Contents

FIGURE INDEX 8

ABOUT THIS MANUAL 11

CONVENTIONS 11

INTRODUCTION 13

OVERVIEW 13

TOP REASONS TO USE SMARTPHONE & CROSS PLATFORM COMMUNICATION TOOLKIT (SCCT) 13

SCCT APPLICATION EXAMPLES 14

COMMUNICATING DATA ACROSS A NETWORK, ON A COMPLEX ASSEMBLY LINE 14

PUBLISHING REAL TIME DATA TO MOBILE DEVICES 14

DELIVERING HIGH QUALITY MAINTENANCE SERVICE TO YOUR CUSTOMER 15

SUPPORTED PLATFORMS 15

PUBLISHER LIBRARY 15

SUBSCRIBER LIBRARY 15

INSTALLING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 16

GETTING STARTED WITH THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 19

COMMUNICATION CONCEPTS 19

SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT’S PALETTE 21

USING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 22

PUBLISHER CLASS 23

CREATING A PUBLISHER 24

PUBLISHING ANALOG DATA 25

DEFINING A SYSTEM CONFIGURATION 26

MANAGING WELCOME KIT 26

PUBLISHING DIGITAL DATA 27

SUBSCRIBER CLASS 29

CREATING A SUBSCRIBER 31

STARTING AND STOPPING DATA TRANSMISSION 31
SENDING MESSAGES 33
RECEIVING MESSAGES 35
HOW TO CHECK THE STATE OF YOUR CONNECTION 35
CLOSING COMMUNICATION 35

CREATING AND MANAGING ALERTS AND USER REQUESTS 36
OVERVIEW 36
READING AVAILABLE MESSAGES 36
SENDING MESSAGE TO A SPECIFIC SUBSCRIBER 36
MANAGING USER MESSAGES 37
NOTIFYING AN ERROR 37

SOURCE SELECTION 38
ENABLING SOURCE FILTER 39
SELECTING SOURCE LIST 39
SENDING A FILTER COMMAND 40
MANAGING BROADCASTING MESSAGES TO BY-PASS SELECTED SOURCE FILTER 42

PUBLISHING AND RECEIVING IMAGES 43
    Publishing an image (example 1) 43
    Publishing an image (example 2) 43
    Receiving an image (example 1) 44

PUBLISHING AND RECEIVING 2D ARRAYS 45

PUBLISHING AND RECEIVING FILES 45
    File exchange example 1 45
    File exchange example 2 46

ADVANCED OPTIONS 46
    CONTROLLING ACTIVE CONNECTIONS 46
    READING ACTIVE CONNECTION COUNT 47
    READING ACTIVE CONNECTION ADDRESSES 47
    READING ACTIVE SUBSCRIBERS Profiles 47
Figure Index

Figure 1—Smartphone & Cross-platform Communication Toolkit’s package information page. .......................... 17

Figure 2—Package information page when upgrading from previous version on Smartphone & Cross-
platform Communication Toolkit. ................................................................................................................ 17

Figure 3—Smartphone & Cross-platform Communication Toolkit’s license agreement page .................. 18

Figure 4—After installation, a summary page shows package status. .......................................................... 18

Figure 5 - Communication scheme of SCCT .............................................................................................. 19

Figure 6—How data packages are organized at subscriber side. ............................................................... 20

Figure 7—An application can process incoming packages according to their data types. ......................... 20

Figure 8—Smartphone & Cross-platform Communication Toolkit’s palette. ............................................. 21

Figure 9—Simple Publisher example with analog data. .............................................................................. 25

Figure 10 - Use of the Welcome Kit with a configuration cluster. ............................................................. 27

Figure 11—Simple Publisher example with digital data. ............................................................................ 27

Figure 12—Simple Subscriber example that reads analog data only. ....................................................... 31

Figure 13—TransferStatus.vi determines the flow of packets between publisher and subscriber. .......... 31

Figure 14—Simple subscriber example that reads digital data only. ......................................................... 32

Figure 15—Simple subscriber that reads configuration cluster before reading analog data. ................... 32

Figure 16—PublishData_Message.vi example ........................................................................................... 34

Figure 17—Message sent by publisher case in broadcast mode (left) and to a single client (right) ............ 34

Figure 18—Reading a message from Publisher. ......................................................................................... 35

Figure 19—Using Properties node and readConnected.vi to check connection status. ......................... 35

Figure 20—Using available Message Count property node to check if subscriber’s messages are present. 36

Figure 21—Sending message to a specific subscriber with PublishData_Message.vi. ............................... 36

Figure 22—Managing received message example .................................................................................... 37

Figure 23—Notifying LabVIEW error to all subscribers with notifyError.vi. ............................................. 37

Figure 24—Scenario with a multiple data source server and clients that need a subset of published data. 38

Figure 25—Set sourceFilterEnabled=true when you plan to take advantage of data filtering at server side. 39

Figure 26—Source list selection ............................................................................................................... 40
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Sending a filter request to server.</td>
</tr>
<tr>
<td>28</td>
<td>Sending two different filter requests to server.</td>
</tr>
<tr>
<td>29</td>
<td>Reading FilterId for every received data package.</td>
</tr>
<tr>
<td>30</td>
<td>Broadcast transmission.</td>
</tr>
<tr>
<td>31</td>
<td>Publishing an image with PublishData/image1.vi.</td>
</tr>
<tr>
<td>32</td>
<td>Publishing an image stored on disk, with PublishData/image2.vi.</td>
</tr>
<tr>
<td>33</td>
<td>Receiving an image from a client with Read Image.vi.</td>
</tr>
<tr>
<td>34</td>
<td>Publishing different types of 2D arrays.</td>
</tr>
<tr>
<td>35</td>
<td>Reading different types of 2D arrays.</td>
</tr>
<tr>
<td>36</td>
<td>Publishing a file with PublishData/file1.vi.</td>
</tr>
<tr>
<td>37</td>
<td>Publishing a file with PublishData/file2.vi.</td>
</tr>
<tr>
<td>38</td>
<td>Use Read activeSubscribersCount.vi to check the count of active connections.</td>
</tr>
<tr>
<td>39</td>
<td>Use Read activeSubscribersAddresses to get info about active subscribers.</td>
</tr>
<tr>
<td>40</td>
<td>Use property nodes to get information about active subscribers.</td>
</tr>
<tr>
<td>41</td>
<td>Use Read subscriberProfile.vi to retrieve information about an active connection.</td>
</tr>
<tr>
<td>42</td>
<td>Enabling the request FIFO with enableEventLogging property node.</td>
</tr>
<tr>
<td>43</td>
<td>This example shows how to manage “Authentication Successful” event.</td>
</tr>
<tr>
<td>44</td>
<td>This example shows how to manage the case a connection has been closed by client.</td>
</tr>
<tr>
<td>45</td>
<td>Adding API-Key to active API-Key set.vi.</td>
</tr>
<tr>
<td>46</td>
<td>Removing an API-Key from active API-Key set.vi.</td>
</tr>
<tr>
<td>47</td>
<td>Changing maxDt to check subscriber’s system time.</td>
</tr>
<tr>
<td>48</td>
<td>Writing and reading safety mode.</td>
</tr>
<tr>
<td>49</td>
<td>Publisher application and subscriber application use code field to identify the proper data type.</td>
</tr>
</tbody>
</table>

Both apps have to know the data type associated to the codes. 

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>With Publisher, use publishData.vi method to send XML documents.</td>
</tr>
<tr>
<td>51</td>
<td>With Subscriber, use sendCustomXMLData.vi to send XML documents to Publisher.</td>
</tr>
<tr>
<td>52</td>
<td>With Publisher, use Read customXMLData.vi to process XML packets.</td>
</tr>
<tr>
<td>53</td>
<td>With Subscriber, use Read customXMLData.vi to process XML packets.</td>
</tr>
</tbody>
</table>
FIGURE 54 – With Publisher and Subscriber you have different methods and property nodes to get the count of available XML unprocessed documents. ........................................................................................................ 55

FIGURE 55 – Use UpdateLocation.vi to send current publisher position to clients......................................................... 55

FIGURE 56 – In Subscriber, use sendCustomXMLData.vi to send XML documents to publisher.......................... 56

FIGURE 57 – In subscriber, use ReadAvailableLocationCount.vi to verify the number of unprocessed location packages. .................................................................................................................................................. 56

FIGURE 58 – "Additional Exclusions" page must be properly configured. .............................................................. 57

FIGURE 59 – "Build errors" with error code 1562. ........................................................................................................ 58
ABOUT THIS MANUAL

The Smartphone & Cross-platform Communication Toolkit User Manual describes the virtual instruments (VIs) used to communicate and pass data between LabVIEW and either a local or remote application. You should be familiar with the operation of LabVIEW, your computer and your computer operating system.

CONVENTIONS

The following conventions appear in this manual:

› The › symbol leads you through nested menu items and dialog box options to a final action. The sequence Tools › Options directs you to pull down the Tools menu, select Options item.

Bold

Bold text denotes items that you must select or click on the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic

Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

monospace

Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.

monospace italic

Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
**INTRODUCTION**

This chapter describes the installation procedure, installed components, and the main features of the Smartphone & Cross-platform Communication Toolkit.

**OVERVIEW**

The Smartphone & Cross-platform Communication Toolkit is an add-on package for communicating data through applications. The toolkit contains a set of high level functions for sending your application data and advanced functions for customized tasks.

The following list describes the main features of the Smartphone & Cross-platform Communication Toolkit.

- Works over any TCP/IP connection
- Works over Local Area Networks as well as Internet connections.
- Implements the publisher – subscriber pattern (also known as Observer pattern)
- Authenticates subscribers through an API-KEY.
- Controls in background the state of every connection to identify loss of communication.
- Publishes GPS coordinates to manage mobile systems.
- Works with platform independent Data format and communicate with multiple platforms at the same time: third party vendors have implemented toolkit to develop on Android platform, Java, .NET and VB, Unix and iOs.

**TOP REASONS TO USE SMARTPHONE & CROSS PLATFORM COMMUNICATION TOOLKIT (SCCT)**

Adopting this toolkit you have the following advantages:

**Simplify communication:** don’t care about communication details over a TCP communication channel, SCCT does it for you.

**Multiple platforms are supported:** exchange your data with a protocol supported on a wide range of platforms and programming languages.

**It’s reliable:** many applications have been created with this toolkit around the world.

**Speed up your development activity:** this toolkit allows the creation of distributed application and let you save a lot of your time.
SCCT has been created by LabVIEW developers for LabVIEW developers; it includes some great features supported on LabVIEW platform only (see Publishing CustomData in this manual) so if you need to exchange data with other LabVIEW applications, take advantage of the power of SCCT to deliver high quality code and reduce developing time.

Today applications need to retrieve information from database, bar code readers, OCR systems, remote data acquisition or technical operators with mobile devices. Often you have to create systems capable to exchange data with legacy applications created on different platforms. Every time you have to design an ad hoc communication protocol, code both server and client side routines, debug them. Every time! With SCCT you have a new tool that let you save a lot of time, lets you focus on your project’s core and manage multiple communications at the time. In the following examples you are going to see how SCCT can dramatically improve the quality of your software solutions.

SCCT application examples
SCCT can be successfully applied in many real world situations. In this chapter we discover where SCCT helps developers to succeed to deliver high valued solutions.

Communicating data across a network, on a complex assembly line
Creating an application that broadcasts part numbers and barcodes coming from databases or bar code readers, to all computers on an assembly line, as schematized below. On modern assembly lines, many computers control single processing stations and they need to exchange data among them to know, part numbers, print codes and certificates, store test results on different databases. SCCT provides many functionality to exchange data with applications developed with heterogeneous programming languages. Besides, when your customer asks you to show some data on a mobile device, SCCT is capable to communicate with a large variety of smart phones and tables and you do not have to change a single line of code of your software. SCCT cares of all connected devices and communicate your data to all of them at the same time, receives user’s requests and organize them in a time-ordered FIFO so that you can process them easily.

Publishing real time data to mobile devices
When you have to publish acquired data in real time, SCCT is the best choice because with few SCCT Vis your applications are supported on a wide range of platforms: Android, iPhone, etc. SCCT is safe because every device must authenticate to your application and you can control who is connected in any time. Using SCCT you deliver open solution to your customers because they can use their favorite mobile platform to connect to your data. And if customers change mobile device family, you don’t need to modify your code!
Delivering High Quality Maintenance Service to Your Customer

Including SCCT into your existing applications, let you offer fast support to your customers. You and your customers can monitor deployed applications everywhere, with a tablet, phone, and desktop. Integrating SCCT capabilities into existing applications makes easy debug and signal analysis of deployed systems around the world.

Supported Platforms

SCCT is composed of two main components:

- publisher library
- subscriber library

Publisher Library

This library let you create a full-featured publisher, which authenticates incoming subscribers, check connection status, sends data to all active publishers and passes their request to your application.

This library is available as a set of Vis for LabVIEW 2010 or later.

To get more details or download an evaluation copy of this library please visit:

http://www.toolsforsmartminds.com/products/SCCT.php

Subscriber Library

This library let you create a subscriber which handles all communication details with a publisher so you don’t have to. It receives data packages and present them to your application according to their data types.

This library is available for the following platforms and languages:

<table>
<thead>
<tr>
<th>Name</th>
<th>Operating System</th>
<th>Development Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCCT Subscriber for LabVIEW</td>
<td>Windows</td>
<td>LabVIEW 2010</td>
</tr>
<tr>
<td>SCCT Subscriber for Java</td>
<td>Java VM 5.0 or later</td>
<td>Java</td>
</tr>
<tr>
<td>SCCT Subscriber for Android</td>
<td>Android 2.1 or later</td>
<td>Java</td>
</tr>
<tr>
<td>SCCT Subscriber for Linux</td>
<td>Linux Kernel 2.6.30</td>
<td>ANSI - C</td>
</tr>
<tr>
<td>SCCT Subscriber for iPhone/iPad</td>
<td>iOS</td>
<td>Objective C</td>
</tr>
<tr>
<td>SCCT Subscriber for Javascript</td>
<td>HTML5 browser</td>
<td>Javascript</td>
</tr>
</tbody>
</table>

To get more details or download your free copy of SCCT subscriber library, please visit:

http://www.toolsforsmartminds.com/products/SCCT.php
INSTALLING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT

Smartphone & Cross-platform Communication Toolkit is shipped as a VI Package Manager. You can download it from

www.toolsforsmartminds.com/products/SCCT.php

Before installing Smartphone & Cross-platform Communication Toolkit you must install a copy of VI Package Manager on your machine. You can get a free copy of VIPM at this address:

http://www.jki.net/vipm/download

To install Smartphone & Cross-platform Communication Toolkit double click open the vip file:

smartphone_&_crossplatform_communication_toolkit-x.x.x.xx.vip

and follow the installation wizard. Package contains LabVIEW VIs as well as documents in PDF format accessible from LabVIEW (Help ▶ TOOLS for SMART MINDS ▶ SCCT User Guide) and libraries to create applications with Java, Android and .Net (in c:\SCCT\cross-platform libs). Visit http://www.toolsforsmartminds.com to get more details.
Double click on `smartphone_&_crossplatform_communication_toolkit-x.x.x.xx.vip` or open it from VIPM and the package information page appears, as shown in the next figure.

![Figure 1 - Smartphone & Cross-platform Communication Toolkit's Package Information Page.](image1)

Press "Install" button in the upper left corner, VIPM shows available actions. When upgrading Smartphone & Cross-platform Communication Toolkit to a new version, VIPM displays "upgrade" in Action column.

![Figure 2 - Package Information Page when Upgrading from Previous Version on Smartphone & Cross-platform Communication Toolkit.](image2)
Press "Continue" in the lower right corner to install/upgrade the selected package.

**Figure 3—Smartphone & Cross-platform Communication Toolkit’s License Agreement Page.**

Before stating installation procedure, you have to accept Smartphone & Cross-platform Communication Toolkit license agreement, as shown above.

**Figure 4—After installation, a summary page shows package status.**

When installation completes, a summary page is displayed. After installation is completed, LabVIEW must be closed and restarted.
GETTING STARTED WITH THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT

COMMUNICATION CONCEPTS

Smartphone & Cross-platform Communication Toolkit implements a fully bidirectional communication channel between an application identified as server and a pool of applications called clients. Your application manages communication through SCCT’s APIs and publish/receive data with high-level functions. SCCT cares of all connected systems and handles data transmission so you don’t have to. The clients have to authenticate themselves with a password (in SCCT terminology, it is called API-Key). The following figure represents the communication scheme:

![Communication Scheme of SCCT](image)

When an application wants to receive data, first asks the publisher to be inscribed among the active subscribers. Publisher will accept all incoming requests with the valid API-KEY.

Received data are organised into separated FIFOs so that subscriber application can process data packages according to their types. The following figure shows how data packages are sent and organized on the publisher and subscriber side, respectively. Starting from a unique FIFO queue of different package types, the subscriber side task arranges the received packages in different FIFO structures.
In this architecture, packages can be processed very easily with dedicated Vis. In many real life applications the exact order in which data packages are generated is not very important, and with LabVIEW multithread capability you can dedicate specific task to every package type and deliver robust software solution as shown below.

Because of the wide range of devices the Smartphone & Cross-platform Communication Toolkit works with, some portability issues remain. Consider the following issues when choosing your way to publish data.

- Some smartphones and tablets use CPU with low computing power so are not able to receive and process large streams of data.
- Smartphone & Cross-platform Communication Toolkit uses a platform independent data format and subscribers require some computing power to decode data streams into their specific binary format.
- Smartphone & Cross-platform Communication Toolkit handles communication with subscribers as a set of peer to peer connections and every data you publish is transmitted individually to each subscriber. So you have to identify the right size of your data streams to avoid band saturation over your communication channel.
- Some data types are not supported on all platforms.
SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT’S PALETTE

After you installed the toolkit, you can start using Smartphone & Cross-platform Communication Toolkit from a blank VI. To get access to library’s palette, browse through Functions ➔ Data communication ➔ TOOLS for SMART MINDS, as shown below.

Publisher VIs have light red background icons, Subscriber VIs have white background icons.

Figure 8- Smartphone & Cross-platform Communication Toolkit’s palette.
Using the Smartphone & Cross-platform Communication Toolkit

This Toolkit is composed of two main components: Publisher that creates the server side of your communication system and the subscriber that implements the client side. Publisher and Subscriber work together to pass data from one application, which holds data to many applications on different systems (MS-Windows OS family, Linux, Apple systems, mobile devices, etc.). Publisher uses a platform independent data format to transmit your data so that all subscribers can read them. In this way, you add a little overhead to a simple transmission that uses binary data format, but gain a great portability and opportunity to communicate with heterogeneous systems. To better understand the way this communication works, consider the following example. A publishing company receives requests from different subscribers that want to receive a magazine. As long as they are subscribed, they receive the magazine. When they don’t want to receive it anymore, simply cancel their subscription. Your application can implement more than one Publisher each of them works on different port of your machine. An application can contain publishers and subscribers together, working with different remote machines at the same time.

Either objects work in background of your application with specific tasks that are created and destroyed automatically. This library has been created by LabVIEW developers and includes some advanced features supported only by LabVIEW apps. Please be careful when design your communication solutions so that your data can be properly treated by all subscriber. So when your system is designed to work with LabVIEW programs only, CustomData packages greatly simplify data transfer among processes, but cannot be processed by subscribers created, for example, with SCCT for Java.
**Publisher Class**

Publisher is a Class with methods and properties detailed in the following tables.

### Publisher Class properties

<table>
<thead>
<tr>
<th>Property name</th>
<th>description</th>
<th>Read/write</th>
</tr>
</thead>
<tbody>
<tr>
<td>availableMessageCount</td>
<td>gets the count of received messages from subscribers</td>
<td>Read only</td>
</tr>
<tr>
<td>enableEventLogging</td>
<td>Sets/get the management of subscriber requests</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Port</td>
<td>Gets the actual port number</td>
<td>Read only</td>
</tr>
<tr>
<td>API-Key</td>
<td>Gets the actual API-Key</td>
<td>Read/Write</td>
</tr>
<tr>
<td>activeSubscribersCount</td>
<td>Gets the count of active connections</td>
<td>Read only</td>
</tr>
<tr>
<td>activeSubscribersAddresses</td>
<td>Gets the IP address and port of each active connection</td>
<td>Read only</td>
</tr>
<tr>
<td>maxDT</td>
<td>Sets/get the maximum allowed time difference between publisher time clock and subscribers time click</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

### Publisher Class methods

<table>
<thead>
<tr>
<th>method name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add API-Key</td>
<td>Add an API-Key. Active connections are not affected by this change</td>
</tr>
<tr>
<td>ClearWelcomeKit</td>
<td>Clears welcome kit package list</td>
</tr>
<tr>
<td>enableEventLogging</td>
<td>Enables the management of requests</td>
</tr>
<tr>
<td>notifyError</td>
<td>Transmit a message to a specific connection or to all active connections with a LabVIEW error code and source.</td>
</tr>
<tr>
<td>publishData</td>
<td>Polymorphic function that adapts to input data and publishes a data package.</td>
</tr>
<tr>
<td>PublishData_2D_DBL_ARRAY</td>
<td>Publishes analog data as 2D array</td>
</tr>
<tr>
<td>PublishData_1D_BOOL_ARRAY</td>
<td>Publishes a 1D array of Boolean</td>
</tr>
<tr>
<td>PublishData_BOOL_ARRAY</td>
<td>Publishes a boolean array</td>
</tr>
<tr>
<td>PublishData_I16_ARRAY</td>
<td>Publishes an array of word signed integers (I16)</td>
</tr>
<tr>
<td>PublishData_I32_ARRAY</td>
<td>Publishes an array of long signed integers (I32)</td>
</tr>
<tr>
<td>PublishData_I64_ARRAY</td>
<td>Publishes an array of quad signed integers (I64)</td>
</tr>
<tr>
<td>PublishData_SGL_ARRAY</td>
<td>Publishes an array of single-precision floating-point data (SGL)</td>
</tr>
<tr>
<td>PublishData_DBL_ARRAY</td>
<td>Publishes an array of double-precision floating-point data (DBL)</td>
</tr>
<tr>
<td>PublishData_STRING_Array</td>
<td>Publishes an array of strings</td>
</tr>
<tr>
<td>publishData_CUSTOM_DATA</td>
<td>Publishes a custom data. Custom data can be a LabVIEW cluster/object. Only LabVIEW clients can handle this data correctly</td>
</tr>
<tr>
<td>publishData_XML_DATA</td>
<td>Publishes an XML document</td>
</tr>
<tr>
<td>publishData_File_1</td>
<td>Publishes a file your application has loaded in memory. This function let you create file on a remote platform without saving it on local disk</td>
</tr>
<tr>
<td>publishData_File_2</td>
<td>Publishes an existing file. This function loads selected file, computes MD5 and transfer data</td>
</tr>
<tr>
<td>publishData_Image_1</td>
<td>Publishes an image present in memory as binary string.</td>
</tr>
<tr>
<td>publishData_Image_2</td>
<td>Publishes an image loading it from disk. This function works only with a specific set of image formats</td>
</tr>
<tr>
<td>publishData_Message</td>
<td>Publishes a message, composed by a message code and its description</td>
</tr>
<tr>
<td>PublishData_NewLocation</td>
<td>Publishes new server GPS location</td>
</tr>
<tr>
<td>Read activeSubscriberAddresses</td>
<td>Returns the IP address and TCP port of active subscribers</td>
</tr>
<tr>
<td>Read activeSubscriberCount</td>
<td>Returns the count of active subscribers</td>
</tr>
<tr>
<td>Read API-Keys</td>
<td>Returns current API-Key list</td>
</tr>
<tr>
<td>Read availableAnalogDataCount</td>
<td>Returns the count of analog data packages (2D array of doubles) to be processed</td>
</tr>
<tr>
<td>Read availableConfigurationCount</td>
<td>Returns the count of configuration packages to be processed</td>
</tr>
<tr>
<td>Read availableCustomDataCount</td>
<td>Returns the count of custom data packages to be processed</td>
</tr>
<tr>
<td>Read availableXMLDataCount</td>
<td>Returns the count of unprocessed Custom XML packages</td>
</tr>
<tr>
<td>Read availableData</td>
<td>Polymorphic function that returns the count of unprocessed packages by the numerator</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Read availableDigitalDataCount</td>
<td>Checks the queue related to a specific data type.</td>
</tr>
<tr>
<td>Read availableEventCount</td>
<td>Returns the count of digital data packages to be processed.</td>
</tr>
<tr>
<td>Read availableFilesCount</td>
<td>Returns the count of unprocessed events.</td>
</tr>
<tr>
<td>Read availableImageCount</td>
<td>Returns the count of unprocessed image packages.</td>
</tr>
<tr>
<td>Read availableMessageCount</td>
<td>Returns the count of unprocessed message packages.</td>
</tr>
<tr>
<td>Read availableNewlocationCount</td>
<td>Returns the count of unprocessed location packages.</td>
</tr>
<tr>
<td>Read configuration</td>
<td>Reads a system’s configuration package.</td>
</tr>
<tr>
<td>Read constrainCheck</td>
<td>Return a system’s configuration package.</td>
</tr>
<tr>
<td>Read CustomData</td>
<td>Reads a custom data package.</td>
</tr>
<tr>
<td>Read XMLData</td>
<td>Reads an XML data package.</td>
</tr>
<tr>
<td>Read DigitalData</td>
<td>Returns a digital data package.</td>
</tr>
<tr>
<td>Read AnalogData</td>
<td>Returns an analog data package.</td>
</tr>
<tr>
<td>Read Array</td>
<td>Polymorphic function that returns an array data package.</td>
</tr>
<tr>
<td>Read Event</td>
<td>Gets the first unprocessed event.</td>
</tr>
<tr>
<td>Read File</td>
<td>Reads a file package.</td>
</tr>
<tr>
<td>Read Image</td>
<td>Reads an image and returns a LabVIEW’s image data cluster.</td>
</tr>
<tr>
<td>Read Image(planData)</td>
<td>Reads an image in its raw format. Image is returned as a binary string.</td>
</tr>
<tr>
<td>Read Message</td>
<td>Returns a message package.</td>
</tr>
<tr>
<td>Read NewLocation</td>
<td>Returns a location package.</td>
</tr>
<tr>
<td>Read Port</td>
<td>Returns the current TCP port used by Publisher.</td>
</tr>
<tr>
<td>Read Request</td>
<td>Reads the first unprocessed request from one of the active connections.</td>
</tr>
<tr>
<td>Read SubscriberProfile</td>
<td>Returns the profile(connection info) of selected subscriber.</td>
</tr>
<tr>
<td>Remove API-Key</td>
<td>Removes an API-Key from API-KEY list. Active connections are not affected.</td>
</tr>
<tr>
<td>RemovePackageFromWelcomeKit</td>
<td>Removes package with specific tag from welcome kit.</td>
</tr>
<tr>
<td>selectedSourceList</td>
<td>Selects the source list when connecting to a data server.</td>
</tr>
<tr>
<td>sendMessage</td>
<td>Transmits a message to a specific connection or to all active connections.</td>
</tr>
<tr>
<td>startPublisher</td>
<td>Implements a publisher object.</td>
</tr>
<tr>
<td>stopPublisher</td>
<td>Destroys a Publisher object.</td>
</tr>
<tr>
<td>updateConfiguration</td>
<td>Transmits a new system configuration to all active connections.</td>
</tr>
<tr>
<td>updateLocation</td>
<td>Updates server location to all clients.</td>
</tr>
</tbody>
</table>

**Creating a Publisher**

To create a publisher in your LabVIEW application use palette **SCCT > Publisher**. The following examples show how to create a publisher in few click.

To create a publisher in your application use **startPublisher.vi**. To create a new Publisher two parameters are required:

- Publisher port, which is the TCP port that Publisher uses to manage all TCP connections;
- API-Key, which is the connection password that subscribers must communicate to publisher to be authenticated.
Take care to use one of the available port of your machine. Some ports are reserved for other common applications like port 21 to FTP, 80 to HTTP and so on. Moreover, you have to check that the chosen port is open on your company firewall.

SCCT Communication with Internet Explorer 9 requires Flash Plug-in. SCCT permits to communicate with Internet Explorer 9 through the boolean input \textit{supportIE9WithFlashPlugin} of \texttt{startPublisher.vi}. By default, \textit{supportIE9WithFlashPlugin} is enabled. If it is disabled, application cannot communicate with IE9. SCCT 3.0 supports Internet Explorer 10 and it does not require Flash Plug-in to communicate with SCCT. For this reason, \textit{supportIE9WithFlashPlugin} can be disabled with Internet Explorer 10.

**Publishing analog data**

With SCCT library, publish analog data requires a single Vi, \texttt{publishData.vi} which is a polymorphic Vi that adapts to input and allows to send different types of data. Analog data are published as a 2D array of double. Notice that every row of your 2D array contains samples of a single channel. If you are publishing data to mobile devices, remember that computational power of smartphones is far lower than PC and they cannot process and display huge amount of data per second. The following figure illustrates a basic example where analog data coming from an acquisition device are sent to all clients. Publishing rate is the ruled by the way your application collects data from physical device.

In the previous example, a Publisher is created with \texttt{startPublisher.vi} that immediately creates all necessary data structures, tasks and takes care of all incoming connections. Before starting to transmit analog data, a system configuration cluster is published through \texttt{updateConfiguration.vi}. Please note that configuration cluster must be filled according to the analog signals you want to transmit. Channel configuration must describe each channel of your data acquisition. The system configuration is explained in detail in the following section.
In the while loop your acquired data are published to the active subscribers directly. If no subscribers are connected, data are discarded.

When the loop terminates, `stopPublisher.vi` closes all active tasks and flushes FIFO with user requests.

**Notice:** `startPublisher.vi` creates some background tasks which handle the data transfer to and from active subscribers. These tasks are also responsible of checking if connections are lost in case your application is not publishing any information. All these tasks periodically check if your top level VI is running and stop automatically when your top level VI stops.  
`stopPublisher.vi` stops background tasks. When this VI is executed, all active connection are terminated by the publisher and no more connections can be created.

### Defining a System Configuration

The system configuration cluster includes several information, such as device name, product type, location, and a detailed description about analog or digital channels. Every analog channel is described by a description, measurement unit, min value, max value, and sampling rate. This information can be used by subscribers to properly adapt graphs and charts. The use of the system configuration package makes sense only when SCCT is used to publish analog and/or digital data. When system configuration changes, `updateConfiguration.vi` can be used to refresh the configuration to all active subscribers at run time.

Publisher requires a system configuration cluster with at least one analog channel description or one digital line description. If you connect an empty configuration cluster, `startPublisher.vi` rises up an error and doesn’t start. The reason of this is behavior because, in past, SCCT was used to publish data from acquisition boards. Now, many new features are available and SCCT can be used to transmit a wide range of different data types, not related to any data acquisition, but configuration constrain remains to keep compatibility with legacy code.

### Managing Welcome Kit

With SCCT you can take advantage of Welcome Kit feature. If a new subscriber connects to server after that a data package has already been published, the new subscriber does not receive this data package. To avoid this problem, the Welcome Kit feature can be used. If the Welcome Kit is specified when a data package is published, the publisher sends this data to subscriber when it connects to the server, despite the subscriber was not connected when the data package was sent.

There are many scenarios where it is useful to send some packages to a new subscriber when connection is established. For example, when you publish analog or digital data, a system configuration package should be sent to describe the transmitted channels. If the Welcome Kit is specified in `updateConfiguration.vi`, the publisher sends the system configuration cluster to subscriber when it connects to the server. In this way, the system configuration is the first package that subscriber receives.
when it establishes a new connection. The following figure shows how to use the Welcome Kit with the configuration package cluster.

**Figure 10 - Use of the Welcome Kit with a configuration cluster.**

You can also use the Welcome Kit when you want that new subscribers receive a welcome message when connection is established and after they have to receive server status. SCCT can manage these situations so your application doesn’t have to. Whenever you publish some data, you just specify which packages have to be kept in a special set called ”Welcome Kit”. SCCT identifies packages you wish to add to Welcome Kit by welcomeTag, a string you connect to the proper connector when you send data to clients, as shown below.

**Publishing digital data**

In this example, a Publisher is created with startPublisher.vi that immediately create all necessary data structures and tasks and takes care of all incoming connections. Please note that configuration cluster must be filled according to the digital lines you want to transmit. Configuration must describes each line of your data acquisition. In the while loop your acquired data are published directly to the active subscribers. If no subscribers are connected, data are discarded. When the loop terminates, stopPublisher.vi closes all active tasks and flushes FIFO with user requests.

**Figure 11 - Simple publisher example with digital data.**

Please note in the above figure that when you start a publisher you create a publisher object with a small set of parameters. The new object will manage by itself most of communication details. If you want to modify the way communications are handled, use LabVIEW property nodes to set desired values to publisher options. The time that occurs between the execution of publishData.vi and the effective
transmission of data package to all active subscribers is usually few milliseconds and depends mostly on data size and not on the active connection number.
**Subscriber Class**

Subscriber is a Class with methods and properties detailed in the following tables:

<table>
<thead>
<tr>
<th>Subscriber Class properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>availableMessageCount</strong></td>
</tr>
<tr>
<td><strong>availableAnalogData</strong></td>
</tr>
<tr>
<td><strong>availableConfiguration</strong></td>
</tr>
<tr>
<td><strong>availableDigitalData</strong></td>
</tr>
<tr>
<td><strong>connected</strong></td>
</tr>
<tr>
<td><strong>connectionStatus</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subscriber Class methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>openConnection</strong></td>
</tr>
<tr>
<td><strong>closeConnection</strong></td>
</tr>
<tr>
<td><strong>Read Message</strong></td>
</tr>
<tr>
<td><strong>Read analogData</strong></td>
</tr>
<tr>
<td><strong>Read digitalData</strong></td>
</tr>
<tr>
<td><strong>Read configuration</strong></td>
</tr>
<tr>
<td><strong>Read customXMLData</strong></td>
</tr>
<tr>
<td><strong>Read customData</strong></td>
</tr>
<tr>
<td><strong>discardData</strong></td>
</tr>
<tr>
<td><strong>transferStatus</strong></td>
</tr>
<tr>
<td><strong>Send Request</strong></td>
</tr>
<tr>
<td><strong>Send CustomXMLData</strong></td>
</tr>
<tr>
<td><strong>Read availableAnalogDataCount</strong></td>
</tr>
<tr>
<td><strong>Read availableDigitalDataCount</strong></td>
</tr>
<tr>
<td><strong>Read availableConfigurationCount</strong></td>
</tr>
<tr>
<td><strong>Read availableMessageCount</strong></td>
</tr>
<tr>
<td><strong>Read availableCustomDataCount</strong></td>
</tr>
<tr>
<td><strong>Read availableCustomXMLDataCount</strong></td>
</tr>
<tr>
<td><strong>Read connected</strong></td>
</tr>
<tr>
<td><strong>Read connectionStatus</strong></td>
</tr>
</tbody>
</table>
Select Source List

Select among the available list of sources a subset of data sources. Server will transmit to client packages that belong to selected sources.

To implement the subscriber in your application and receive data from a source, you must know three parameters:

- Data source address, which is usually the IP address of the machine where Publisher is running on.
- Data source port that is the TCP port of the Publisher.
- API-Key is the key necessary to be authenticated by publisher. If a subscriber uses a wrong API-Key connection is refused by publisher.

When your application succeeds to connect, publisher sends immediately a configuration of remote system i.e. analog channel descriptions, measurement unit, and range of all signals, digital line descriptions and system location GPS coordinates.
CREATING A SUBSCRIBER

In the following example you create a simple subscriber with openConnection.vi that needs three parameters: Publisher address (default value is localhost), Publisher port (default value in 8081) and API-Key. Data transmission is started through transferStatus.vi.

![Image](image-url)

**Figure 12 - Simple Subscriber Example that reads analog data only.**

STARTING AND STOPPING DATA TRANSMISSION

Publisher doesn’t start data transmission automatically. After connection, your subscriber must tell to the Publisher to start sending data. If your subscriber doesn’t need fresh data, use transferStatus.vi with a FALSE constant to tell the publisher to stop sending data. To re-start data transmission use transferStatus.vi with TRUE constant.

Publisher doesn’t buffer data if transmission is stopped. The following diagram illustrates how SCCT manages data transmission.

![Image](image-url)

**Figure 13 — TransferStatus.vi determines the flow of packets between Publisher and Subscriber.**
To get analog data use **Read analogData.vi** that returns a packet of data. Background tasks takes care of all received packets and enqueues them in a FIFO so you can process all packets without data loss. To know the number of available packet use the property node that returns the data packets count. The following figure illustrates the case of reading digital lines only. Notice that at the end of the while loop you always have to close communication with publisher. When communication is closed and you want to open again the communication, use **openCommunication.vi**.

![Figure 14](image1.png)

**Figure 14- Simple subscriber example that reads digital data only.**

Publisher automatically sends a communication cluster that describes the remote system. Use this cluster to properly format your graph and chart setting x and y scales, as shown in the following figure.

![Figure 15](image2.png)

**Figure 15- Simple Subscriber that reads configuration cluster before reading analog data.**

Notice: **openCommunication.vi** creates a background task which handles the data transfer to and from the publisher. This task is also responsible of checking if connection is lost in case publisher is not sending any information. This task periodically checks if your top level VI is running and stops automatically when your top level VI stops.

Optionally, you can add an application description to the openConnection.vi. Server side application will use the description to identify your application. This is useful when the server has to properly identify subscriber’s name, but they change IP address or port at every new connection, typically when DHCP is used to assign IP addresses. If description is not specified, server will identify the connection with the...
string “xxx.xxx.xxx.xxx:yyyy”, xxx.xxx.xxx.xxx is subscriber application IP address and yyyy is local port used to connect to server.

**Sending Messages**

Subscribers can send textual requests to the Publisher with PublishData_Message.vi. A message is a cluster composed of a numeric code and a message string. The following figure illustrates how messages are transmitted from a client application to a SCCT server.

Every package regardless of its data type follow the same path indicated in the above figure. When data is transmitted from server to client, it follows the schema indicated below.

The main difference between data transmitted by a client (=subscriber) and a server (=publisher) is that when a client sends a package, this is received by only one application: the server regardless of the number of active connection, instead when a server sends a package, this can be received by all active clients (broadcasting transmission) or by a specific client identified by its IP address.

Messages are sent immediately and receiver processes them in the same order they are received. If message string is empty, message is not posted. You can specify an additional numeric code if your application uses numeric codes to identify message sets.
The above figure illustrates two cases where a publisher sends a message: in the first case, message package is transmitted to all active clients, instead in the second case only one client will receive the message. Receiver address is a string composed by client IP address and TCP port. You can retrieve client addresses with Read activeSubscriberAddresses.vi.
RECEIVING MESSAGES
Publisher/Subscribers can receive textual messages with Read message.vi. A message is a cluster composed of a timestamp, a string and a numeric code. The following figure illustrates the path of alerts.

Messages are received and enqueued in a dedicated FIFO. Your application has to process incoming messages and take care of FIFO size. You can use Read Message.vi to extract from the message FIFO the oldest received message.

HOW TO CHECK THE STATE OF YOUR CONNECTION
You can monitor connection state with connected property node or readConnected.vi, which return TRUE if your connection is still alive. A connection is alive also if Publisher is not sending data to your subscriber. Publisher and subscriber exchange acknowledge packets to verify if connection is still active so you don’t have to.

CLOSING COMMUNICATION
When connection is no more necessary, use closeCommunication.vi to close the open connection. This VI destroys all unprocessed data and closes background tasks. After this VI, subscriber object cannot be used and a new instance must be created with openCommunication.vi.
CREATING AND MANAGING ALERTS AND USER REQUESTS

OVERVIEW
When an active communication is established between publisher and subscriber, they can exchange some messages. Every message is composed of a numeric code and a message string. Every message contains a numeric code, a timestamp, a connection identifier, an event code and an optional data string. Publisher uses a FIFO to enqueue all incoming messages in the order they are received. Your application can identify which subscriber is sending the message through its ipAddress:port identifier.

READING AVAILABLE MESSAGES
Subscribers send their request to your application and Publisher keeps them in a FIFO together with some messages it sends to inform your application about the connection status and the communication between publisher and subscribers. Use availableRequests property node to retrieve the number of received request that your application has to process. When a request is processed with getRequest.vi, availableRequests is decremented by 1.

![Figure 20 - Using available message count property node to check if subscriber's messages are present.](image)

SENDING MESSAGE TO A SPECIFIC SUBSCRIBER
Your application can communicate with active subscribers with custom messages. To send a message use PublishData_Message.vi, as shown in the following figure:

![Figure 21 - Sending message to a specific subscriber with PublishData_Message.vi.](image)

Use message string to add additional information to your message. To send a message to every active subscriber receiverAddress must be empty string, instead, if you want to send a message to a specific subscriber, connect receiverAddress to the specific address and port of its connection. Message string cannot be empty. If try to send an empty string message, error is generated with error code 5002.
MANAGING USER MESSAGES

User message (i.e. message from subscriber) is inserted into a FIFO by Publisher so your application can process all messages in the order they are received from Publisher. The following figure illustrates the right way to manage the messages:

![Managing received message example](image)

First, you have to check that `found` indicator is TRUE, if `found` is FALSE, no message is available. Every message has its `time` field, `senderAddress` field that identifies the subscriber that sent the message. If connected client manages multiple data sources, `source` indicator specifies which source has generated the message.

NOTIFYING AN ERROR

When a LabVIEW error arises and must be notified to one or more subscribers, use `notifyError.vi` which composes a message with error code and error description from LabVIEW error cluster. The following example shows how to use it. Notice that you can specify a `ipAddress:port` reference to send error only to a specific listener. When error cluster contains no error, no message is sent.

![Notifying LabVIEW error to all subscribers with notifyError.vi](image)
**Source selection**

To reduce data traffic, SCCT includes a feature called source selection that can significantly reduce the used band on your communication channel. It applies in cases where server manages multiple data sources (every data source is identified by its ID number, a 32bit integer) and clients need to receive data from a subset of these sources, at the same time. The following figure illustrates the case you have a server with 4 acquisition devices and you client app needs data from 2 devices at the same time.

In scenarios like the one above, you have to consider source selection at server side. Use the following schema to improve your communication performance and reduce used band between server and client.

**STEP 1 - CONNECTION WITH FILTER ENABLED**

Client connects to server with filterSelectionEnabled=True option

After connection is established, server has an empty list of selected sources so no data packages are sent to client.

**STEP 2 - SERVER INFORMS CLIENT ABOUT AVAILABLE SOURCES**

Your server has to publish a package that contains a list of available data sources (usually it is part welcome kit packages).

**STEP 3 - CLIENT SENDS ITS DATA SOURCE SELECTION**

At client side, your application shows source list and user chooses which sources wants to use. Client
sends a selected source list package, with its specific VI.

**STEP 4 - SERVER SETS UP SOURCE FILTER (AND WAITS DATA TRANSFER=TRUE)**

Server receives the selected source list package and updates the client's list so data packages are filtered properly.

**STEP 5 - CLIENT ENABLES DATA TRANSFER**

Client app sends start=true with dataTransfer.vi and enables data transmission.

Selected source list can be changed at any time during connection so clients can switch among available data sources without closing existing connection.

**Enabling Source Filter**

Source filter is enabled at connection time so your subscriber has to inform the server that is going to take advantage of source selection filter to reduce data traffic. Specify `sourceFilterEnable=True` when try to open a connection as show in the following figure:

![Figure 25 - Set `sourceFilterEnable=true` when you plan to take advantage of data filtering at server side.](image)

By default `sourceFilterEnabled` is set to false, so server sends all data packages regardless of their source.

**Selecting Source List**

Use `selectSourceList.vi` to select the sources your client wants to receive. This command has only if you specified `filterEnabled=true` at openConnection.vi. Before you select one or more sources, server keeps its source list empty so no data packages are sent. The following figure illustrates the right way to select a source list:
Source list can be selected before the connection start or during the connection session. Your client has to know the source ID and description list. Your server has to inform clients about available sources. Source list can be published with different type packages: message, custom XML Data, customData. Define a source list so that clients can decode it easily.

**SENDING A FILTER COMMAND**

As mentioned before, subscribers can use `selectSourceList.vi` in order to receive data from specific sources only and reduce the amount of transmitted data. Moreover, SCCT allows to select a portion of data. For example, if a subscriber needs only a subset of available published signals, it can request the channels it needs through `FilterRequest.vi`. In this way, for each selected source, subscribers can request to receive only a portion of data. Notice that with both filtering and source list, SCCT performs a server-side filtering.

Filtering is available for analog data, 2D array of booleans, strings, I16, I32, I64, DBL, and SGL data. For analog data, filtering allows to select specific channels. Otherwise, for 2D arrays, filtering permits to select a subset of the available published array data. The following figure shows how to send a filter command to server for different data types.
A different FilterId is generated for every filter request. When server performs a new filtering operation, it increments the FilterId field in the same way it is modified by the client. In this way, server and client maintain a counter in common. The FilterId permits the client to relate every received data package to the relative filter request. FilterId is fundamental to process received data in the right way. For example, if server manages a 100x100 array and a client needs a 11x11 subarray (from index 5,5 to 15,15), the client sends a filter request. As it is the first filter request sent to the server, the filter request must generate a FilterId = 1. When the filter request is performed, server sends the required subarray and marks transmitted packages with FilterId = 1. Later, if the client needs a different 11x11 subarray, it sends another filter request. As it is the second filter request, the filter request generates a FilterId = 2. The server performs this new filter request, sends the new subarray and marks transmitted packages with FilterId = 2. The following figure schematizes this example. The use of FilterId permits the client to link every received package with the first or the second filter request.

The FilterId of every data package is stored in the `additionalInfo` cluster. Use the following code to read the FilterId.
MANAGING BROADCASTING MESSAGES TO BY-PASS SELECTED SOURCE FILTER

In previous paragraphs we have learned that we can filter data packages at server side and server doesn't send data packages if they don't match the source list filter. Source filter can be bypassed by broadcasting transmission. When server has to notify special packages (for example alarm messages or server state changes) you can set Broadcast flag to True and your package regardless of data type, is sent to all active connections.

![Diagram of broadcasting transmission](image)

The previous example illustrates how to force broadcast transmission. First package is sent to clients who added source with ID=1 to their source list. Second and third package are sent to all clients regardless of their source filter. Notice that when broadcast flag is set to True, source ID and description are ignored by transmission task.

![Diagram of broadcast transmission with XML data](image)
**PUBLISHING AND RECEIVING IMAGES**

SCCT 2.1 allows clients and server to exchange images. Images are passed with a list of attributes assigned at run time by user applications. The following examples illustrates how to send and receive image with a publisher, the same concepts can be used to send and receive images with a client.

**PUBLISHING AN IMAGE (EXAMPLE 1)**

In this example is shown the case a server publishes images to all connected clients.

**PUBLISHING AN IMAGE (EXAMPLE 2)**

The following example illustrates the case a server publishes images saved on disk. SCCT can load and publish images with a single function, as shown below. When images you need to publish are stored on disk and you don’t need to display them in your application, use `publishData_image2.vi` to transmit selected image to all connected clients. Every image can be described by a name and with a list of attributes.

![Image](image.png)
Receiving an Image (Example 1)

In this example is shown the case server receives an image from a client. The strategy used to receive data from connected clients is always the same for all data type. Read Image.vi returns a Boolean indicator to indicate if images are available. Is at least one image is available, it is
PUBLISHING AND RECEIVING 2D ARRAYS

SCCT permits clients and server to publish and receive 2D arrays of different data types. The supported data types are array 2D of booleans, strings, word signed integers (I16), long signed integers (I32), quad signed integers (I64), double-precision floating-point (DBL), and single-precision floating-point (SGL).

The following examples illustrate how to publish and read 2D arrays. Array data are published through `publishData_BOOL_Array.vi`, `publishData_I16_Array.vi`, and `publishData_string_Array.vi`; otherwise, you can use the polymorphic VI `publishData.vi`. You can read arrays through `read.vi` that is a polymorphic VI.

![Figure 34 - Publishing different types of 2D arrays](image1)

![Figure 35 - Reading different types of 2D arrays](image2)

PUBLISHING AND RECEIVING FILES

SCCT can enable clients and server to exchange images. Images are passed in their native formats with a list of attributes assigned at run time by applications.

FILE EXCHANGE EXAMPLE 1

In this example is shown the case a server publishes files to all connected clients. Files are loaded from a folder and transferred to clients with some attributes specified at application level.
In the above example file is loaded at application level and sent with three attributes.

**FILE EXCHANGE EXAMPLE 2**

Files can be sent in a very way when their full path is known. The following example sends a file and set **computeMD5** flag to True so that SCCT computes MD5 value and associate it to the transmitted file.

Remember that SCCT validates every transmitted and received package regardless of package data type so MD5 value is not necessary to clients to validate integrity of received files.

**ADVANCED OPTIONS**

This chapter illustrates advances features of SCCT for LabVIEW.

**CONTROLLING ACTIVE CONNECTIONS**

Publisher Vis handle all incoming connections and close inactive connections so you just focus on your main application and forget all issues related to data transmission. In some cases you want to know the number of active connections and the address of subscribers. Usually you can map all incoming
connections using getRequest.vi and filtering events such “connection successful” and “connection closed” and “connection timeout”, which help you to map all active and closed connections.

**Reading active connection count**

To know the count of active connections at a specific time, use Read **activeSubscribersCount.vi** that returns the number of active connections.

![Figure 38 - Use Read activeSubscribersCount.vi to check the count of active connections.](image)

**Reading active connection addresses**

To know the addresses of active subscribers, use Read **activeSubscribersAddresses.vi** that returns a string array: every string is a subscriber’s address in the form ipAddress:port. The array is ordered by connection time so the first element of the array is related to the active subscribers that connected first. Closed connections don’t appear in the array.

![Figure 39 - Use Read activeSubscribersAddresses to get info about active subscribers.](image)

You can use **activeSubscriberCount** property node to get active connection count and **activeSubscriberAddress** property node to get active connection addresses, as shown below:

![Figure 40 - Use property nodes to get information about active subscribers.](image)

**Reading active subscribers profiles**

At server side you can get detailed information about active connections. Every connection is identified by client IP address and port number. Use address:port as search key to retrieve subscriber’s profile. The client profile is composed of:
• Start connection time. It indicates time and date when client has established connection.
• Description. It contains the description of the client application, specified when client has connected to server.
• Platform. It indicates the client platform: LabVIEW, Android, iOS, HTML5.
• Release. It indicates the SCCT release.
• Connection ID. It is an univocal code generated for every active connection.
• Selected Sources. It is a list of the selected sources by the client.
• Data transfer status. It indicates the current data transfer status of the client (enabled or disabled).
• Source Filter status. It indicates the current source filter status of the client (enabled or disabled).
• API-Key. It is the API-Key used by the client to connected with server.
• Timeout. It represents the timeout of the client.

The following example illustrates the case you search user profile who establishes a new connection.

![Figure 41 - Use Read subscriberProfile.vi to retrieve information about an active connection.](image)

When a connection terminates, its data are removed immediately form publisher internal database so you cannot retrieve its parameters. If Read subscriberProfile.vi returns an empty cluster, means that the connection has been closed after the request has been queued.

**Managing events**

Publisher provides a FIFO used to keep track of communication events. This FIFO is enabled by default and can be enabled / disabled with eventLogging property node (see above). Event data cluster contains the following information:

- **timestamp** is the local time when event is occurred.
- **Event** describes the type of event. The available event types are:
- **authentication successful**, occurs when a new connection is properly established. Is a remote device tries to connect, but its connection parameters are not correct and connection is refused, this event is not generated.

- **conn closed by client**, occurs when a client terminates connection.

- **timeout expired**, occurs when a connection is lost abnormally, for example when communication channel fails. In this situation your application cannot be sure that all transmitted packets have been received by client associated to aborted connection.

- **senderAddress** is the address of new subscriber, in the form IpAddress:port.

Use **availableEventCount** property node to check is new events have been generated. When a new event is present you can extract it from its event Queue with **getEvent.vi** method as shown in the following figures.

![Figure 42 – Enabling the request FIFO with enableEventLogging property node.](image)

As shown above, instead of property node you can set the flag “enableLogging” with **enableLogging.vi** and get its value with **isEventLoggingEnabled.vi**, respectively.

Events are useful when your application keeps track of existing connections and must execute special tasks when a new connection is established or an existing one is closed. Below you see a simple example: a welcome message is sent only to new connected applications. **senderAddress** is the address associated to the connection that generated the event.
In the following figure a different situation is managed: when a connection is terminated, and your application has to do some action like deleting temporary files associated to the connection.

**Figure 43 – This example shows how to manage “Authentication Successful” event.**

**Figure 44 – This example shows how to manage the case a connection has been closed by client.**

### Changing API-Key at Run-Time

Publishers are created with an API-Key they use to authenticate every incoming connection. Notice that SCCT doesn’t accept null API-KEY values. When publisher is started you can add and remove API-KEYs programmatically using **Add API-KEY.vi** and **Remove API-KEY.vi**. Publisher can use unlimited number of API-KEYs and when a subscriber tries to connect and provides its API-KEY, publisher search that KEY among the API-KEY set created at run time. You can also remove existing API-KEY except when you have only one KEY: publisher aborts KEY removal when it has only one KEY.

### Adding a New API-Key

The following example illustrates the case you want to add an API-KEY to the active set of KEYS. Existing connections are not affected by this command.
Adding API-KEYs can be done with Add API-KEY.vi or a property node as shown above.

**Removing an existing API-KEY**

The following example illustrates how to remove existing API-KEY from active set of KEYS. Existing connections are not affected by API-Key changes.

Adding API-KEYs can be done with Remove API-KEY.vi or a property node as shown above.

**Changing max DT**

When subscribers try to connect, the request includes their system time. If subscriber’s system time differs from publisher’s system time more than maxDt and maxDt is greater than zero, connection is refused. By default, maxDt value is zero so subscriber’s system time is not checked. To enable time control, use maxDt property node to set the absolute value of maximum distance, in seconds, between local and remote system’s time.

Writing and Reading Safety Mode

Packages sent with SCCT are enqueued in a FIFO structure. If data are produced at higher rate than data processing speed, or if connection bandwidth is limited with respect to data production rate, the number of pending packages and memory requirement of server application increase. To avoid this problem, SCCT permits to enable Safety Mode. When safe mode is enabled, no more than 100 packages
can be pending (awaiting to be transmitted) for every connection. Exceeding packages are discarded. To enable/disable safety mode, use Write safetyModeEnabled. By default, Safety Mode is disabled. To read Safety Mode, use read safetyModeEnabled.

\[\text{FIGURE 48 – WRITING AND READING SAFETY MODE.}\]
Publishing and receiving custom data (Supported only by LabVIEW applications)

If both side applications use SCCT for LabVIEW, then some advanced features are available. These features rely on LabVIEW variant data type and their data package are not supported on other programming languages. If more than one custom data type is used, custom data types must be associated to a numeric code or string description so that subscribers can properly identify what data has been sent from publisher. In the following example, publisher side, on the left, encode two different data types and send them with publishData.vi. Notice they are associated to code 1 and 2. Subscriber application, on the right, receive the packages with Read customData.vi, and uses code field to properly process data type 1 and data type 2.

Figure 49—Publisher application and subscriber application use code field to identify the proper data type. Both apps have to know the data type associated to the codes.
Publishing and Receiving Custom XML Data

In addition to messages and requests, both sides of communication channel can exchange XML documents. XML packets have been added to SCCT to allow server and client applications exchanging complex custom data with a single function. Remember that XML documents are not checked by SCCT Vis. You must verify XML document’s correctness before transmission to avoid unpredictable behaviors by receiving applications. The following figures illustrate the available set of Vis in Publisher and Subscriber’s libraries. Notice that on Publisher side, XML transmission is done with the Polymorphic VI PublishData.vi.

**Figure 50 — With Publisher, use publishData.vi method to send XML documents.**

**Figure 51 — With Subscriber, use sendCustomXMLData.vi to send XML documents to publisher.**

**Figure 52 — With Publisher, use readCustomXMLData.vi to process XML packets.**
Publisher and Subscriber manage XML document transfer in the same way.

**Publishing and receiving server location**

When publisher is running on a mobile system (for example on a vehicle) and it is provided of a GPS or similar localization system, it can update its current location to subscribers with `updateLocation.vi`. At client side, subscribers can manage location packages in the same way they handle other package types, with `Read availableLocationCount.vi` and `Read newLocation.vi`. The following examples illustrate how to integrate location management in your applications.
Use the same programming approach used to handle incoming data packages, check found indicator before processing location packages. Use Read \textit{availableLocationCount.vi} to retrieve the number of unprocessed location packages.

\textbf{Figure 56 – In Subscriber, use \texttt{sendCustomXMLData.vi} to send XML documents to publisher.}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure56.png}
\end{figure}

\textbf{Figure 57 – In subscriber, use \texttt{Read availableLocationCount.vi} to verify the number of unprocessed location packages.}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure57.png}
\end{figure}
**Build Specifications**

Application with Subscriber VIs

Creating executables that include VIs from Smartphone & Cross-platform Communication Toolkit requires a specific setting of "Additional Exclusions" tab page. Flags from "Modify project library file after removing unused members" must be unchecked, as shown in the following figure.

![Additional Exclusions Page](image)

**Figure 58 – "Additional Exclusions" page must be properly configured.**

Smartphone & Cross-platform Communication Toolkit is password-protected so if above additional exclusion is not unchecked, the following error stops building process:
Figure 59 – "Build errors" with error code 1562.
### APPENDIX A – ERROR TABLE

In the following tables are indicated the error codes generated by VIs

#### PUBLISHER ERROR CODES

<table>
<thead>
<tr>
<th>code</th>
<th>description</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>Publisher is not started</td>
<td>A method or property has been called before creating the Publisher</td>
</tr>
<tr>
<td>5001</td>
<td>Invalid port</td>
<td>When you start the publisher you have to provide a valid TCP port.</td>
</tr>
<tr>
<td>5002</td>
<td>invalid message text</td>
<td>You cannot send a message with an empty text</td>
</tr>
<tr>
<td>5003</td>
<td>Invalid API-Key</td>
<td>API-Key cannot be an empty string</td>
</tr>
<tr>
<td>5004</td>
<td>Invalid configuration</td>
<td>Configuration cluster must contain at least 1 analog channel or 1 digital line</td>
</tr>
<tr>
<td>5005</td>
<td>Invalid analog data</td>
<td>Analog data must be a 2D array with exactly the number of rows equal to the number of analog channels as defined in configuration cluster.</td>
</tr>
<tr>
<td>5006</td>
<td>Invalid digital Data</td>
<td>Digital data must be a 1D array with size equal to the number of digital lines as defined in configuration cluster.</td>
</tr>
<tr>
<td>5007</td>
<td>Empty data are not published</td>
<td>Empty arrays are not published</td>
</tr>
<tr>
<td>5008</td>
<td>internal library is corrupted</td>
<td>Unexpected error during creation of communication tasks</td>
</tr>
</tbody>
</table>
### Subscriber Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000</td>
<td>Connection is not established</td>
<td>A method or property has been called before establishing the connection or it has been closed</td>
</tr>
<tr>
<td>6001</td>
<td>Invalid port</td>
<td>When you start the publisher you have to provide a valid TCP port.</td>
</tr>
<tr>
<td>6002</td>
<td>Invalid timeout</td>
<td>Timeout must be a value greater than zero</td>
</tr>
<tr>
<td>6003</td>
<td>Invalid API-Key</td>
<td>API-Key cannot be an empty string</td>
</tr>
<tr>
<td>6004</td>
<td>Connection cannot be created twice</td>
<td><code>openConnection.vi</code> cannot be used on an active connection. You have to close a connection before re-open</td>
</tr>
<tr>
<td>6005</td>
<td>Connection refused from Publisher: wrong API-Key</td>
<td>Publisher refused the connection because API-Key is not correct</td>
</tr>
<tr>
<td>6006</td>
<td>Publisher not found</td>
<td>No answer from specified address: possible reasons: publisher is down or not reachable</td>
</tr>
<tr>
<td>6007</td>
<td>Publisher Address cannot be empty</td>
<td>Publisher Address cannot be an empty string</td>
</tr>
</tbody>
</table>
INDEX

A
active connection
addresses ................................................... 47
count .......................................................... 47
Analog data
Publish .................................................. 25; 26
Read ........................................................... 32
API-Key
changing ..................................................... 50
API-KEY
adding ........................................................ 50
removing .................................................... 51
Arrays ............................................................. 45
authentication successful ................................... 49
available Requests .......................................... 36
C
Communication
Close ........................................................... 35
Open ........................................................... 32
Custom data
Publish ........................................................ 53
custom XML data ............................................ 54
D
data transmission
start ............................................................ 31
stop ............................................................ 31
DHCP ............................................................... 32
Digital data
Publish ........................................................... 27
E
Error
Notifying an error ....................................... 37
error codes
Publisher ..................................................... 59
Subscriber ................................................... 60
Event
authentication successful ........................... 48
connection closed by client ......................... 48
managing .................................................... 48
timeout expired .......................................... 48
F
Files publishing and receiving ......................... 45
Filter
Server-side filtering .................................... 40
Filtering
FilterId ........................................................ 41
I
Images publishing and receiving ..................... 43
M
Messages
Receiving ..................................................... 35
P
Publisher
Creating ...................................................... 24
**methods** .................................................. 23

**properties** .................................................. 23

Publisher Class .................................................. 23

Publishing

Analog Data .................................................. 25
Custom Data ........................................... 53; 54; 55
Digital Data .................................................. 27
Request ...................................................... 33

**R**

receiving

XML ............................................................ 54

Requests

Sending ....................................................... 33

**S**

sending

XML ............................................................ 54

Source List

broadcast messages ................................... 42
Source selection ........................................... 38
filter enabling ............................................. 39
selecting .................................................... 39

start

data transmission ........................................ 31
stop

data transmission ........................................ 31

**Subscriber**

Creating .................................................. 31

methods ...................................................... 29

**properties** .................................................. 29

Read Profile ............................................... 47

Subscriber Class .............................................. 29

**T**

timeout expired .............................................. 49

**U**

User Request ................................................... 37

**W**

Welcome Kit .................................................... 26

**X**

XML ................................................................. 54

receiving ..................................................... 54
sending ....................................................... 54

63