Solace JMS Integration with Spark Streaming 1.3

Document Version 1.0

November 2015

This document is an integration guide for using Solace JMS as a JMS provider for a Spark Streaming custom receiver.

Apache Spark is a fast and general-purpose cluster computing system. It provides an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLib for machine learning, GraphX for graph processing, and Spark Streaming for high-throughput, fault-tolerant stream processing of live data streams. The Spark Streaming custom receiver is a simple interface that allows third party applications to push data into Spark in an efficient manner.

The Solace message router supports persistent and non-persistent JMS messaging with high throughput and low, consistent latency. Thanks to very high capacity and built-in virtualization, each Solace message router can replace dozens of software-based JMS brokers in multi-tenant deployments. Since JMS is a standard API, client applications connect to Solace like any other JMS broker so companies whose applications are struggling with performance or reliability issues can easily overcome them by upgrading to Solace's hardware.

Table of Contents

1 Overview

This document demonstrates how to integrate Solace Java Message Service (JMS) with the Spark Streaming custom receiver for consumption of JMS messages. The goal of this document is to outline best practices for this integration to enable efficient use of both the Spark Streaming and Solace JMS.

The target audience of this document is developers using the Hadoopv2 with knowledge of both the Spark and JMS in general. As such this document focuses on the technical steps required to achieve the integration. For detailed background on either Solace JMS or Spark refer to the referenced documents below.

This document is divided into the following sections to cover the Solace JMS integration with Spark Streaming:

- Integrating with Spark Streaming
- Performance Considerations
- Working with Solace High Availability
- Debugging Tips
- Advanced Topics including:
    - Using SSL Communication
    - Working with Solace Disaster Recovery

1.1 Related Documentation

These documents contain information related to the feature defined in this document

Document ID	Document Title	Document Source
[Solace-Portal]	Solace Developer Portal	http://dev.solacesystems.com
[Solace-JMS-REF]	Solace JMS Messaging API Developer Guide	http://dev.solacesystems.com/docs/solace-jms-api-developer-guide
[Solace-JMS-API]	Solace JMS API Online Reference Documentation	http://dev.solacesystems.com/docs/solace-jms-api-online-reference
[Solace-FG]	Solace Messaging Platform – Feature Guide	http://dev.solacesystems.com/docs/messaging-platform-feature-guide
[Solace-FP]	Solace Messaging Platform – Feature Provisioning	http://dev.solacesystems.com/docs/messaging-platform-feature-provisioning
[Solace-CLI]	Solace Message Router Command Line Interface Reference	http://dev.solacesystems.com/docs/cli-reference
[Spark-REF]	Spark Streaming Custom Receivers Documentation	http://spark.apache.org/docs/latest/streaming-custom-receivers.html
[Spark-API]	Spark Receiver Class Documentation	http://spark.apache.org/docs/latest/api/java/org/apache/spark/streaming/receiver/Receiver.html

Table 1 - Related Documents

2 Why Solace

Solace technology efficiently moves information between all kinds of applications, users and devices, anywhere in the world, over all kinds of networks. Solace makes its state-of-the-art data movement capabilities available via hardware and software "message routers" that can meet the needs of any application or deployment environment. Solace's unique solution offers unmatched capacity, performance, robustness and TCO so our customers can focus on seizing business opportunities instead of building and maintaining complex data distribution infrastructure.

Superior Performance

Solace's hardware and software messaging middleware products can cost-effectively meet the performance needs of any application, with feature parity and interoperability that lets companies start small and scale to support higher volume or more demanding requirements over time, and purpose-built appliances that offer 50-100x higher performance than any other technology for customers or applications that require extremely high capacity or low latency.

Robustness

Solace offers high availability (HA) and disaster recovery (DR) without the need for 3rd party products, and fast failover times no other solution can match. Distributing data via dedicated TCP connections ensures an orderly, well-behaved system under load, and patented techniques ensure that the performance of publishers and high-speed consumers is never impacted by slow consumers.

Simple Architecture

Modern enterprises run applications that demand many kinds of data movement such as persistent messaging, web streaming, WAN distribution and cloud-based communications. By supporting all kinds of data movement with a unified platform that can be deployed as a small-footprint software broker or high-capacity rack-mounted appliance, Solace lets architects design an end-to-end infrastructure that's easy to build applications for, integrate with existing technologies, secure and scale.

Simple Operations

Solace's solution features a shared administration framework for all kinds of data movement, deployment models and network environments so it's easy for IT staff to deploy, monitor, manage and upgrade their Solace-based messaging environment.

Cost Savings

Solace reduces expenses with high-capacity hardware, flexible software, and the ability to deploy the right solution for each problem. Solace's support for many kinds of messaging lets you replace multiple messaging products with just one, built-in HA, DR, WAN and Web functionality eliminate the need for third-party products.

3 Integrating with Spark Streaming

The general Spark Streaming support for custom receivers is documented here [Spark-REF]. The configuration outlined in this document makes use of a custom receiver to achieve the desired integration with Solace.

This integration guide demonstrates how to configure a Spark Streaming application to receive JMS messages using a custom receiver. Accomplishing this requires completion of the following steps.

- Step 1 – Obtain access to Solace message router and JMS API, see [Solace-Portal]
- Step 2 - Configuration of the Solace Message Router.
- Step 3 – Coding a JMS custom receiver.
- Step 4 – Deploying JMS receiver

3.1 Description of Resources Required

This integration guide will demonstrate creation of Solace JMS custom receiver and configuring the receiver to receive messages. This section outlines the resources that are required/created and used in the subsequent sections.

3.1.1 Solace Resources

The following Solace Message Router resources are required.

Resource	Value	Description
Solace Message Router IP:Port	__IP:Port__	The IP address and port of the Solace Message Router message backbone. This is the address clients use when connecting to the Solace Message Router to send and receive message. This document uses a value of __IP:PORT__.
Message VPN	Solace_Spark_VPN	A Message VPN, or virtual message broker, to scope the integration on the Solace Message Router.
Client Username	spark_user	The client username.
Client Password	spark _password	Optional client password.
Solace Queue	Q/receiver	Solace destination of persistent messages consumed
JNDI Connection Factory	JNDI/CF/spark	The JNDI Connection factory for controlling Solace JMS connection properties
JNDI Queue Name	JNDI/Q/receiver	The JNDI name of the queue used in the samples

Table 2 – Solace Resources

3.1.2 Spark Resources

The following Spark resources are required for code integration:

Resource	Value
org.apache.spark.storage.StorageLevel	
org.apache.spark.streaming.receiver.Receiver	

Table 3 – Spark Resources

3.2 Step 1 – Obtain access to Solace message router and JMS API

The Solace messaging router can be obtained one of 2 ways.

1. If you are in an organization that is an existing Solace customer, it is likely your organization already has Solace Message Routers and corporate policies about their use. You will have to contact your middleware operational team in regards to access to a Solace Message Router.
2. If you are new to Solace or your company does not have development message routers, you can obtain a trail Solace Virtual Message Router from the [Solace-Portal] in the Downloads-> Products -> Virtual Message Router section.

The following Solace libraries are required. They can be obtained on [Solace-Portal] in the Downloads-> Enterprise Messaging APIs-> JMS section.

Resource	Value	Description
Solace Common	sol-common-7.1.x.x.jar	Solace common utilities library
Solace JCSMP	sol-jcsmp-7.1.x.x.jar	Underlying Solace wireline support libraries
Solace JMS	sol-jms-7.1.x.x.jar	Solace JMS 1.1 compliant libraries
Apache Commons language	commons-lang-2.6.jar	Common language libraries
Apache Commons logging	commons-logging-1.1.3.jar	Common logging libraries
Apache Geronimo	geronimo-jms_1.1_spec-1.1.1.jar	Apache Geronimo is an open source server runtime that integrates the best open source projects to create Java/OSGi server runtimes that meet the needs of enterprise developers and system administrators. Our most popular distribution is a fully certified Java EE 6 application server runtime.

3.3 Step 2 – Configuring the Solace Message Router

The Solace appliance needs to be configured with the following configuration objects at a minimum to enable JMS to send and receive messages within the Spark application.

- A Message VPN, or virtual message broker, to scope the integration on the Solace appliance.
- Client connectivity configurations like usernames and profiles
- Guaranteed messaging endpoints for receiving messages.
- Appropriate JNDI mappings enabling JMS clients to connect to the Solace appliance configuration.

For reference, the CLI commands in the following sections are from SolOS version 6.2 but will generally be forward compatible. For more details related to Solace appliance CLI see [Solace-CLI]. Wherever possible, default values will be used to minimize the required configuration. The CLI commands listed also assume that the CLI user has a Global Access Level set to Admin. For details on CLI access levels please see [Solace-FG] section “User Authentication and Authorization”.

Also note that this configuration can also be easily performed using SolAdmin, Solace’s GUI management tool. This is in fact the recommended approach for configuring a Solace appliance. This document uses CLI as the reference to remain concise.

3.3.1 Creating a Message VPN

This section outlines how to create a message-VPN called “Solace_Spark_VPN” on the Solace appliance with authentication disabled and 2GB of message spool quota for Guaranteed Messaging. This message-VPN name is required in the Spark configuration when connecting to the Solace messaging appliance. In practice appropriate values for authentication, message spool and other message-VPN properties should be chosen depending on the end application’s use case.

```
(config)# create message-vpn Solace_Spark_VPN
(config-msg-vpn)# authentication
(config-msg-vpn-auth)# user-class client
(config-msg-vpn-auth-user-class)# basic auth-type none
(config-msg-vpn-auth-user-class)# exit
(config-msg-vpn-auth)# exit
(config-msg-vpn)# no shutdown
(config-msg-vpn)# exit
(config)#
(config)# message-spool message-vpn Solace_Spark_VPN
(config-message-spool)# max-spool-usage 2000
(config-message-spool)# exit
(config)#
```

3.3.2 Configuring Client Usernames & Profiles

This section outlines how to update the default client-profile and how to create a client username for connecting to the Solace appliance. For the client-profile, it is important to enable guaranteed messaging for JMS messaging and transacted sessions if using transactions.

The chosen client username of “spark_user” will be required by the Spark application when connecting to the Solace appliance.

```
(config)# client-profile default message-vpn Solace_Spark_VPN
(config-client-profile)# message-spool allow-guaranteed-message-receive
(config-client-profile)# message-spool allow-guaranteed-message-send
(config-client-profile)# message-spool allow-transacted-sessions
(config-client-profile)# exit
(config)#
(config)# create client-username spark_user message-vpn Solace_Spark_VPN
(config-client-username)# acl-profile default
(config-client-username)# client-profile default
(config-client-username)# no shutdown
(config-client-username)# exit
(config)#
```

3.3.3 Setting up Guaranteed Messaging Endpoints

This integration guide shows receiving messages within the Spark application from a single JMS Queue. For illustration purposes, this queue is chosen to be an exclusive queue with a message spool quota of 2GB matching quota associated with the message VPN. The queue name chosen is “Q/requests”.

```
(config)# message-spool message-vpn Solace_Spark_VPN
(config-message-spool)# create queue Q/receive
(config-message-spool-queue)# access-type exclusive
(config-message-spool-queue)# max-spool-usage 2000
(config-message-spool-queue)# permission all delete
(config-message-spool-queue)# no shutdown
(config-message-spool-queue)# exit
(config-message-spool)# exit
(config)#
```

3.3.4 Setting up Solace JNDI References

To enable the JMS clients to connect and look up the Queue destination required by Spark, there are two JNDI objects required on the Solace appliance:

- A connection factory: JNDI/CF/spark
- A queue destination: JNDI/Q/receive

They are configured as follows:

```
(config)# jndi message-vpn Solace_Spark_VPN
(config-jndi)# create connection-factory JNDI/CF/spark
(config-jndi-connection-factory)# property-list messaging-properties
(config-jndi-connection-factory-pl)# property default-delivery-mode persistent
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property direct-transport false
(config-jndi-connection-factory-pl)# property "reconnect-retry-wait" "3000"
(config-jndi-connection-factory-pl)# property "reconnect-retries" "20"
(config-jndi-connection-factory-pl)# property "connect-retries-per-host" "5"
(config-jndi-connection-factory-pl)# property "connect-retries" "1"
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# exit
(config-jndi)#
(config-jndi)# create queue JNDI/Q/receive
(config-jndi-queue)# property physical-name Q/receive
(config-jndi-queue)# exit
(config-jndi)#
(config-jndi)# no shutdown
(config-jndi)# exit
(config)#
```

3.4 Step 3 – Coding a JMS custom receiver.

From [Spark-REF] there is details on how to build a custom receiver and a template. In this section of the document will use this template and build a JMSReciever.

The JMSReceiver extends the org.apache.spark.streaming.receiver.Receiver and implements the javax.jms.MessageListener. This will result in the following methods created:

- `JMSReceiver` constructor – Synchronously called once as the Receiver is initially created.
- `org.apache.spark.streaming.receiver.Receiver.onStart()` – Asynchronously called once as the Receiver is started.
- `org.apache.spark.streaming.receiver.Receiver.onStop()` – Asynchronously called once as the Receiver is stopped
- `javax.jms.MessageListener.onMessage()` – Asynchronously called on every message received from Solace

```
public class JMSReceiver extends Receiver<String> implements MessageListener{
       private static final long serialVersionUID = 1L;
       private static final String SOLJMS_INITIAL_CONTEXT_FACTORY =
             "com.solacesystems.jndi.SolJNDIInitialContextFactory";
      public JMSReceiver( ) throws NamingException {
        super(StorageLevel.MEMORY_ONLY_SER_2());
        }

       @Override
       public void onStart() {
        // TODO Auto-generated from spark.streaming.receiver
       }

       @Override
       public void onStop() {
              // TODO Auto-generated from spark.streaming.receiver
       }

       @Override
       public void onMessage(Message arg0) {
              // TODO Auto-generated from javax.jms.MessageListener
       }
}
```

In the constructor we need to collect information to information needed to connect to Solace and build the JMS environment.

```
        String jndiQueue_s;
        String connectionFactory_s;

        public JMSReceiver(String brokerURL, String vpn,
                    String username, String password,
                    String jndiQueue, String connectionFactory) throws NamingException {
        super(StorageLevel.MEMORY_ONLY_SER_2());
        Hashtable<String, String> env = new Hashtable<String, String>();
        env.put(InitialContext.INITIAL_CONTEXT_FACTORY, SOLJMS_INITIAL_CONTEXT_FACTORY);
        env.put(InitialContext.PROVIDER_URL, brokerURL);
        env.put(Context.SECURITY_PRINCIPAL, username);
        env.put(Context.SECURITY_CREDENTIALS, password);
        env.put(SupportedProperty.SOLACE_JMS_VPN, vpn);

        jndiQueue_s = jndiQueue;
        connectionFactory_s = connectionFactory;
        }
```

- Next in the onStart() method we need to look up the JMS connection factory and queue then connect to receive messages.

```
        @Override
        public void onStart() {
        InitialContext initialContext = null;
        try {
                ConnectionFactory factory = (ConnectionFactory)
                            initialContext.lookup(connectionFactory_s);
                connection_s = factory.createConnection();
                Destination queue = (Destination) initialContext.lookup(jndiQueue_s);

                Session session = connection_s.createSession(false,
                                  Session.CLIENT_ACKNOWLEDGE);
                MessageConsumer consumer = session.createConsumer(queue);
                consumer.setMessageListener(this);
                connection_s.start();

                } catch (NamingException e) {
                        e.printStackTrace();
                } catch (JMSException e) {
                        e.printStackTrace();
                }
        }
```

Finally, when receiving messages from Solace they need to be stored into Spark.

```
        @Override
        public void onMessage(Message message) {
                try {
                        store(message.toString());
                        message.acknowledge();
                } catch (JMSException e) {
                        e.printStackTrace();
                }
        }
```

3.5 Step 4 – Deploying JMS Receiver

The complete JMS Receiver including imbedded example is attached in Appendix A. In the complete example the package is test to:

```
package org.apache.spark.examples.streaming;
```

This is set as such to easily allow execution within the Spark example directory structure and may need to be changes to best fit your operational environment.

To invoke the JMS receiver:

```
SparkConf sparkConf = new SparkConf().setAppName("JMSReceiver");
JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, new Duration(1000));
JavaReceiverInputDStream<String> lines = ssc.receiverStream(
                    new JMSReceiver(broker, vpn, username, passwd, queue, cf));
```

4 Performance Considerations

In the provided example above persistent messaging was used on the appliance and the Spark Streaming client connected to a queue. This design pattern provides the highest level of reliability as each message is persisted on the Solace message router and will not be lost in case of a client failure. This message pattern consumes the most resources on the Solace Message Router and is not the most performant.

If the client does not want to receive messages that where missed while it was off line, does not want to receive older messages if it is unable to keep up to the published message flow, or wants the highest throughput with lowest latency; then direct message is the correct pattern.

To achieve direct messaging, configure the connection-factory to enable this feature.

```
(config)# jndi message-vpn Solace_Spark_VPN
(config-jndi)# connection-factory JNDI/CF/spark
 (config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property direct-transport true
```

```
(config)# jndi message-vpn Solace_Spark_VPN
(config-jndi)# create topic JNDI/T/receive
(config-jndi-queue)# property physical-name T/receive
(config-jndi-queue)# exit
```

5 Working with Solace High Availability (HA)

The [Solace-JMS-REF] section “Establishing Connection and Creating Sessions” provides details on how to enable the Solace JMS connection to automatically reconnect to the standby appliance in the case of a HA failover of a Solace appliance. By default Solace JMS connections will reconnect to the standby appliance in the case of an HA failover.

In general the Solace documentation contains the following note regarding reconnection:

> Note: When using HA redundant appliances, a fail-over from one appliance to its mate will typically occur in under 30 seconds, however, applications should attempt to reconnect for at least five minutes.

In section 3.3.4 Setting up Solace JNDI References, the Solace CLI commands correctly configured the required JNDI properties to reasonable values. These commands are repeated here for completeness.

```
config)# jndi message-vpn Solace_Spark_VPN
(config-jndi)# connection-factory JNDI/CF/park
(config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property "reconnect-retry-wait" "3000"
(config-jndi-connection-factory-pl)# property "reconnect-retries" "20"
(config-jndi-connection-factory-pl)# property "connect-retries-per-host" "5"
(config-jndi-connection-factory-pl)# property "connect-retries" "1"
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# exit
(config-jndi)# exit
(config)#
```

Finally ensure that the JNDI Destination you are using points to a Topic not a Queue:

```
(config)# jndi message-vpn Solace_Spark_VPN
(config-jndi)# create topic JNDI/T/recieve
(config-jndi-queue)# property physical-name Topic/Recieve
(config-jndi-queue)# exit
```

6 Debugging Tips for Solace JMS API Integration

The key component for debugging integration issues with the Solace JMS API is the API logging that can be enabled. How to enable logging in the Solace API is described below.

6.1 How to enable Solace JMS API logging

Spark was written using Jakarta Commons Logging API (JCL), Solace JMS API also makes use of the Jakarta Commons Logging API (JCL), configuring the Solace JMS API logging is very similar to configuring any other Spark application. The following example shows how to enable debug logging in the Solace JMS API using log4j.

One note to consider is that since the Solace JMS API has a dependency on the Solace Java API (JCSMP) both of the following logging components should be enabled and tuned when debugging to get full information. For example to set both to debug level:

```
log4j.category.com.solacesystems.jms=DEBUG
log4j.category.com.solacesystems.jcsmp=DEBUG
```

By default info logs will be written to the consol. This section will focus on using log4j as the logging library and tuning Solace JMS API logs using the log4j properties. Therefore in order to enable Solace JMS API logging, a user must do two things:

- Put Log4j on the classpath.
- Create a log4j.properties configuration file in the root folder of the classpath

Below is an example Log4j properties file that will enable debug logging within the Solace JMS API.

```
log4j.rootCategory=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n
log4j.category.com.solacesystems.jms=DEBUG
log4j.category.com.solacesystems.jcsmp=DEBUG
```

With this you can get output in a format similar to the following which can help in understanding what is happening within the Solace JMS API.

```
14:35:01,171 DEBUG main client.ClientRequestResponse:75 - Starting request timer (SMP-
EstablishP2pSub) (10000 ms)
14:35:01,171 DEBUG Context_2_ReactorThread client.ClientRequestResponse:83 - Stopping
request timer (SMP-EstablishP2pSub)
14:35:01,173  INFO main jms.SolConnection:151 - Connection created.
14:35:01,173  INFO main connection.CachingConnectionFactory:298 - Established shared JMS
Connection: com.solacesystems.jms.SolConnection@ca3f2d
14:35:01,180  INFO main jms.SolConnection:327 - Entering start()
14:35:01,180  INFO main jms.SolConnection:338 - Leaving start() : Connection started.
14:35:01,180  INFO jmsContainer-1 jms.SolConnection:252 - Entering createSession()
```

7 Advanced Topics

7.1 Authentication

JMS Client authentication is handled by the Solace appliance. The Solace appliance supports a variety of authentications schemes as described in [Solace-FG] in the Section “Client Authentication and Authorization.

In this section we will show how to configure the Solace Message Router to pass the authentication username/password through to an LDAP,(Active-Directory) server to incorporate with enterprise level authentication mechanisms. TLS certificates and Kerberos are also possible.

- First an LDAP profile needs to be created, this includes:
- Admin Username and Password to do LDAP lookups
- Part of the LDAP structure to check for users
- Location of LDAP server(s)
- Search filter, how to compare Client Username to LDAP Structure.

```
(config)# create authentication ldap-profile ActiveDirectoryIntegration
(config/authentication/ldap-profile)# admin dn DomainAdmin password xxxxxx
(config/authentication/ldap-profile)# search base-dn dc=lab,dc=solace,dc=com
(config/authentication/ldap-profile)# ldap-server ldap://192.168.1.56 index 1
(config/authentication/ldap-profile)# search filter "(sAMAccountName = $CLIENT_USERNAME)"
(config/authentication/ldap-profile)# no shut
(config/authentication/ldap-profile)# exit
```

Finally the LDAP profile will need to be enabled for the message VPN. Note that there is no code change from the Application/API. As the authentication is pass-through from the appliance to the LDAP server.

```
(config)# message-vpn Solace_Spark_VPN
(config/message-vpn)# authentication user-class client
(...message-vpn/authentication/user-class)# basic
(...e-vpn/authentication/user-class/basic)# auth-type ldap ActiveDirectoryIntegration
(...e-vpn/authentication/user-class/basic)# exit
```

7.1.1 Configuring the Solace Message Router

To enable secure connections to the Solace appliance, the following configuration must be updated on the Solace appliance.

- Server Certificate
- TLS/SSL Service Listen Port
- Enable TLS/SSL over SMF in the Message VPN

The following sections outline how to configure these items.

7.1.1.1 Configure the Server Certificate

Before, starting, here is some background detail on the server certificate required by the Solace Message Router. This is from the [Solace-FP] section “Setting a Server Certificate”

> To enable the exchange of information through TLS/SSL-encrypted SMF service, you must set the TLS/SSL server certificate file that the Solace Message Router is to use. This server certificate is presented to a client during the TLS/SSL handshakes. A server certificate used by an appliance must be an x509v3 certificate and it must include a private key. The server certificate and key use an RSA algorithm for private key generation, encryption and decryption, and they both must be encoded with a Privacy Enhanced Mail (PEM) format.
>
> The single server certificate file set for the appliance can have a maximum chain depth of three (that is, the single certificate file can contain up to three certificates in a chain that can be used for the certificate verification).

To configure the server certificate, first copy the server certificate to the Solace Message Router. For the purposes of this example, assume the server certificate file is named “mycert.pem”.

```
# copy sftp://[<username>@]<ip-addr>/<remote-pathname>/mycert.pem /certs
<username>@<ip-addr>'s password:
#
```

Then set the server certificate for the Solace Message Router.

```
(config)# ssl server-certificate mycert.pem
(config)#
```

7.1.1.2 Configure TLS/SSL Service Listen Port

By default, the Solace Message Router accepts secure messaging client connections on port 55443. If this port is acceptable then no further configuration is required and this section can be skipped. If a non-default port is desired, then follow the steps below. Note this configuration change will disrupt service to all clients of the Solace Message Router and should therefore be performed during a maintenance window when this client disconnection is acceptable. This example assumes that the new port should be 55403.

```
(config)# service smf
(config-service-smf)# shutdown
All SMF and WEB clients will be disconnected.
Do you want to continue (y/n)? y
(config-service-smf)# listen-port 55403 ssl
(config-service-smf)# no shutdown
(config-service-smf)# exit
(config)#
```

7.1.1.3 Enable TLS/SSL within the Message VPN

By default within Solace message VPNs both the plain-text and SSL services are enabled. If the Message VPN defaults remain unchanged, then this section can be skipped. However, if within the current application VPN, this service has been disabled, then for secure communication to succeed it should be enabled. The steps below show how to enable SSL within the SMF service to allow secure client connections from the Spring Framework.

```
(config)# message-vpn Solace_Spring_VPN
(config-msg-vpn)# service smf
(config-msg-vpn-service-smf)# ssl
(config-msg-vpn-service-ssl)# no shutdown
(config-msg-vpn-service-ssl)# exit
(config-msg-vpn-service-smf)# exit
(config-msg-vpn-service)# exit
(config-msg-vpn)# exit
(config)#
```

7.1.2 Configuring Spark

The configuration is done via a spark.confugration file and jndi.properties file.

- Updating the provider URL to specify the protocol as secure (smfs)
- Adding the required parameters for the secure connection

7.1.2.1 Updating the provider URL

In order to signal to the Solace JMS API that the connection should be a secure connection, the protocol must be updated in the URI scheme. The Solace JMS API has a URI format as follows:

```
<URI Scheme>://[username]:[password]@<IP address>[:port]
```

To connect to :

```
smf://spark_user@__IP:Port__
```

This specified a URI scheme of “smf” which is the plaint-text method of communicating with the Solace Message Router. This should be updated to “smfs” to switch to secure communication giving you the following configuration:

```
smfs://spark_user@__IP:Port__
```

7.1.2.2 Adding SSL Related Configuration

Additionally, the Solace JMS API must be able to validate the server certificate of the Solace Message Router in order to establish a secure connection. To do this, the following trust store parameters need to be provided in the jndi.properties file

First the Solace JMS API must be given a location of a trust store file so that it can verify the credentials of the Solace Message Router server certificate during connection establishment. This parameter takes a URL or Path to the trust store file.

```
env.put(SupportedProperty.Solace_JMS_SSL_TrustStore, ___Path_or_URL___)
```

It is also required to provide a trust store password. This password allows the Solace JMS API to validate the integrity of the contents of the trust store. This is done through the following parameter.

```
env.put(SupportedProperty.Solace_JMS_SSL_TrustStorePassword, ___Password___)
```

There are multiple formats for the trust store file. By default Solace JMS assumes a format of Java Key Store (JKS). So if the trust store file follows the JKS format then this parameter may be omitted. Solace JMS supports two formats for the trust store: "jks" for Java Key Store or "pkcs12". Setting the trust store format is done through the following parameter.

```
env.put(SupportedProperty.Solace_JMS_SSL_TrustStoreFormat, jks)
```

And finally, the authentication scheme must be selected. Solace JMS supports the following authentication schemes for secure connections:

- AUTHENTICATION_SCHEME_BASIC
- AUTHENTICATION_SCHEME_CLIENT_CERTIFICATE

This integration example will use basic authentication. So the required parameter is as follows:

```
env.put(SupportedProperty.Solace_JMS_Authentication_Scheme,AUTHENTICATION_SCHEME_BASIC)
```

7.2 Working with the Solace Disaster Recovery Solution

The [Solace- FG] section "Data Center Replication" contains a sub-section on "Application Implementation" which details items that need to be considered when working with Solace's Data Center Replication feature. This integration guide will show how the following items required to have a Spark application successfully connect to a backup data center using the Solace Data Center Replication feature.

- Configuring a Host List within the Spark Framework
- Configuring JMS Reconnection Properties within Solace JNDI
- Disaster Recovery Behavior Notes

7.2.1 Configuring a Host List within the Spring Framework

As described in [Solace-FG], the host list provides the address of the backup data center.

When connecting, the Solace JMS connection will first try the active site and if it is unable to successfully connect to the active site, then it will try the standby site. This is discussed in much more detail in the referenced Solace documentation. To configure a host list simple join a list of Solace Message Router URLs with ",", eg:

```
smf://messageRouter1,smf://messageRoutr2,smf://messageRouter3
```

7.2.2 Configuring reasonable JMS Reconnection Properties within Solace JNDI

In order to enable applications to successfully reconnect to the standby site in the event of a data center failure, it is required that the Solace JMS connection be configured to attempt connection reconnection for a sufficiently long time to enable the manual switch-over to occur. This time is application specific depending on individual disaster recovery procedures and can range from minutes to hours depending on the application. In general it is best to tune the reconnection by changing the "reconnect retries" parameter within the Solace JNDI to a value large enough to cover the maximum time to detect and execute a disaster recovery switch over. If this time is unknown, it is also possible to use a value of "-1" to force the Solace JMS API to reconnect indefinitely.

The reconnect retries is tuned in the Solace Message Router CLI as follows:

```
config)# jndi message-vpn Solace_Spark_VPN
(config-jndi)# connection-factory JNDI/CF/spark
(config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property "reconnect-retries" "-1"
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# exit
(config-jndi)# exit
(config)#
```

7.2.3 Disaster Recovery Behavior Notes

When a disaster recovery switch-over occurs, the Solace JMS API must establish a new connection to the Solace Message Routers in the standby data center. Because this is a new connection there are some special considerations worth noting. The [Solace-FG] contains the following notes:

Java and JMS APIs

For client applications using the Java or JMS APIs, any sessions on which the clients have published Guaranteed messages will be destroyed after the switch-over. To indicate the disconnect and loss of publisher flow:

- The Java API will generate an exception from the `JCSMPStreamingPublishCorrelatingEventHandler.handleErrorEx()` that contains a subcode of `JCSMPErrorResponseSubcodeEx.UNKNOWN_FLOW_NAME`.
- The JMS API will generate an exception from the `javax.jms.ExceptionListener` that contains the error code `SolJMSErrorCodes.EC_UNKNOWN_FLOW_NAME_ERROR`.

Upon receiving these exceptions the client application will know to create a new session.

After a new session is established, the client application can republish any Guaranteed messages that had been sent but not acked on the previous session, as these message might not have been persisted and replicated.

To avoid out-of-order messages, the application must maintain an unacked list that is added to before message publish and removed from on receiving an ack from the appliance. If a connection is re-established to a different host in the hostlist, the unacked list must be resent before any new messages are published.

Note: When sending persistent messages using the JMS API, a producer's send message will not return until an acknowledgment is received from the appliance. Once received, it is safe to remove messages from the unacked list.

Alternatively, if the application has a way of determining the last replicated message—perhaps by reading from a last value queue—then the application can use that to determine where to start publishing.

8 Appendix - Configuration and Java Source Reference

8.1 JMSReciever.java

```
package com.solacesystems.jms.samples;
import org.apache.log4j.Logger;
import org.apache.spark.storage.StorageLevel;
import org.apache.spark.streaming.receiver.Receiver;

import com.solacesystems.jms.SupportedProperty;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;

import java.util.Hashtable;

public class JMSReceiver extends Receiver<String> implements MessageListener
{
	private static final Logger log = Logger.getLogger(JMSReceiver.class);
    private static final String SOLJMS_INITIAL_CONTEXT_FACTORY =
"com.solacesystems.jndi.SolJNDIInitialContextFactory";

    private StorageLevel _storageLevel;
    private String _brokerURL;
    private String _vpn;
    private String _username;
    private String _password;
    private String _queueName;
    private String _connectionFactory;
    private Connection _connection;
    public JMSReceiver(String brokerURL, String vpn, String username, String password,
String queueName, String connectionFactory, StorageLevel storageLevel)
    {
        super(storageLevel);
        _storageLevel = storageLevel;
        _brokerURL = brokerURL;
        _vpn = vpn;
        _username = username;
        _password = password;
        _queueName = queueName;
        _connectionFactory = connectionFactory;
    }

    public void onStart()
    {

        log.info("Starting up...");

        try
        {



            Hashtable<String, String> env = new Hashtable<String, String>();
            env.put(InitialContext.INITIAL_CONTEXT_FACTORY,
SOLJMS_INITIAL_CONTEXT_FACTORY);
            env.put(InitialContext.PROVIDER_URL, _brokerURL);
            env.put(Context.SECURITY_PRINCIPAL, _username);
            env.put(Context.SECURITY_CREDENTIALS, _password);
            env.put(SupportedProperty.SOLACE_JMS_VPN, _vpn);

            javax.naming.Context context = new javax.naming.InitialContext(env);

            ConnectionFactory factory = (ConnectionFactory)
context.lookup(_connectionFactory);
            Destination queue = (Destination) context.lookup(_queueName);

            _connection = factory.createConnection();
            _connection.setExceptionListener(new JMSReceiverExceptionListener());

            Session session = _connection.createSession(false,
Session.CLIENT_ACKNOWLEDGE);
```

```
                MessageConsumer consumer;
                consumer = session.createConsumer(queue);
                consumer.setMessageListener(this);

                _connection.start();

                log.info("Completed startup.");
            } catch (Exception ex)
            {
                // Caught exception, try a restart
                log.error("Callback onStart caught exception, restarting ", ex);
                restart("Callback onStart caught exception, restarting ", ex);
            }
        }

        public void onStop()
        {
            log.info("Callback onStop called");
            try
            {
                _connection.close();
            } catch (JMSException ex)
            {
                log.error("onStop exception", ex);
            }
        }

        @Override
        public void onMessage(Message message)
        {
                log.info("Callback onMessage received" + message);
                store(message.toString());
                try {
                                message.acknowledge();
                        } catch (JMSException ex) {
                                log.error("Callback onMessage failed to ack message", ex);
                        }
        }

        private class JMSReceiverExceptionListener implements ExceptionListener
```

```
    {
        @Override
        public void onException(JMSException ex)
        {
            log.error("JMS exceptionListener caught exception, , restarting ", ex);
            restart("JMS exceptionListener caught exception, , restarting ");
        }
    }

    @Override
    public String toString()
    {
        return "JMSReceiver{" +
                "brokerURL='" + _brokerURL + '\'' +
                ", vpn='" + _vpn + '\'' +
                ", username='" + _username + '\'' +
                ", queueName='" + _queueName + '\'' +
                ", connectionFactory='" + _connectionFactory + '\'' +
                '}';
    }
}
```

8.2 JMSReceiverTest.java

Note that this test is the simple word count test in Spark, against the entire JMS message as a string. More practical use of this receiver would likely parse the JMS message into a serializable object prior to Spark analysis.

```
package com.solacesystems.jms.samples;

import java.util.regex.Pattern;

import javax.naming.NamingException;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.storage.StorageLevel;
import org.apache.spark.streaming.Duration;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.api.java.JavaReceiverInputDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;

import scala.Tuple2;

import com.google.common.collect.Lists;

public class JMSReceiverTest {

private static final Pattern SPACE = Pattern.compile(" ");
public static void main(String[] args)  {
           if (args.length < 6) {
             System.err.println("Usage: JMSReceiverTest <brokerURL> <vpn> <username>
<password> <queue> <connectionFactory>");
             System.exit(1);
           }

           // Create the context with a 1 second batch size
           SparkConf sparkConf = new SparkConf().setAppName("JavaCustomReceiver");
           JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, new
Duration(1000));

           // Create a input stream with the custom receiver on target ip:port and count
the
           // words in input stream of \n delimited text (eg. generated by 'nc')
           JavaReceiverInputDStream<String> lines;
```

```
               lines = ssc.receiverStream(
                        new JMSReceiver(args[0], args[1], args[2], args[3], args[4],
args[5], StorageLevel.MEMORY_ONLY_SER_2()));

           JavaDStream<String> words = lines.flatMap(new FlatMapFunction<String,
String>() {
             @Override
             public Iterable<String> call(String x) {
               return Lists.newArrayList(SPACE.split(x));
             }
           });
           JavaPairDStream<String, Integer> wordCounts = words.mapToPair(
             new PairFunction<String, String, Integer>() {
               @Override public Tuple2<String, Integer> call(String s) {
                 return new Tuple2<String, Integer>(s, 1);
               }
             }).reduceByKey(new Function2<Integer, Integer, Integer>() {
               @Override
               public Integer call(Integer i1, Integer i2) {
                 return i1 + i2;
               }
             });

           wordCounts.print();
           ssc.start();
           ssc.awaitTermination();
           ssc.close();
         }
}
```