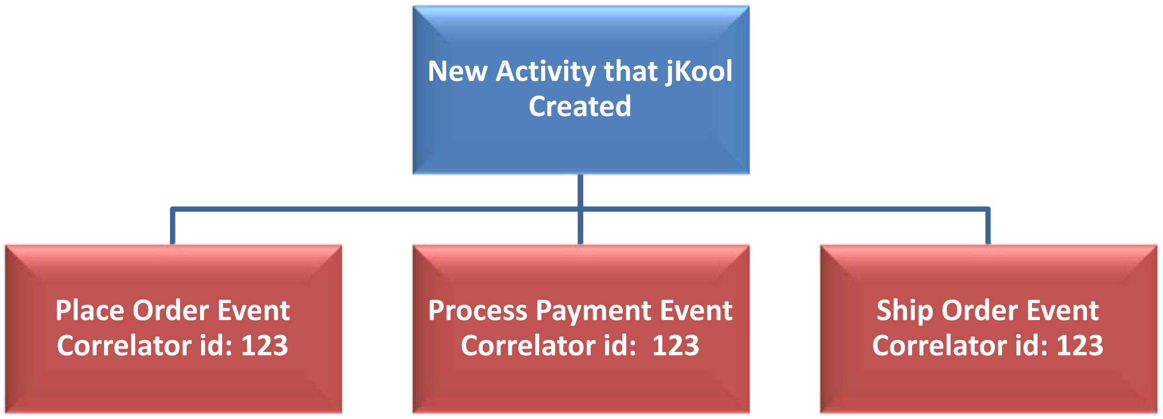
jKool does something that we call "stitching" – a process of creating relationships between seemingly independent events, activities based on correlators

Stitching

jKool uses correlators to measure performance, discover event flows and dependency models (e.g., business transaction). Behind the scenes it will "stitch the data". Stitching the data is how jKool makes associations in the data based on the correlators. This is best explained by example.

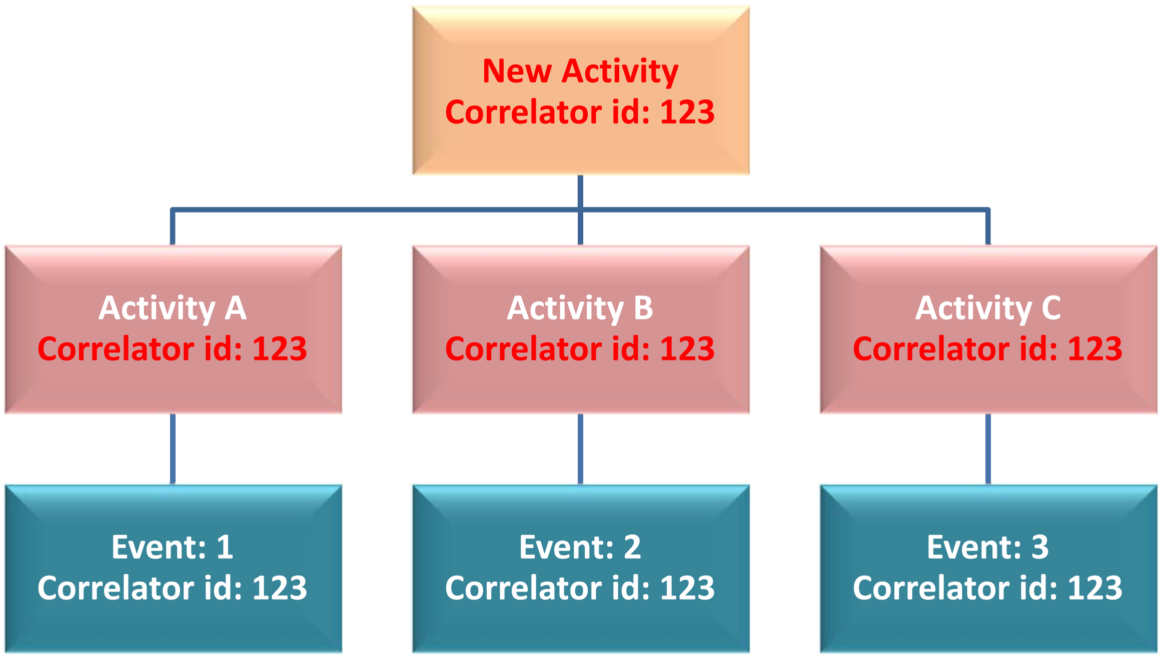
Example 1

Three events have the same correlator id and each event does not have a parent activity. jKool will create a brand new activity to associate all three events.


New Activity that jKool Created
Place Order Event Correlator id: 123
Process Payment Event Correlator id: 123
Ship Order Event Correlator id: 123

Example 2

Three events have the same correlator id, but each event also has its own activity. jKool will "not" create a new parent event activity in this situation. However, it will: 1) associate the correlator-id to the existing activities attached to the three events and then 2) associate those activities with a new "super-activity."



Red text represents fields and objects that stitching created.

There are more rules that jKool uses to stitch that are outside the scope of this document and will be published in a future document. What is important to understand however is "why jKool is doing this." jKool is doing this so that despite the fact that data is coming in from different data sources, when doing analytics, the data can be easily associated. This becomes especially important when one is working with and analyzing messaging systems. And as explained above, jKool is doing this to allow for more insight into your data.

Example 3

In the following example we have three data centers in the US that process our data. Data moves from one data center to the next via message queues.

The data center in New York receives orders *(Figure 1)*; then passes the orders on to a queue in California that processes the payments *(Figure 2)*; and then sends the orders on to a queue in North Carolina that processes the shipping *(Figure 3)*. Notice in this example a failure occurred due to an expired credit card, so a failure is registered in jKool. The important things to notice are:

- **Correlator IDs** – please notice how we use these correlator-ids to associate the events in the different data sources
- **Event types** of SEND and RECEIVE.
- **Times and durations** – We can use these timings to measure system performance and see where any processing bottlenecks are occurring.

Figure 1: Orders Received in New York Data Center

Two orders are received: **abc** and **xyz**. These orders are processed and then sent out to the next queue where payments will be processed. Please notice the type field that specifies **SEND** and **RECEIVE**.

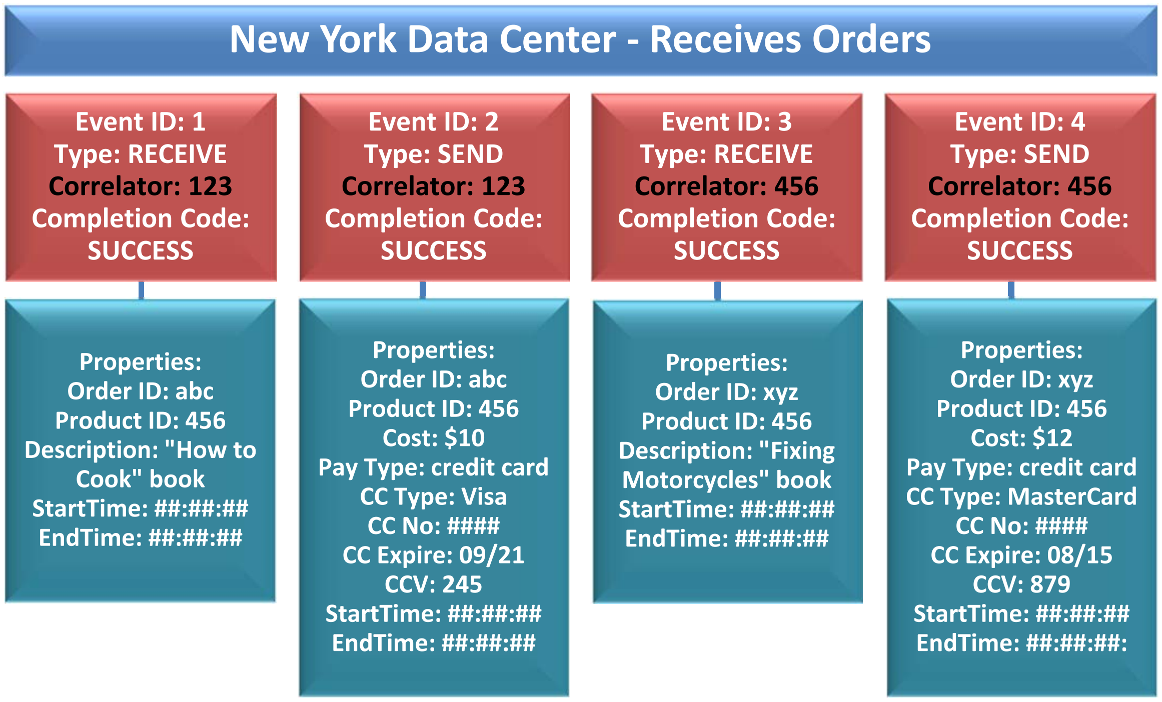

New York Data Center - Receives Orders
Event ID: 1
Type: RECEIVE
Correlator: 123
Completion Code: SUCCESS
Event ID: 2
Type: SEND
Correlator: 123
Completion Code: SUCCESS
Event ID: 3
Type: RECEIVE
Correlator: 456
Completion Code: SUCCESS
Event ID: 4
Type: SEND
Correlator: 456
Completion Code: SUCCESS
Properties: Order ID: abc Product ID: 456 Description: "How to Cook" book StartTime: ##:##:## EndTime: ##:##:##
Properties: Order ID: abc Product ID: 456 Cost: $10 Pay Type: credit card CC Type: Visa CC No: #### CC Expire: 09/21 CCV: 245 StartTime: ##:##:## EndTime: ##:##:##
Properties: Order ID: xyz Product ID: 456 Description: "Fixing Motorcycles" book StartTime: ##:##:## EndTime: ##:##:##
Properties: Order ID: xyz Product ID: 456 Cost: $12 Pay Type: credit card CC Type: MasterCard CC No: #### CC Expire: 08/15 CCV: 879 StartTime: ##:##:## EndTime: ##:##:##:

Figure 2: Payments Processed in California Data Center

Payment data for the two orders are now received on the second queue. Payments will be processed and then sent on to the third queue where shipping information will be processed. Please notice in this diagram that a failure occurred on the second order. Therefore, a failure event will be created.

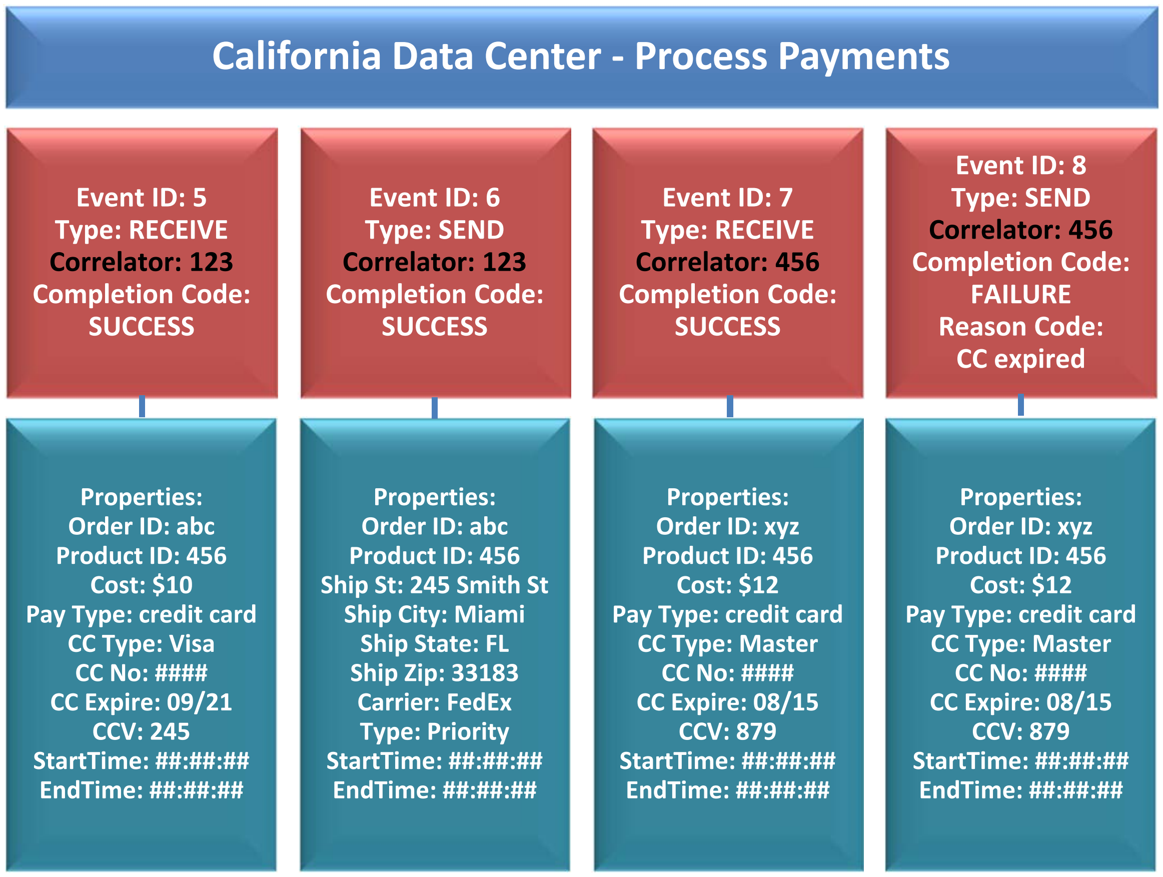
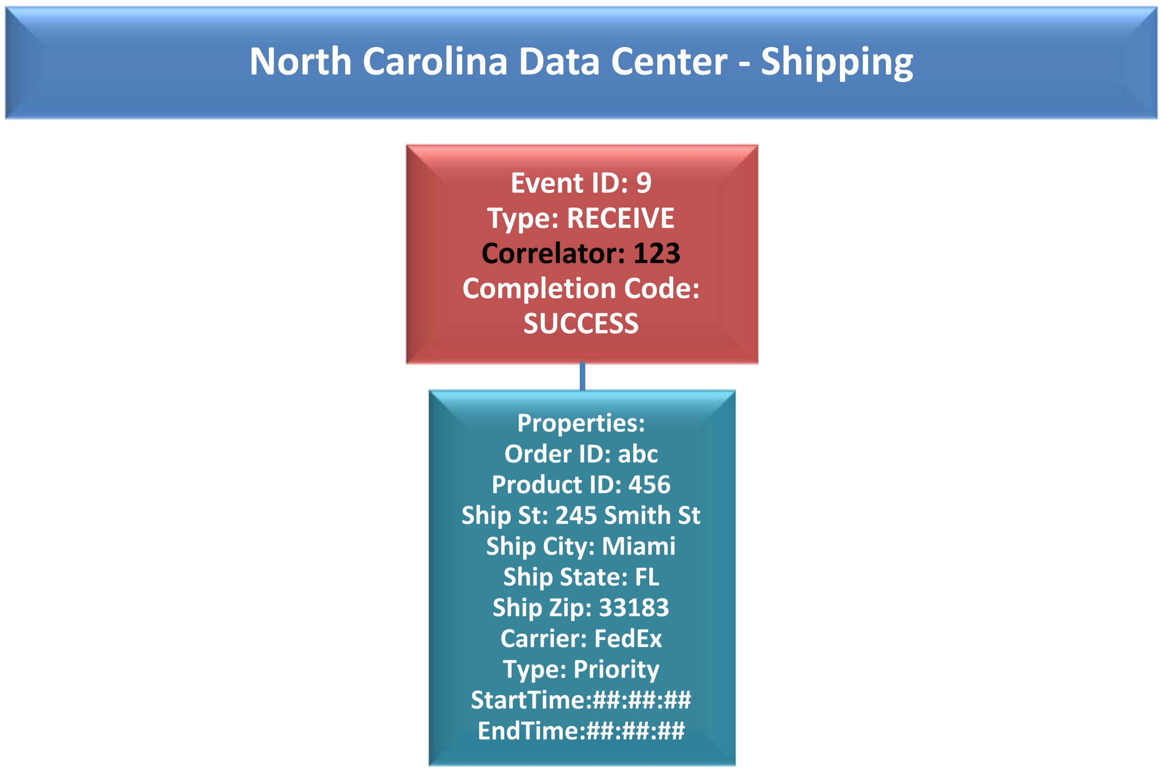


Figure 3: Orders are Shipped from North Carolina

Only one event was received on this queue because a failure occurred on the prior queue.



In this example, jKool can use the correlators to create the following activities behind the scenes:

- **Activity 1** – will contain events 1, 2, 5, 6, 9. It will have a start time of … , an end time of … and a duration of …
- **Activity 2** – will contain events 3, 4, 7, 8, 7. It will have a start time of … , an end time of … and a duration of …

The timings and durations can be used to diagnose system performance issues.

jKool Fields

Fields used to hold your custom data are in red. All other fields are predefined. Most predefined fields primarily deal with storing data that pertains to computer system performance and message queueing analytics. If you are not storing this type of data, these fields can be ignored. Please use your own custom properties instead.

The following table describes the jKool fields on each of the four data types (general description is given in beginning of this document):

- Activity
- Event
- Snapshot
- Property.

Please note that the correlator described in this document is a field on the "event."

All fields are indexed.

Field Name	Type	Format	Description
Event			
tracking-id	string		Identifies the data associated with the event. This identifier can be shared among events if they are operating on the same data.
corr-id	string		Correlator ID. Used to correlate events that may not be acting upon the same data, but should be associated (i.e.; data coming from different data sources but related to each other).
exception	string		Reason for a failed event.
resource	string		A resource the event acted upon; for example, a queue or file.
wait-time-used	integer	int32	If a blocking operation, how long the event was waiting for the data to appear.
source-fqn	string		Identifies very specifically where the event came from. Must be in a special format. ** Please see format below.*
source-url	string		The URL that sourced this event.
severity	string		Severity of the event. Defaults to SUCCESS. Valid values are: NONE, TRACE, DEBUG, INFO, SUCCESS, WARNING, ERROR, FAILURE, CRITICAL, FATAL, HALT.
pid	integer	int32	Process identifier.
tid	integer	int32	Thread identifier.
comp-code	string		Completion code. Valid values are: SUCCESS, WARNING, ERROR.

Field Name	Type	Format	Description
reason-code	integer	int32	Numeric reason for the event. Default is 0.
location	string		Location of the event. Can be a geo location, file number, etc.
operation	string		Name of the event.
user	string		User associated with the event.
time-usec	string	date-time	Time this event is reported.
start-time-usec	string	date-time	Time the event began.
end-time-usec	string	date-time	Time the event completed.
elapsed-time-usec	integer	int32	Elapsed time between when the event began and when it ended.
msg-text	string		Event message.
msg-size	integer	int32	Message size.
encoding	string		Message encoding.
charset	string		Message character set.
mime-type	string		Message mime type.
msg-age	integer	int32	How long message was waiting to be processed. Pertains to JMS or other types of messaging.
msg-tag	string		Message search label.
parent-tracking-id	string		Unique identifier of the parent activity.
snapshots	array		The list of snapshots associated with the event.
properties	array		List of properties associated with the event.
type	string		Type of event. Valid values are: EVENT, SEND, RECEIVE.
Snapshot			
name	string		Name of the snapshot.
type	string		Must be “SNAPSHOT.”
parent-id	string		The unique identifier of the snapshot’s parent event.
time-usec	string	date-time	Time this snapshot occurred.
category	string		The snapshot category.
properties	array		The list of properties associated with this snapshot.
Property			
name	string		Name of the custom field.
type	string		Data type of custom field.
value	string		Value of custom field.

Field Name	Type	Format	Description
Activity			
tracking-id	string		The unique identifier of this activity.
operation	string		The name of this activity.
start-time-used	string		Time the activity began.
end-time-used	string		Time the activity ended.
time-usec	string	date-time	Time this activity is reported. Use if begin time is equal to end time.
status	string		The status of this activity. Valid values are: UNKNOWN, BEGIN, END, EXCEPTION.
source-fqn	string		Identifies where the activity came from. Must be in a special format. ***** *Please see format below.*

***source-fqn** format is as follows:

APPL=<*application-name*>#SERVER=<*server-name*>#NETADDR=<*network-address*>
#DATACENTER=<*data-center*>#GEOADDR=<*geographic-address*>

jKool Streaming Ecosystem

jKool offers several mechanisms for streaming your data ranging from developer APIs, RESTful to specific collectors designed for specific data sources. Please note that the concepts explained in this document will be most helpful for those using the RESTful API. Data source specific collectors take advantage of the underlying data source interfaces, integrations and other elements to extract, parse and map data to jKool model. jKool streaming ecosystem is fast evolving: visit us @ GitHub.

Streaming for Developers:

Restful API

- Any type of data can be sent over to jKool using RESTful POST requests. Data is put into the request body in JSON format. A restful client that also contains sample code has been provided to make this easy.
- Please visit our GitHub repository at https://github.com/Nastel/jKoolRestClient to obtain this client.

Java Event Streaming Library (JESL)

- Stream events, metrics, and transactions directly from your Java application.
- Once you have registered with jKool, you may download this API at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Streaming Agent Framework (tnt4j-streams)

- Run-time platform for streaming logs, metrics, transactions
- Implements Stream, Parser Architecture with many available concrete implementations
- Supports File, HDFS, Apache Flume, Logstash, JMS, Kafka and many others
- Once you have registered with jKool, you may download this API at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Streaming for Specific Data Sources:

Stream Simulator

- Want to just try out streaming to jKool? Simulate the data you would see using a collector without deploying one by using our Simulator.
- Learn in advance the benefits you will receive from using jKool.
- Once you have registered with jKool, you may download Simulator at: https://www.jkoolcloud.com/product/technology/big-data-collectors.

Log4j (tnt4j-log4j12)

- Use this collector to consolidate log4j application logs on a single pane of glass. Once you have registered with jKool, you may download the Log4J Collector at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Syslog (tnt4j-syslogd)

- Consolidate Syslog log messages from multiple servers on a single pane of glass.
- Once you have registered with jKool, you may download the Syslog Collector at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Java JMX (tnt4-stream-jmx)

- Keep history of JMX metrics across multiple JVMs
- Understand JVM capacity, performance
- Compare how JVM performance changes with changes to your applications
- Once you have registered with jKool, you may download the Stream JMX Collector at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Java Servlet Filter Tracking (tnt4j-servlet-filter)

- Measure performance of your JEE Web applications
- Keep history of response times
- Track end-user behavior, location
- Once you have registered with jKool, you may download the TrackingFilter Collector at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Java Method & Transaction Tracing (tnt4j-streams-zorka)

- Measure performance of your java application to method level
- Track transactions from HTTP, LDAP, SQL, JMS and method calls
- User defined method intercepts without messing with bytecode
- Once you have registered with jKool, you may download the TrackingFilter Collector at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Java GC (tnt4j-stream-gc)

- Stream Garbage Collection data from a single or multiple JVMs
- Measure Garbage Collection frequency, duration
- Minor, Major, Full collections
- Measure Memory Pool Utilization
- Before and After GC collection
- Provides metrics required for GC troubleshooting and tuning
- Once you have registered with jKool, you may download the Stream GC Collector at https://www.jkoolcloud.com/product/technology/big-data-collectors.

Contact Us

Please contact us with any comments, questions, or concerns at support@jkoolcloud.com.