DOCUMENT

SLE API Java Software Release Document

Prepared by	**SLE Java Team**
Reference	**SLES-SLE-SRELD-1001**
Issue	**5**
Revision	**0**
Date of Issue	**2019-10-29**
Status	**FINAL**
Document Type	**SRELD**
Distribution	

CHANGE LOG

Reason for change	Issue	Revision	Date
First draft	1	0	2016-05-02
sleapi_j#1, sleapi_j#6, sleapi_j#7	1	1	2016-11-28
SLEAPIJ-2	2	0	2018-12-11
SLEAPIJ-21,SLEAPIJ-23,SLEAPIJ-55, SLEAPIJ-60,SLEAPIJ-73	2	1	2019-04-30
SLEAPIJ-64, SLEAPIJ-66, SLEAPIJ-79, SLEAPIJ-80, , SLEAPIJ-81, , SLEAPIJ-82	5	0	2019-10-29

CHANGE RECORD

Issue 5	**Revision** 0		
Reason for change	**Date**	**Pages**	**Paragraph(s)**
SLEAPIJ-64, SLEAPIJ-66,SLEAPIJ-79, SLEAPIJ-80, SLEAPIJ-81, SLEAPIJ-82	2019-10-29		1.1,4.1,4.2.1,4.2.2,5, 6,8,10.2,10.3

Table of contents:

1 INTRODUCTION

1.1 Purpose and Scope

This document is the Software Release Document for the version 5.1.0 of SLE API Java. It provides details of the delivery together with the necessary information to build and install the SLE API Java on SLES15 with OpenJDK 11. In addition, it lists the changes applied during the porting of SLEAPI Java to SLES15 with java-11-openjdk-11 and describes how to get started.

1.2 Document Overview

The document contains the following major sections:

Section 1 - Introduction (this section) provides the purpose, scope and this document's overview.
Section 2 - Applicable and Reference Documents provides the list of reference documents.
Section 3 - Terms, Definitions and Abbreviated Terms provides a list of acronyms and terms used throughout this document.
Section 4 - Installation provides the installation details.
Section 5 - Setting Up the Eclipse Environment for importing the projects in Eclipse.
Section 6 - Differences Between C++ and JAVA SLE API describes the changes applied during the porting to Java activity.
Section 7 - Getting started describes how to set-up the classpath to start an application that uses the library, how to configure the library and how to start the Communication Server and the Default Logger.
Section 8 - SICF Parser describes the library and its usage.
Section 9 - SLE API and OSGi describes how the SLE API is used within the OSGI framework
Section 10 - Local Communication Server explains what is and how to use the local communication server
Section 11 - Limitations provides a list of limitations.

2 APPLICABLE AND REFERENCE DOCUMENTS

2.1 Applicable Documents

Ref.	Document Title	Issue and Revision, Date
[AD-1]		

2.2 Reference Documents

Ref.	Document Title	Issue and Revision, Date
[RD-1]	*ESA SLE API Reference Manual, SL-ANS-RF-0001*	Issue 4.8, 30 October 2009
[RD-2]	*Space Link Extension – Return All Frames Service Specification,* Recommendation for Space Data System Standards, CCSDS 911.1-B-4.	Blue Book, Issue 4, August 2016
[RD-3]	*Space Link Extension – Return Channel Frames Service Specification,* Recommendation for Space Data System Standards, CCSDS 911.2-B-3.	Blue Book, Issue 3, August 2016
[RD-4]	*Space Link Extension – Return Operational Control Field Service Specification,* Recommendation for Space Data System Standards, CCSDS 911.5-B-3.	Blue Book, Issue 3, August 2016
[RD-5]	*Space Link Extension - Forward Space Packet Service Specification,* Recommendation for Space Data System Standards, CCSDS 912.3-B-3.	Blue Book, Issue 3, August 2016
[RD-6]	*Space Link Extension – Forward CLTU Service Specification* CCSDS 912.1-B-4	Blue Book, Issue 4, August 2016
[RD-7]	*Space Link Extension – Internet Protocol for Transfer Services,* Recommended Standard, CCSDS 913.1-B-2	Blue Book, Issue 2, September 2015
[RD-8]	*Service Instance Configuration File ICD- Space Link Extension Services. Reference: EGOS-MDW-SLES-ICD-0001-i1r0*	2008-11-14
[RD-9]	*Service Link Extension – Application Program Interface for Return All Frames Service*	Magneta Book, September 2019

3 TERMS, DEFINITIONS AND ABBREVIATED TERMS

3.1 Acronyms

Acronyms	Description
API	Application Program Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CCSDS	Consultative Committee for Space Data Systems
CLTU	Communication Link Transmission Unit
DCW	Down Call Wrapper
ESA	European Space Agency
ESOC	European Space Operations Centre
ESTRACK	ESA Tacking Network
FSP	Forward Space Packet
GPL	GNU General Public License
IP	Internet Protocol
ISO	International Standardisation Organisation
LGPL	GNU Lesser General Public License
PDU	Protocol Data Unit
RAF	Return All Frames
RCF	Return Channel Frame
ROCF	Return Operational Control Fields
SE	Service Element
SHA	Secure Hash Algorithm
SI	Service Instance
SLE	Space Link Extension
SLES	SLE Services
TCP	Transmission Control Protocol
XML	Extensible Mark-up language

4 INSTALLATION

In order to compile the library, clone the SLEAPI java to an appropriate place on the target machine(SLES15 with OpenJDK11) , following the steps below:

Step	Description
1	Clone the SLEAPI repo
2	Enter the SLEAPI project folder using command ‘cd SLEAPI’
3	Build the library using ‘mvn install’ command on terminal.

SLEAPI version 5.1.0 uses maven build and compatible only with OpenJDK11 on SLES15 platform. Please make sure that OpenJDK11 and Maven is installed on SLES15 machine using below command on terminal.

1. `mvn –version`

The output of the command should be similar to below example snippet.

```
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-
17T18:33:14Z)
Maven home: /usr/local
Java version: 11.0.3, vendor: Oracle Corporation, runtime: /usr/lib64/jvm/java-11-
openjdk-11
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.12.14-197.4-default", arch: "amd64", family: "unix"
```

2. `java –version`

The output of the command should be similar to below example snippet.

```
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment (build 11.0.3+7-suse-3.24.1-x8664)
OpenJDK 64-Bit Server VM (build 11.0.3+7-suse-3.24.1-x8664, mixed mode)
```

In order to compile all the projects, enter the folder SLEAPI and type the command

```
mvn install
```

The terminal will show the success message for each project compilation. The relative jars are created for all the necessary projects in respective ‘target’ folder.

5 SETTING UP THE ECLIPSE ENVIRONMENT

The following table provides the necessary steps to import the SLEAPI projects into Eclipse, once the build procedure described in 4 has been executed. The SLEAPI version 5.1.0 is only compatible with Eclipse IDE 2018-12 or higher version of Eclipse IDE due to JAVA 11.

Step	Description
1	Open Eclipse
2	Set the Eclipse Workspace to the <SLE API Java Installation Env> directory
3	Import Maven->Existing Maven ProjectsImport ->Browse to the <SLE API Java Installation Env> directory. Import all projects and pom.xml from the <SLE API Java Installation Env> directory in Eclipse.
4	Change the target platform by including the SLEJAVA API Jars. From Eclipse Windows -> Preferences ->Plug-in Development-> Target Platform -> Select the Running Platform ->/ sle.java.api.target.target / sle.java.api.target.target

6 DIFFERENCES BETWEEN C++ AND JAVA SLE API

6.1 Introduction

This section describes the changes that have been applied for the migration of the SLE API from C++ to the native Java implementation [RD-9].

6.2 Migration Conventions

The following migration conventions have been applied:

- **Reference objects and output parameters**
  In the API implementation the signature of those C++ methods, which take a reference to an object as argument, has been modified in order to take a Reference object which mimics the method out-argument. For this purpose a Reference<T> class has been introduced in the IFS/GEN package.

- **HRESULT and the use of Exceptions**
  The signature of the interface methods, which return a HRESULT and take as argument a reference to an object, has been modified in the migration process as shown in the following example (interface ISLE_OpFactory):

  ```
  HRESULT CreateOperation(GUID& iid, SLE_OpType opType, void** operation);
  ```

  To

  ```
  <T extends ISLE_Operation> T CreateOperation(Class<T> classIid, SLE_OpType opType) throws SleApiException;
  ```

This approach avoids the use of a reference/holder class at interface level to mimic the out-argument of the C++ operation. The introduction of Java Exceptions allows handling error cases without losing information.

- **QueryInterface implementation**
  The QueryInterface method from the IUnknown interface has the following signature in C++

  ```
  HRESULT QueryInterface(GUID& iid, void** output);
  ```

  This has been modified in Java to

  ```
  <T extends IUnknown> T QueryInterface(Class<T> classIid);
  ```

  For this specific method there's no need to throw a SleApiException because in the new implementation the method returns null in case the requested interface is not supported.

- **Implementation of singletons**
  In C++ implementation of singletons includes:
  - A creator class that creates the instance of the singleton
  - The singleton class itself
  - In one specific case, a holder class that stores the reference to the singleton object

  The creator class has been removed in Java. In the singleton class the constructor is private and the public methods `initialiseInstance()` and `getInstance()` are used to create an instance of the class if it has never been created before and to access it.

- **AddRef() and Release() operations**
  The AddRef and Release methods have been removed due to the different memory management performed by the Java garbage collector.

- **IMalloc interface removed**
  The IMalloc interface and related implementation have been removed. Creation of byte array is performed using the standard <<new>> operator.

- **Enumerations**
  For each C++ enumeration an Enum class has been created in Java.

- **Types**
  The following table summarizes the mapping used for types and data structures

C++	Java
unsigned int	long
unsigned short	int
unsigned char	int
unsigned long	long
const char *	String
SLE_Octet * (alias for unsigned char *)	byte[]
void *	Object
list from STL	LinkedList

6.3 Tracing

The Java logging mechanism provided in the java.util.logging package has been widely used. In every class where logging is foreseen, a static private Logger attribute has been defined and used to log information with different logging

levels. This allows logging configuration from an application using the standard configuration mechanism provided by the java.util.logging framework.

6.4 SLE API Infrastructure Package

The SLE API Infrastructure package (IFS) implementation in C++ has been considerably simplified in Java, since many of its functionalities are already provided by the Java standard library.
The THR package inside IFS has been removed in favour of the standard implementations provided by the java.lang and java.concurrent packages, part of the Java standard library. These include thread creation, mutex, semaphores, atomic objects and thread-safe scalable collections.
The ISTL package inside IFS has been removed in favour of the standard implementations provided in the java.lang and java.util packages, part of the Java standard library. These include Lists, Sets, Maps and support for iterators.

6.5 Multiple SLE Instances

The ESA C++ SLE API design and implementation does not allow using the library in different roles (e.g. provider role and user role) concurrently in the same application. The Java SLE API allows the instantiation of the library several times, with independent configuration. As a result, it is possible for instance to have, in the same application a provider and a user role, using different configurations and working in a completely decoupled way.

In order to support a multi-instance design, the singletons present in the SLE API design have been converted into multi-tons. An instance of the library is identified by a string, which must be provided to the SLE API Builder at initialisation time, when retrieving the implementation of the builder:

```
ELSE_APIBuilder getESLEAPIBuilder(String instanceKey);
```

For backward compatibility reasons, the method

```
ELSE_APIBuilder getESLEAPIBuilder();
```

has been kept. Internally this method uses an auto-generated string by means of a UUID.

7 GETTING STARTED

7.1 Set-up the classpath

In order to run an application which uses the library, the jar files generated in the `dist` folder must be added to the classpath. The following example shows how to set-up the classpath to start the Test Harnes

```
export JAVA_HOME=<Path to Java 11>

export SLE_JAR=<SLE API Java Runtime Env>/esa.sle.java.api.core/target/

export CLASSPATH=<SLE API Java Runtime Env>/esa.sle.java.api.core/extlib/

export CP=$SLE_JAR/esa.sle.java.api.core-5.1.0.jar:$CLASSPATH/jasn1-compiler-
1.11.2.jar:$CLASSPATH/antlr-2.7.7.jar:$CLASSPATH/jasn1-1.11.2.jar

java –cp $CP esa.sle.impl.tst.systst.THApiexe -u|-p [-e] [-q] [-T] -x <proxy
database> -s <service element database> [-a <command file>] [-t <tracelevel>]
```

The thapi.sh script, used to start the Test Harness, is included in the runtime delivery and can be found in the folder <SLE API Java Runtime Env>/esa.sle.java.api.feature/test_scripts/ folder.

7.2 Configure the library

The following portion of code shows how in a user application it is possible to initialise the builder and obtain the interfaces ISLE_SIFactory and ISLE_UtilFactory that are used to create the Service Instance as explained in [RD-1]:

```
SLE_BindRole bindRole = SLE_BindRole.sleBR_initiator;
String configFilePathProxy = <Path to the Proxy User configuration
file>/DBProxyUser.txt";
String configFilePathSE = <Path to the SE User configuration file>/DBSEUser.txt";
ISLE_Reporter reporter = new ISLE_Reporter()
{
    @SuppressWarnings("unchecked")
    @Override
    public <T extends IUnknown> T queryInterface(Class<T> iid)
    {
        if (iid == IUnknown.class)
        {
            return (T) this;
        }
        else if (iid == ISLE_Reporter.class)
        {
            return (T) this;
        }
        Elseant de
        {
            return null;
        }
    }

    @Override
    public void logRecord(SLE_Component arg0, ISLE_SII arg1, SLE_LogMessageType
    arg2, long arg3, String arg4)
    {
        System.out.println(" ISLE_Reporter logRecord: " + arg4);
```

```
    }

    @Override
    public void notify(SLE_Alarm arg0, SLE_Component arg1, ISLE_SII arg2, long
    arg3, String arg4)
    {
        System.out.println("notify");
    }
};

// get the builder instance
ESLE_APIBuilder builder = ESLE_APIBuilder.getESLEAPIBuilder();
// ESLE_APIBuilder builder = ESLE_APIBuilder.getESLEAPIBuilder("SLEINSTANCE1");


// initialise the builder
builder.initialise(configFilePathSE,  configFilePathProxy,  reporter,  bindRole,
null);

// start the builder
builder.start();

// retrieve the interfaces
ISLE_SIFactory siFactory = builder.getSIFactory();
ISLE_UtilFactory utilFactory = builder.getUtilFactory();
```

7.3 Start the Communication Server and the Default Logger

The following example shows how to start the Communication Server from command line:

```
export JAVA_HOME=<Path to Java 11>

export SLE_JAR=<SLE API Java Runtime Env>/esa.sle.java.api.core/target/

export CLASSPATH=<SLE API Java Runtime Env>/esa.sle.java.api.core/extlib/

export CP=$SLE_JAR/esa.sle.java.api.core-5.1.0.jar:$CLASSPATH/jasn1-compiler-
1.11.2.jar:$CLASSPATH/antlr-2.7.7.jar:$CLASSPATH/jasn1-1.11.2.jar


java  -cp  $CP  esa.sle.impl.api.apipx.pxcs.Slecsexe  -d  <proxy  database>  [-t
<tracelevel>]
```

In order to launch the Default Logger from command line the following steps should be executed:

```
export JAVA_HOME=<Path to Java 11>

export SLE_JAR=<SLE API Java Runtime Env>/esa.sle.java.api.core/target/

export CLASSPATH=<SLE API Java Runtime Env>/esa.sle.java.api.core/extlib/

export     CP=$SLE_JAR/esa.sle.java.api.core-5.1.0.jar:$CLASSPATH/jasn1-compiler-
1.11.2.jar:$CLASSPATH/antlr-2.7.7.jar:$CLASSPATH/jasn1-1.11.2.jar

java -cp $CP esa.sle.impl.eapi.dfl.Sledflexe –d <proxy database> [-t <tracelevel>]
```

The scripts ‘slecs.sh’ and ‘sledfl.sh’, used to start the Communication Server and the Default Logger, are included in the runtime delivery under the <SLE API Java Runtime Env>/esa.sle.java.api.feature/test_scripts/ folder.

8 SICF PARSER

8.1 Introduction

This section describes the SICF file parser library, used to extract structured configuration information from the SICF files. This information is needed for the initialization of the SLEAPI service instances.

8.2 Concepts

Each SICF file contains the required configuration information to initialize the service instances in the context of the SLE API. Resource [RD-8] exposes the details of the services configuration structure. Each SICF file contains a list of services instance configuration parameters. The terms “service configuration” and “service descriptor” describe the data extracted from the SICF files, and therefore they will be interchangeable within this and the next chapter.
The SICF parser reads the SICF files from the specified directory and creates internally a data structure for handling the multiple service configurations.
The following table shows the main java data structure of the SICFParser project:

Table 1: SICFParser data structure

Class	Description
`Parser`	Creates for each file a `ServiceInstanceContainer` where a list of service descriptors is retained.
`ServiceInstanceContainer`	Each SICF file is mapped to an instance of this class. It contains a list of `SIDescriptor`.
`SIDescriptor`	This class is abstract. It contains the common configuration description of all the service instances. For example the service instance id, service instance start time and service instance stop time are present in this class. All the other service specific classes are extended from it.

8.3 Usage of the SICF Parser

The following code snippet describes how to use the SICF Parser:

```
Parser parser = new Parser(filePath);
parser.parseFile();
ServiceInstanceContainer value = parser.getServInstanceContainer();
```

In the first line a parser instance with the specified path to a SICF file is created. In the second one, the method `parseFile` processes each line of the file by adding the necessary information into an instance of `ServiceInstanceContainer`. The last line returns the extracted data from the SICF file configuration.

9 SLE API AND OSGI

9.1 Introduction

This chapter describes how the SLE API is used within the OSGI framework.

9.2 Structure SLE API and OSGI

The SLEAPI can be used within an OSGi-compliant system composed of the bundles shown in Table 2. Here the projects are listed in the order of execution of the build.xml files. For each project, a short description and the dependencies between bundles are provided.

Table 2: OSGi projects and bundles

Project Name	Project Description	Bundle name	Dependencies
esa.sle.sicf.si.descriptors	Contains service descriptors specific to each service instance type. Since the data types are used from the `SLE JAVA` project, the dependencies are on the `SLE JAVA API` and `SLE JAVA IMP`.	esa.sle.sicf.si.descriptors	esa.sle.java.api.core, esa.sle.java.api.core.test, esa.sle.java.api.core.test.harness
esa.sle.osgi	Provides an instance of the SLEAPI based on a specific configuration that includes the paths to the configuration files and the instance name.	esa.sle.osgi	esa.sle.java.api.core, esa.sle.java.api.core.test, esa.sle.java.api.core.test.harness
esa.sle.sicf.si.parser.file	Contains the tools for parsing and extracting the data from the SICF files. The collected configuration data is retained in the data structures defined in `SIDescriptor`.	esa.sle.sicf.si.parser.file	esa.sle.sicf.si.descriptors, esa.sle.java.api.core, esa.sle.java.api.core.test, esa.sle.java.api.core.test.harness
esa.sle.service.loader	Creates the service instance based on the configuration information notified by `SIRepository` and using the SLEAPI instance provided by `SLEJAVA_OSGi`.	esa.sle.service.loader	esa.sle.sicf.si.descriptors, esa.sle.osgi, esa.sle.java.api.core, esa.sle.java.api.core.test, esa.sle.java.api.core.test.harness
esa.sle.si.repository	Uses the services offered by `ServiceLoader` and the Java 7 `WatchService` to notify the subscribed `ISLE_SIRepositoryInform` in case of file changes.	esa.sle.si.repository	esa.sle.sicf.si.descriptors, esa.sle.sicf.si.parser.file, esa.sle.service.loader, esa.sle.java.api.core, esa.sle.java.api.core.test, esa.sle.java.api.core.test.harness

9.3 Building with Maven

Each project from Table 2 contains the source folder, a META-INF folder and the pom.xml file. The pom.xml file allows compiling the packages of the project, creating the jars of the project and the MANIFEST.MF for the

corresponding jars. For all the projects the pom.xml file contains a property which points to the latest standard OSGI jars (i.e. `osgi.cmpn-6.0.0.jar` and `osgi.core-6.0.0.jar)`, which are included in the delivery. A global pom.xml file is provided at projects level to build the projects in the order specified in Table 2. Therefore, in order to compile all the projects, type `mvn install` from the command line.

9.4 Input Configuration

Since most of the services exposed by the bundles in Table 2 are registered to the OSGi platform in a declarative way, an application that handles service instances must provide multiple xml configuration files in its OSGI-INF folder. Those files are described in the following sections where the notations listed in Table 3 are applied.

Table 3: Notations

Notation	Description
<proxy database >	Should be replaced with the path to the configuration file of the proxy.
<service element database>	Should be replaced with the path to the configuration file of the service element database.
<watched folder>	Should be replace with the path to the SICF folder

9.4.1 Library Instance

When the activate method is invoked in the EE_SLE_LibraryInstance class, an instance of the SLE API based on a specific configuration (that includes the paths to the configuration files and the instance name) is created. In case the client application publishes the ISLE_Reporter and the ISLE_TimeSource services, they will be used to instantiate the library. If the ISLE_TimeSource is not available at this stage, the implementation provided by the SLE API of the ISLE_TimeSource interface will be used. The Library Instance property is as instance key to instantiate the SLE API. In this way, it is possible to deploy several instances with different configurations.

The created object is then registered with the OSGi platform under the interface ISLE_LibraryInstance in a declarative way using the following xml file:

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
activate="activate" deactivate="deactivate" immediate="true" name="SLE Library
Service">
   <implementation class="esa.sle.osgi.impl.EE_SLE_LibraryInstance"/>
   <property name="seConfigFilePath" value="database/DBSEUser.txt" />
   <property name="proxyConfigFilePath" value="database/DBProxyUser.txt" />
   <property name="instanceName" value="myLibraryInstance" />

   <service>
      <provide interface="esa.sle.osgi.ISLE_LibraryInstance"/>
   </service>

   <reference bind="setReporter"
         cardinality="0..1"
         interface="ccsds.sle.api.isle.iapl.ISLE_Reporter"
         name="myReporter"
         policy="dynamic"
         unbind="unsetReporter" />

   <reference bind="setTimeSource"
         cardinality="0..1"
         interface="ccsds.sle.api.isle.iapl.ISLE_TimeSource"
         name="myTimeSource"
```

```
         policy="static"
         unbind="unsetTimeSource" />

</scr:component>
```

9.4.2 Service Instance Repository

The `esa.sle.si.repository` bundle creates the SIRepository object and registers it with the OSGi platform under the interface ISLE_SIRepository in a declarative way. The properties needed for this service are the instance name and the path to the SICF folder.

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
activate="activate" deactivate="deactivate" immediate="true" name="SI
repository">
   <implementation class="esa.sle.si.repository.SIRepository"/>
   <property name="folderPath" value="watched"/>
   <property name="instanceName" value="repo1"/>

   <service>
      <provide interface="esa.sle.service.loader.ISLE_SIRepository"/>
   </service>

</scr:component>
```

9.4.3 Service Loader

The esa.sle.service.loader bundle creates the EE_SLE_ServiceLoader object and registers it with the OSGi platform under the interface ISLE_ServiceLoader in a declarative way, using the following xml file:

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
activate="activate" deactivate="deactivate" immediate="true" name="Service
Loader">
   <implementation class="esa.sle.service.loader.impl.EE_SLE_ServiceLoader"/>
   <property name="name" value="myServiceLoader" />

   <service>
      <provide interface="esa.sle.service.loader.ISLE_ServiceLoader"/>
   </service>

   <reference bind="setLibraryInstance"
         cardinality="1..1"
         interface="esa.sle.osgi.ISLE_LibraryInstance"
         name="sleLibraryInstance"
         policy="static"
         target="(instanceName=myLibraryInstance)"
         unbind="unsetLibraryInstance" />

   <reference bind="setSIRepository"
         cardinality="1..1"
         interface="esa.sle.service.loader.ISLE_SIRepository"
         name="siRepository"
         policy="static"
         target="(instanceName=repo1)"
         unbind="unsetSIRepository" />

</scr:component>
```

In order to be instantiated, this service needs a reference to an instance of the library with a specific name and an instance of the SI repository with a specific name.
The ISLE_LoadedServiceInstance service is published programmatically on creation when the presence of a new Service Instance in the SICF folder is notified to the EE_ServiceLoader object.

10 LOCAL COMMUNICATION SERVER

This version of the SLE API includes a local communication server that can be started automatically by the SLE API library. The local communication server is useful the SLE provider application is designed to manage all the SLE service instances within a single JVM. If this is the case, there is no need to start a separate communication server process.

The local communication server behaves exactly as a remote communication server, with the following differences:

- Data exchange uses in-memory piped streams, hence avoiding the overhead of interprocess communication;
- The server is started within the SLE provider application.

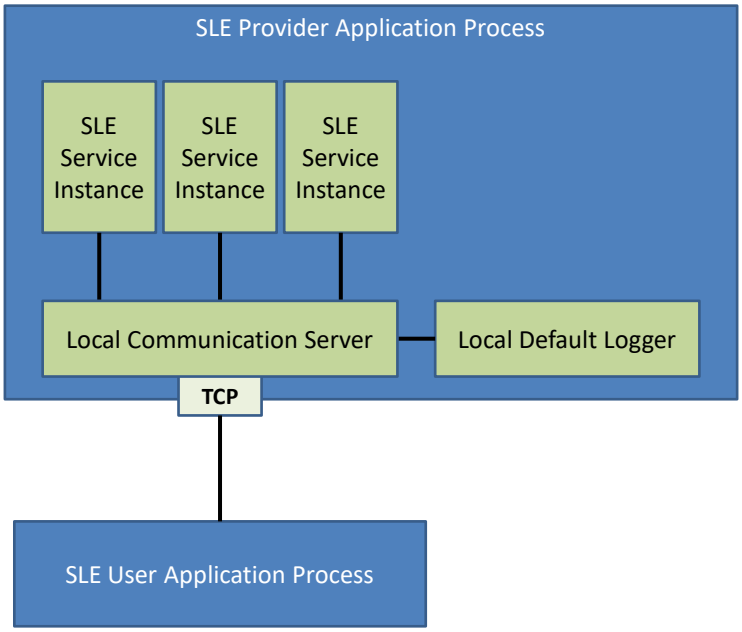

Figure 1 - Local Communication Server

In order to use the local communication server, it is enough to change the properties “CS_ADDRESS” and “DEFAULT_REPORTING_ADDRESS” of the DB Proxy Provider configuration file: if the property “CS_ADDRESS” starts with the prefix “LOCAL-“, then the SLE API library builder will start the local communication server and it will use an internal local default logger for it. The value of the property is used to identify the internal “local socket” to be used for internal communication: in this way, the support for future multi-instance SLE API libraries within the same JVM is ensured.
If the local communication server is used, it is mandatory that also the “DEFAULT_REPORTING_ADDRESS” property start with the “LOCAL-“prefix.

11 LIMITATIONS

The bundle `esa.sle.si.repository` listed in Table 2 uses the File Watcher service as provided by the java.nio package. The purpose of the watcher instance is to detect changes of the SICF files within the watched directory. If the watched directory resides on a UNIX system, the changes should be conducted from the same UNIX machine and not from another remote server. This limitation is also described in the Java Platform, Standard Edition 7 API Specification for the Interface `WatchService`.