

# Multi-Language Programming

## Part II - Extending Flash Lite with Java ME or Symbian C++

Felipe Andrade, Jukka Hämäläinen, Ugur Kaner and Mark Wilcox

Published by the Symbian Developer Network

Version: 1.0 – November 2008

<b>1</b>	<b>INTRODUCTION</b> .....	<b>2</b>
<b>2</b>	<b>FLASH LITE OVERVIEW</b> .....	<b>2</b>
<b>3</b>	<b>CONNECTING FLASH LITE AND JAVA ME</b> .....	<b>4</b>
	3.1 JAVA ME OVERVIEW.....	4
	3.2 JARPA.....	5
	3.3 EXAMPLE: USING JARPA TO ACCESS LOCATION DATA IN FLASH LITE.....	7
<b>4</b>	<b>CONNECTING FLASH LITE AND SYMBIAN C++</b> .....	<b>11</b>
	4.1 IMPLEMENTING COMMUNICATION BETWEEN FLASH LITE AND SYMBIAN C++.....	12
	4.2 KUNERILITE.....	13
	4.3 APPLICATION IMPLEMENTATION PROCESS .....	14
	4.4 PACKAGING AND SIGNING .....	15
	4.5 INSTALLATION AND SECURITY .....	16
	4.6 EXTENDING KUNERILITE WITH SYMBIAN C++.....	17
<b>5</b>	<b>ALTERNATIVES</b> .....	<b>18</b>
<b>6</b>	<b>RESOURCES</b> .....	<b>18</b>
<b>7</b>	<b>CONCLUSION</b> .....	<b>20</b>

# 1 Introduction

This paper is the second in a series examining the possibilities of building Symbian OS applications by mixing multiple languages and runtime environments. The first paper<sup>1</sup> presented an overview of the runtimes that are available and the combinations which make sense. The paper also detailed a general pattern for implementing multi-language applications.

In this paper, we show in depth two examples of solutions that combine the strengths of two different runtimes. The examples both use Flash Lite for the UI and interface with either Java ME or native Symbian C++. Luckily, we don't have to start from scratch; there are several open source and commercial frameworks which provide the necessary glue.

In the first half of this paper, Felipe explains the rationale for creating Jarpa, a Flash Lite to Java ME bridge. He details how it works as well as its limitations in terms of support for various devices and signing/security considerations. He also provides an example showing how to use Jarpa to access location data from a Flash Lite application.

Jarpa was released by Felipe Andrade and Alessandro Pace as a basic framework which packages Flash Lite content with a Java ME application. Jarpa facilitates the launching of the application and the communication between the two runtimes to enable access to additional functionality. Jarpa is released as open source under the permissive Apache License. This allows it to be used in both open source and closed source projects with very little restriction.

In the second half of this paper, Ugur Kaner and Jukka Hämäläinen of Kureri explain how to extend Flash Lite with Symbian C++ on Symbian OS in an extensible manner. They describe the KureriLite architecture and how to use it to create a hybrid application without writing any Symbian C++ code at all.

In contrast to the basic framework of Jarpa, the KureriLite Rapid Application Development toolkit provides a complete solution for developers wishing to extend the capabilities of Flash Lite with native C++ functionality. KureriLite includes a local HTTP server along with a growing set of plug-ins to access native Symbian APIs such as the accelerometer, camera, Bluetooth, GPS, file systems, etc. An emulated environment and capable test framework, built on top of the S60 (3rd Edition and later) SDKs, is also available. KureriLite is closed source and has a dual licensing model which is free for non commercial use, but charged for commercial projects.

Before going into further detail, let's have a quick overview of Flash Lite.

## 2 Flash Lite Overview

Adobe Flash Lite is a version of Adobe Flash Player targeted at mobile phones and other embedded devices. Flash Lite allows the end users of these devices to view multimedia content and applications developed using Adobe's Flash tools.<sup>2</sup>

At the time of writing there are six versions of Flash Lite available:

- Flash Lite 1.0  
Flash 4 ActionScript support.
- Flash Lite 1.1  
Flash 4 ActionScript support and additional device APIs added.

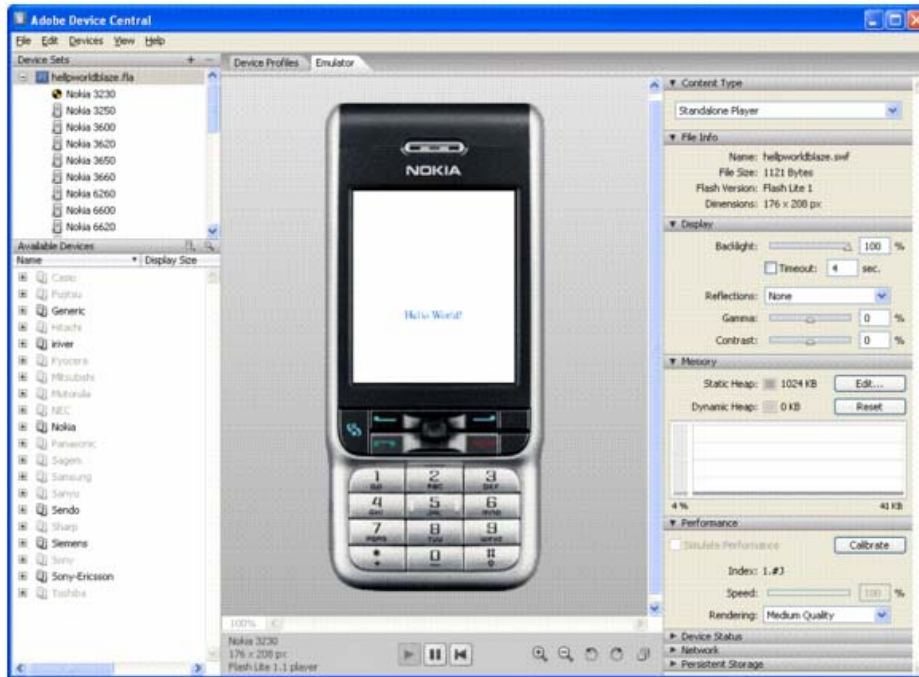
---

<sup>1</sup> The first paper can be downloaded from [developer.symbian.com/main/downloads/papers/Multi\\_Language\\_Programming\\_Part1\\_Overview.pdf](http://developer.symbian.com/main/downloads/papers/Multi_Language_Programming_Part1_Overview.pdf).

<sup>2</sup> Further details about Flash Lite and other runtimes can be found at [developer.symbian.com/runtimes](http://developer.symbian.com/runtimes).

- Flash Lite 2.0 and 2.1  
Added support for Flash 7 ActionScript 2.0 and some additional fcommand2 APIs.
- Flash Lite 3.0 and 3.1  
Added support for Flash 8 ActionScript 2.0 and also FLV video playback. Also inherits the security sandbox implementation from Flash 8, causing compatibility issues.<sup>3</sup>

Adobe Device Central is a component of the Adobe Creative Suite 3 that allows Flash developers to build, preview and debug applications for a broad range of mobile phones on the desktop before loading them on the target device. Device Central accelerates the development in much the same way as the S60 emulator, but it should not be considered a replacement for tests on a real device.



**Figure 1: Adobe Device Central**

Adobe's Flash authoring tools provide rapid application development which can cut down the development time of UIs significantly. Flash Lite can be extended by:

- using HTTP polling or the launching of an external service providing some parameters
- establishing a communication channel through XMLSocket objects.

We use the latter for the examples in this paper.

Flash Lite 2.1 and higher support the XMLSocket class that implements client sockets to allow the device running the Flash Player to communicate with a local server identified by an IP address and a port. All XMLSocket messages are based on XML as a data format for messages in both directions. An unlimited number of XML messages can be sent and received over a single and persistent XMLSocket connection.

<sup>3</sup> See the following wiki page for more details:

[wiki.forum.nokia.com/index.php/KIC000917\\_-\\_Security\\_sandbox\\_issue\\_in\\_Flash\\_Lite\\_may\\_prevent\\_network\\_access](http://wiki.forum.nokia.com/index.php/KIC000917_-_Security_sandbox_issue_in_Flash_Lite_may_prevent_network_access).

### 3 Connecting Flash Lite and Java ME

Smartphones that support both Java ME and Flash Lite have become more common in the last few years. This has opened the door for rich applications that combine the best of both technologies.

In this paper we aim to provide the basic knowledge required to start distributing Flash Lite content with optional Java ME extensions. This includes:

- The use of Java Archive Packaging.
- Making content appear as Java ME applications in the smartphone's menu. This combines the benefits of Flash Lite's fast development and Java ME's rich APIs to produce high quality content for mobile devices.

A smartphone supporting Java ME offers a simple procedure to allow the end user to install after-market applications, which can then be run from the smartphone's application menu. The same is not true for Adobe's Flash Lite; the user must first launch the Flash Lite application from the menu and then locate the desired Flash application from the file system.

The Flash Lite runtime is very restrictive. It does not allow access to features such as Bluetooth or the camera. In fact, it offers fewer additional local capabilities than a web application running in the web browser.

The Jarpa project was created in order to address both the usability issue of installation and the technical limitation of the restrictive runtime.

#### 3.1 Java ME Overview

Java 2 Platform, Micro Edition (Java ME) is a variant of the Java platform that is targeted at small and resource-constrained devices. Until now, Java ME has been considered the de facto standard technology for mobile development due to the maturity of the platform and the sheer number of Java ME-capable devices on the market.

The Java ME technology is based on configurations, profiles and optional packages.

A configuration provides the basic set of libraries and virtual machine capabilities. There are two configurations available: the Connected Limited Device Configuration (CLDC) targeted at devices with limited resources, and the Connected Device Configuration (CDC) for more capable devices. S60 devices support the CLDC, although the current Nokia implementation far exceeds the minimum specifications for the configuration (in terms of memory available and the use of the CLDC Hotspot Implementation JVM, rather than the more basic KVM). UIQ 3 devices onwards support the CDC.

On top of the different configurations, the Java ME platform also specifies a number of profiles defining a set of high-level APIs. The most common profile is the Mobile Information Device Profile (MIDP). It is layered on top of CLDC and adds APIs for application management (install, run, uninstall), user interface, networking and persistent storage.

A complete definition of a Java ME environment includes both a configuration and a profile. In addition, optional packages offer further capabilities which may or may not be present on a device; for instance, the location-based services package and Bluetooth package only make sense on devices that support the relevant hardware.

Unlike Flash, Java ME does not provide a consistent design tool that allows graphic designers and software developers to interact. In many cases, the cost of porting graphics and code for an application to a new platform can be very significant.

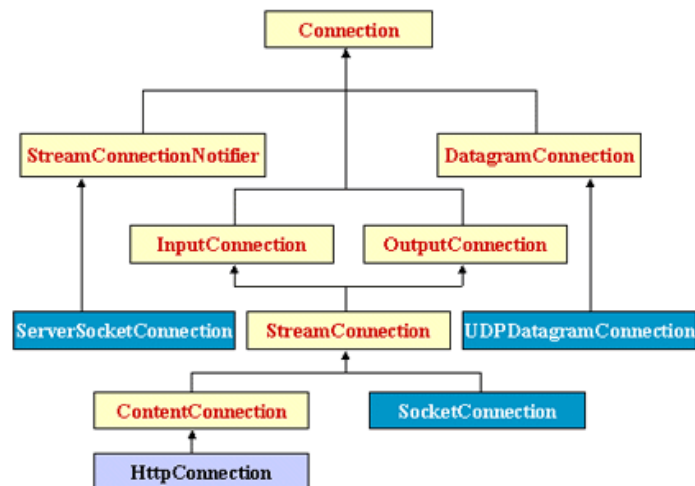
### 3.2 Jarpa

The Jarpa framework was originally developed to provide a convenient solution for packaging Flash Lite applications in a JAR file, a compressed file that contains all of the classes and assets required to run a Java application file. The concept was extended to include a communication mechanism to enable access to Java ME APIs from Flash Lite. Hence the two main scenarios for using Jarpa are to:

- package Flash Lite content (games, applications, etc.) in a JAR to use the Java ME distribution mechanism
- use Flash Lite as the presentation layer while Java ME is used to provide access to services like file handlers, Bluetooth and location.

The Jarpa Framework exposes many Java features and packages (including some notable JSRs<sup>4</sup>) to provide developers access to the rich set of Java APIs from Flash Lite. Jarpa only works in a restricted number of smartphones and cannot be considered fully portable.

Jarpa uses sockets as a bridge between Java ME and Flash Lite. The MIDP 2.0 specification (JSR 118) added support for sockets and datagrams; this support is optional and based on the Generic Connection Framework (GCF) of the CLDC and is the feature essential to exchanging data between these runtime environments.



**Figure 2: The Interface Hierarchy of the Generic Connection Framework**

By providing a local socket connection, Jarpa creates a communication channel so that Flash Lite and Java ME can talk with each other. The following code creates a local Java ME socket server defined on port 9002.

```
(ServerSocketConnection) Connector.open("socket://: 9002");
```

<sup>4</sup> Java Specification Requests – the system used to standardize new interfaces.

### 3.2.1 How Jarpa Works

Jarpa wraps Flash Lite content inside a JAR and, using Java's APIs, copies the file to the mobile phone's file system, thereby invoking the platform's Application Management System (AMS) to launch the content. (AMS is the software in the device that manages the download, installation, execution and removal of applications and other resources on the device.)

The JSR 75 or PDA optional packages give developers access to the FileConnection API that lets the Jarpa Framework copy Flash Lite resources to the smartphone's file system. (There are some security issues related to this that will be covered below.)

The following code invokes the AMS to handle the launch of the native Flash Player.<sup>5</sup>

```
platformRequest("file:///E:/Others/trusted/Jarpa_105.swf");
```

The complete application start-up can be seen in Figure 3.

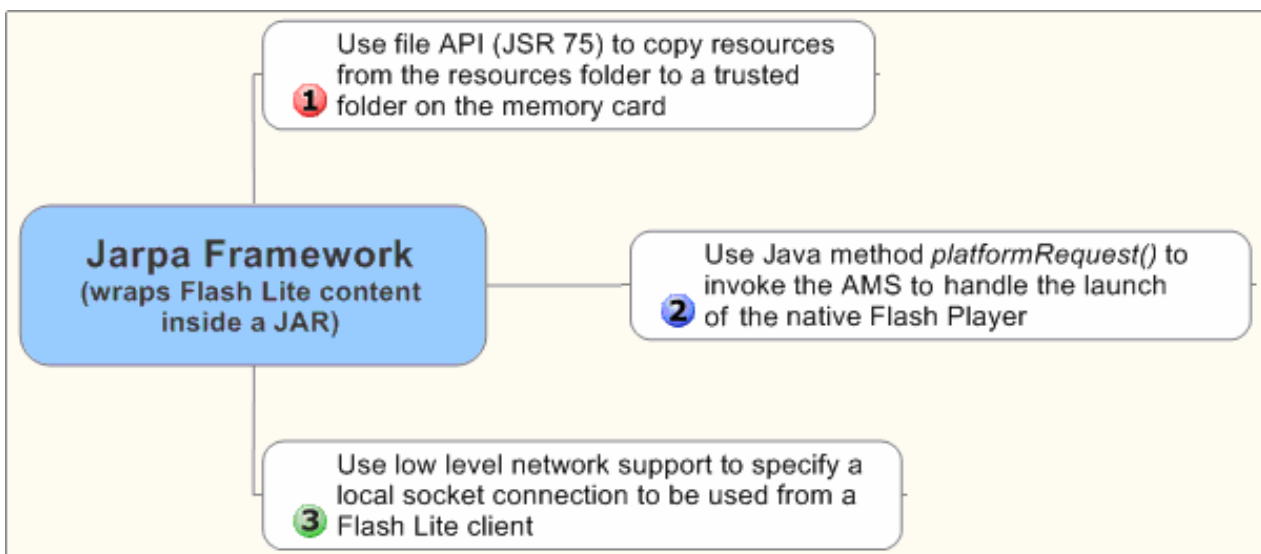


Figure 3: Jarpa start-up flow

### 3.2.2 Security Sandbox and Software Limitations

Jarpa is not supported on all devices. It's supported on some MIDP 2.0 devices but other devices are more restricted or omit key optional packages that are required for Jarpa to function.

Obviously JSR 75 (PDA optional package) must be supported on the device or Jarpa cannot work at all. In addition to this, current Nokia Series 40 devices will have a very restricted use of the framework (basically packaging only) since Jarpa needs a MIDlet running in the background (this feature isn't introduced into the standard until MIDP 3.0).

Flash Lite 3 introduced a new security model which is not compatible with previous versions of the Flash Lite Player. If Flash Lite content published for Flash Lite 2.x is run and needs to connect to the Internet, the application will fail to make the connection unless it's updated with the tool

<sup>5</sup> The `platformRequest()` method is used to invoke external platform services. It takes a single URL as an argument. In this case, we use the 'file' scheme which causes the AMS to launch the specified file with its default player/handler.

provided by Adobe.<sup>6</sup> Additionally, Flash Lite content cannot access both a local server and a remote one unless it is running from a 'trusted' content folder. This folder is either C: /data/others/trusted for content in the device memory, or E: /Others/trusted for content on the memory card.

If using a non-signed MIDlet, the system will ask for the end user's permission every time that restricted APIs are used. In the example presented below, you will be prompted with many security messages to read and write files and access location-based services.

### 3.2.2.1 How to test on a real device

There are many reasons why you need to test your mobile applications on the target smartphones, and often purchasing all of the target devices is not feasible. We will briefly introduce some of the solutions currently available in the market.

#### Nokia Remote Device Access (RDA)

([www.forum.nokia.com/main/technical\\_services/testing/rda\\_instructions.html](http://www.forum.nokia.com/main/technical_services/testing/rda_instructions.html))

The remote device access (RDA) service provides access to a live, Symbian OS-based device over the Internet. Nokia RDA allows you to control, install and run applications in real time, reducing the effort for testing and application development. This service is provided by Nokia free of charge.

#### DeviceAnywhere

([www.deviceanywhere.com/nokia/welcome.htm](http://www.deviceanywhere.com/nokia/welcome.htm))

DeviceAnywhere is a third-party service that enables automated tests to be run on over 1000 real devices in live global networks. The service supports many S60 and Series 40 devices and so offers comprehensive real world testing. This is a commercial service and fees are payable based on the devices accessed and the amount of time used.

### 3.2.3 Jarpa Packaging Tool (JPT)

With the Jarpa Packaging Tool, Flash developers can benefit from the ability to seamlessly package Flash content into a JAR. If you have no previous experience with Java ME, then there is a tutorial for packaging with Jarpa written by Thomas Joos, which is available at [vilebody.wordpress.com/category/jarpa/](http://vilebody.wordpress.com/category/jarpa/).

You can follow the Jarpa project page, at [code.google.com/p/jarpa](http://code.google.com/p/jarpa), to discover new additions.

## 3.3 Example: Using Jarpa to Access Location Data in Flash Lite

The following example is used to connect Flash Lite and Java ME through socket connections. The first thing to do is set up your environment. There are two articles on the Forum Nokia wiki that cover this process: 'Installing Java ME development tools for S60' ([wiki.forum.nokia.com/index.php/Installing\\_Java\\_ME\\_development\\_tools\\_for\\_S60](http://wiki.forum.nokia.com/index.php/Installing_Java_ME_development_tools_for_S60)) or 'Getting started with Java ME' ([wiki.forum.nokia.com/index.php/Getting\\_started\\_with\\_Java\\_ME](http://wiki.forum.nokia.com/index.php/Getting_started_with_Java_ME)).

You can find the complete source code for this example on the Jarpa code page ([code.google.com/p/jarpa](http://code.google.com/p/jarpa)). The following piece of code is part of the application MIDlet and is used to copy, launch and exchange messages between Flash Lite and Java ME.

---

<sup>6</sup> You can download the updater tool from: [www.adobe.com/support/flashplayer/downloads.html#cu](http://www.adobe.com/support/flashplayer/downloads.html#cu).

```

private String _targetFolder = "file:///E:/Others/trusted/Jarpa_105.swf" ;

public void run() {
    try {
        Resources.getInstance().copyResources(_targetFolder);
        platformRequest(_targetFolder);
    } catch(Exception e) {
        ...
    }

    while(true) {
        try {
            Connection.getInstance().write("[Status] Loading GPS data");

            Ip = LocationProvider.getInstance(null);
            Location = Ip.getLocation(MAX_GPS_TIMEOUT);
            Coordinates coordinates = Location.getQualifiedCoordinates();

            msg = "[RESULTS] " + getHour() + " - " + getDate() + "-";
            msg += "Latitude: " + coordinates.getLatitude() + "-";
            msg += "Longitude: " + coordinates.getLongitude() + "-";
            msg += "Altitude: " + coordinates.getAltitude();

            // send a message to Flash Lite
            Connection.getInstance().write(msg);

            // read any data from Flash Lite
            recv = Connection.getInstance().read();

            Thread.sleep(MAX_INTERVAL);
        } catch (LocationException e) {
            // In case of error write a message back to the Flash client
            try {
                Connection.getInstance().write("[Status] GPS request timed out");
                Thread.sleep(SHORT_INTERVAL);
            } catch(Exception ex) {
                ...
            }
        } catch (...) {
        }
    }
}
}

```

As mentioned previously, Flash Lite 3 introduced a new security restriction, so if you need local and network access you have to create a sub-directory of the Flash Lite content directory called 'Trusted.' The code below manages this, creating the folder and putting the content there if necessary.

```

public void copyResources(String aFolder) {
    _toFolder = aFolder;
    int index = aFolder.lastIndexOf('/');
    String fileName = aFolder.substring(index + 1, aFolder.length());

    _contentFolder = aFolder.substring(0, index);
    _fromFolder = "/res/" + (fileName);
}

```

```

try {
    // get resources from the application directory
    this._incomingData =
        this.getClass().getResourceAsStream(this._fromFolder);

    FileConnection fConn = (FileConnection)Connector.open(this._toFolder);
    FileConnection fContent =
        (FileConnection)Connector.open(this._contentFolder);

    if(!fContent.exists()){
        fContent.mkdir();
    }

    fContent.close();

    // if not exists creates the file
    if (!fConn.exists()) {
        fConn.create();

        int ch = 0;
        DataOutputStream dataStorage = fConn.openDataOutputStream();

        // write to the file
        while ((ch = this._incomingData.read()) > -1) {
            dataStorage.write(ch);
        }

        // closes
        dataStorage.flush();
        dataStorage.close();

        if(this._incomingData != null)
            this._incomingData.close();
    }

    fConn.close();
} catch (Exception e) {

}
}

```

The Java code below is responsible for creating a local socket server, then reading and writing messages to the connected Flash Lite client.

```

public String read() {
    StringBuffer recv = new StringBuffer();

    synchronized (this) {
        if(_client != null) {
            try {
                int ch;
                if(_in.available() > 0) {
                    while((ch = _in.read()) != '\0') {
                        recv.append( (char) ch );
                    }
                }
            }
        }
    }
}

```

```

        }
        } catch (Exception e) {
            _status = "read: " + e.toString();
        }
    }
}

return recv.toString();
}

public void close() {
    try {
        _server.close();
        _client.close();
    } catch (IOException e) {
        ...
    }
}

public void run() {
    try {
        // start a local socket on port 9002
        _server = (ServerSocketConnection) Connector.open("socket://:9002");
        _client = null;
    } catch (Exception e) {
        ...
    }

    try {
        // wait for incoming connections
        _client = (SocketConnection) _server.acceptAndOpen();
        _client.setSocketOption(SocketConnection.DELAY, 0);
        _client.setSocketOption(SocketConnection.KEEPALIVE, 2);

        _out = _client.openOutputStream();
        _in = _client.openInputStream();
    } catch (Exception e) {
        ...
    }
}
}

```

The following ActionScript class handles socket connections and the related events.

```

import mx.utils.Delegate;
import mx.events.EventDispatcher;

class com.i2tecnologia.jarpa.Jarpa extends EventDispatcher {
    private var host_str: String;
    private var port_num: Number;
    private var jarpa_xmls: XMLSocket;
    private var isConnected_bool: Boolean = false;

    /**
     * Jarpa(host_str: String, port_num: Number)
     * @host_str: server host
    */
}

```

```

        @port_num: port number ( > 1024 )
    */
    public function Jarpa(host_str: String, port_num: Number) {
        this.host_str = host_str;
        this.port_num = port_num;
        this.jarpa_xml_s = new XMLSocket();
    }

    public function connect():Void {
        this.jarpa_xml_s.connect(this.host_str, this.port_num);
        this.jarpa_xml_s.onConnect = Delegate.create(this, onConnect);
        this.jarpa_xml_s.onClose = Delegate.create(this, onClose);
    }

    public function onConnect(success_bool: Boolean):Void {
        setStatus(true);
        dispatchEvent({type: "onJarpaConnect", status: success_bool});
        stateReading();
    }

    public function write(data_str: String):Void {
        this.jarpa_xml_s.send(data_str);
    }

    public function onReadData(data_str: String):Void {
        dispatchEvent({type: "onReadData", data: data_str});
    }

    public function onClose():Void {
        setStatus(false);
        dispatchEvent({type: "onJarpaClose"})
    }

    public function stateReading():Void {
        this.jarpa_xml_s.onData = Delegate.create(this, onReadData);
    }

    ...
}

```

Having demonstrated the Jarpa framework, we now move on to consider how we can extend Flash Lite applications with Symbian C++ rather than Java ME.

## 4 Connecting Flash Lite and Symbian C++

The combination of Flash Lite and Symbian C++ is excellent for creating great-looking applications with access to all native features and maximum performance on Symbian OS. Compared to Java and Python, using Symbian C++ has the advantages of high performance, smaller size in executable files, wider access to the system and no pre-installation requirements.

One can argue that Symbian C++ is not the easiest dialect to learn, offsetting the benefits we just described. For many tasks, Flash Lite provides much more rapid development. For Flash Lite developers with little or no knowledge of Symbian C++, it would be very useful to be able to access the additional APIs without having to invest in learning a relatively low-level language.

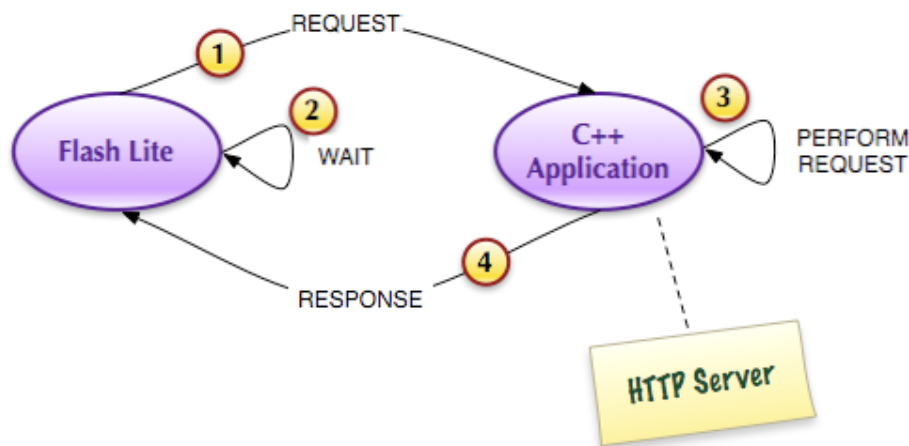
KuneriLite Rapid Application Development toolkit for Flash Lite provides an answer but, before getting into further details, let's analyze the anatomy of the Flash Lite and Symbian C++ combination.

#### 4.1 Implementing Communication between Flash Lite and Symbian C++

The model for extending Flash Lite with Symbian C++ is simple: a Symbian C++ component acts as a local server. Flash Lite sends a request to the server, it performs the action and returns the results back to Flash Lite.

Currently, the best way to communicate between Flash Lite and the outer world is a local connection to the Symbian C++ server application with a simple HTTP protocol implementation.

It's perhaps not the most efficient way of extending the runtime, but it's the only feasible approach available to third-party developers. Though not ideal, it works reasonably well since the overhead of the local connection is small and doesn't affect the performance drastically.



**Figure 4: Using a local HTTP server to communicate with a Flash Lite application**

In the past, there have been several attempts to create a solution to augment Flash Lite, for instance, Leonardo Risuleo's 'WriteFile' and Spaghettisort's 'Janus' product (for which the Symbian version is also developed by Risuleo).<sup>7</sup> While these are good initiatives, they have not progressed rapidly. WriteFile does exactly what it says and can only offer basic reading and writing of text files, which is useful for saving settings or game progress. Although Janus offers fewer features than KuneriLite at present, it provides basic services and, being open source, it can be modified and extended as needed.

Currently, KuneriLite is the only project in active development which offers a stable, working solution. As our implementation details depend heavily on the framework used we will focus on KuneriLite while explaining how to extend Flash Lite on Symbian OS.

<sup>7</sup> Janus is now available as open source at [www.janus-flash.com](http://www.janus-flash.com). See Section 5 for further information about the development of Janus.

## 4.2 KunerLite

### 4.2.1 Introducing KunerLite

KunerLite is a Rapid Application Development toolkit for Flash Lite developers, which provides a basic Symbian C++ extension mechanism. It allows developers to escape the Flash Lite sandbox to create enterprise level S60 (3rd Edition and later) applications with a great-looking user interface. KunerLite brings a rich set of native Symbian features to Flash Lite, and aids the development of hybrid Flash Lite/Symbian applications without writing a single line of Symbian C++ code. It also provides a simple-to-use PC interface to package and deploy projects (as SIS files). Applications created using KunerLite do not require any other SIS files or runtimes to be installed; they launch out of the box.

Although KunerLite supports any runtime that can open a local socket, it focuses on Flash Lite 1.1, as this has by far the largest installed user base (the original S60 3rd Edition devices shipped with Flash Lite 1.1 – often these can have firmware updates that support later Flash Lite versions but in order to guarantee compatibility it is necessary to target the lowest common denominator).

### 4.2.2 How Does KunerLite Work?

KunerLite utilizes the same logic and communication principles stated above. Commands are called from Flash Lite and passed to the KunerLite local HTTP server, which is launched at the same time as the Flash Lite Player. The KunerLite server passes the command to the right plug-in, which then performs the desired action and returns the results back to Flash Lite.

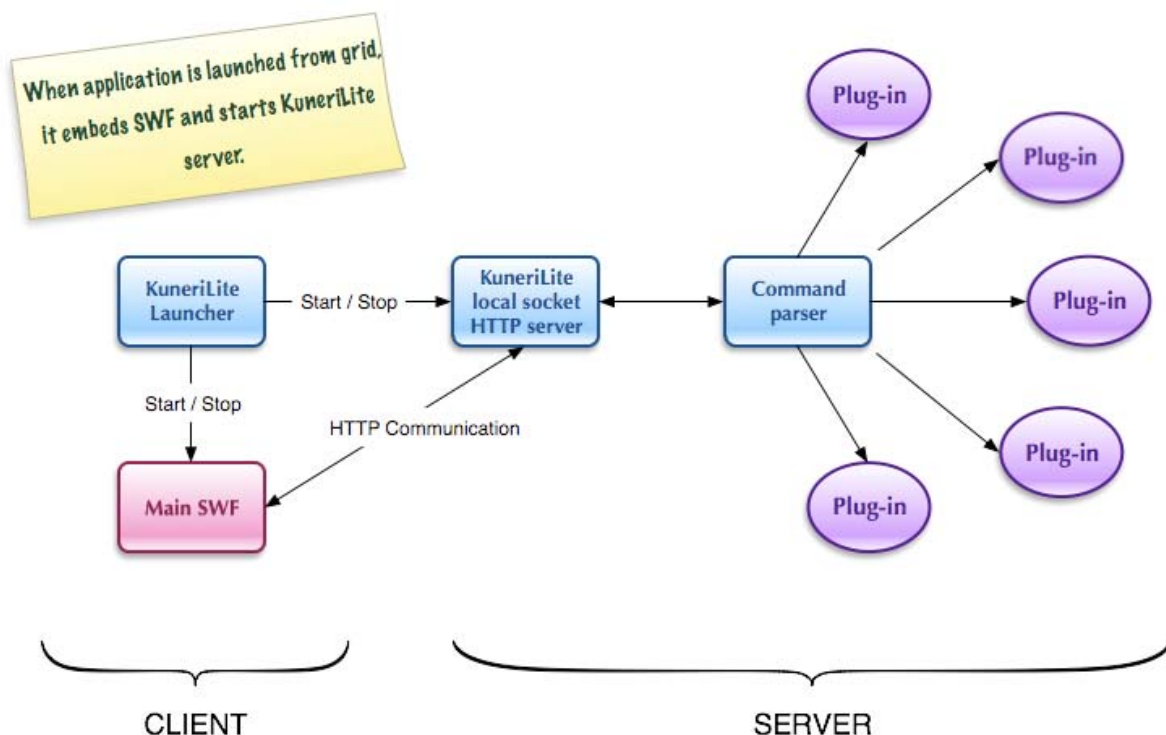


Figure 5: The KunerLite architecture

KuneriLite's plug-in architecture enables developers to select any combination of functionality depending on the features present on a smartphone. This not only helps the system to be modular, but also generates optimal results in terms of memory and package file size.

The KuneriLite localhost server is started along with the Flash Lite application, and the Flash Lite file (SWF) can communicate with KuneriLite via the `LoadVariables()` command in ActionScript. KuneriLite also supports any number of applications running at the same time, i.e., it supports multiple application instances as well installations. This is achieved by using separate UIDs for the server application and plug-ins in each application to avoid clashes. A unique set of plug-ins must be requested from Kuneri for this to work.

An alternative method to `LoadVariables()` would be the `XMLSocket` class, but since it is not available in Flash Lite 1.x, it is not supported by KuneriLite to maintain application compatibility with all available players.

For launching external applications, Flash Lite has a `FSCommand("Launch", "arguments")` command, but it is not suitable for two-way communication and therefore is not usable for most use cases.

The KuneriLite plug-in architecture relies extensively on the Symbian client-server architecture and dynamic library (DLL) plug-ins. The usage of resources is effective and safe: when an application is not running, all resources are freed, except in certain conditions where a plug-in is working as a long-running background task (for example, the timer). Background tasks can be closed later exclusively by the application. This is an area where care has to be taken with this kind of hybrid architecture because it is possible to close the Flash Lite Player and leave the background server running and consuming resources. Similarly, it is possible to close the server application while it's servicing the UI, potentially causing a panic.

To test the functionality of the application on the desktop, KuneriLite works with the standard Windows-based emulator from the S60 SDK to provide an emulated environment for testing. The Symbian C++ code can be accessed via HTTP from the SWF in the same way as on the device. Most functionality can be tested on the emulator; however, some plug-ins will not work. For example, proper testing of the camera, accelerometer or GPS functionality requires a target device since the necessary hardware and software support is not available in the emulator.

Since platform security was first implemented on Symbian OS, all Symbian applications need to be signed. By default, applications created with KuneriLite are signed with a self-generated certificate, but some features (for example, GPS and Ringtone plug-ins) require signing with a certificate from Symbian Signed, even for testing because they use sensitive capabilities. These capabilities are detailed on the KuneriLite wiki pages, at [wiki.kunerilite.net/index.php?title=KuneriLite\\_Plug-ins](http://wiki.kunerilite.net/index.php?title=KuneriLite_Plug-ins). Plug-ins do not need signing, but applications using these particular plug-ins do need to be signed, according to the platform security requirements of Symbian OS.

In order to test these features, signing with the Open Signed process, either Online or Offline, is a good option, whereas for application releases, the Express Signed procedure can be followed as a more convenient practice than Certified Signed, given that restricted capabilities are not used.

### **4.3 Application Implementation Process**

The application implementation process does not require the direct use of any Symbian tool or knowledge. Everything is done with Flash and the Flash IDE, which basically means that developers only need basic Flash skills to develop a Symbian application. In order to access extra features, KuneriLite commands are used. These commands have a very simple logic and are not difficult to implement. The basic principle is: **Call** (the command), **Wait** (for the result from KuneriLite server) and **React** (to the result).

For instance, if we want to create a simple application to read accelerometer values from the device, we should first **call** the following command from Flash Lite:

```
_root.klEnd = -99;
_root.klError = -99;
loadVariables("http://127.0.0.1:1001/Basic/accelerometer?klCommand=readsensor",
             _root);
```

This will reset the KureriLite variables and then send the command.

After sending this, Flash Lite should **wait** for the answer and **react** according to the response received from KureriLite, as follows:

```
myObj = new Object();
myObj.KLExec = function(message){
    if (_root.klEnd == -99) {
        // Response not received. Display waiting message
    } else {
        // We have a response
        clearInterval(ID);
        if (_root.klError == 0) {
            // Sensor started successfully. Display axis values
            trace(_root.klXAxis + ", ");
        } else {
            // Error reading sensor data. Display error
        }
    }
}
ID = setInterval(myObj, "KLExec", 100);
```

**Note:** This polling sequence is mandatory because there can only be one `loadVariables()` request at a time. Although Flash Lite allows sequential `loadVariables()` calls, they can only be served one at a time on Symbian OS.

## 4.4 Packaging and Signing

Once the Flash Lite application is complete, the project should be packed into SIS packages. SIS packages allow end users to download and install Flash Lite applications as with any other S60 application. The KureriLite PC Interface provides a very easy way to prepare such packages. You can find a detailed tutorial about the packaging process on the KureriLite wiki.<sup>8</sup>

For Flash Lite developers, Symbian's **Express Signed** procedure is the preferred way to sign applications for distribution.

For testing purposes, Symbian's **Open Signed** procedure(s) can be followed to create IMEI-locked SIS packages without cost for a single IMEI (or multiple IMEIs with an **annual Publisher ID**, which does incur a fee).

For the Express Signed and Open Signed Offline signing options, purchasing a Publisher ID is required, and then these certificate files must be converted.<sup>9</sup> Having completed their conversion:

<sup>8</sup> See the following wiki article for details:  
[wiki.kurerilite.net/index.php?title=KureriLite\\_Wizard\\_Beginner%27s\\_Guide](http://wiki.kurerilite.net/index.php?title=KureriLite_Wizard_Beginner%27s_Guide).

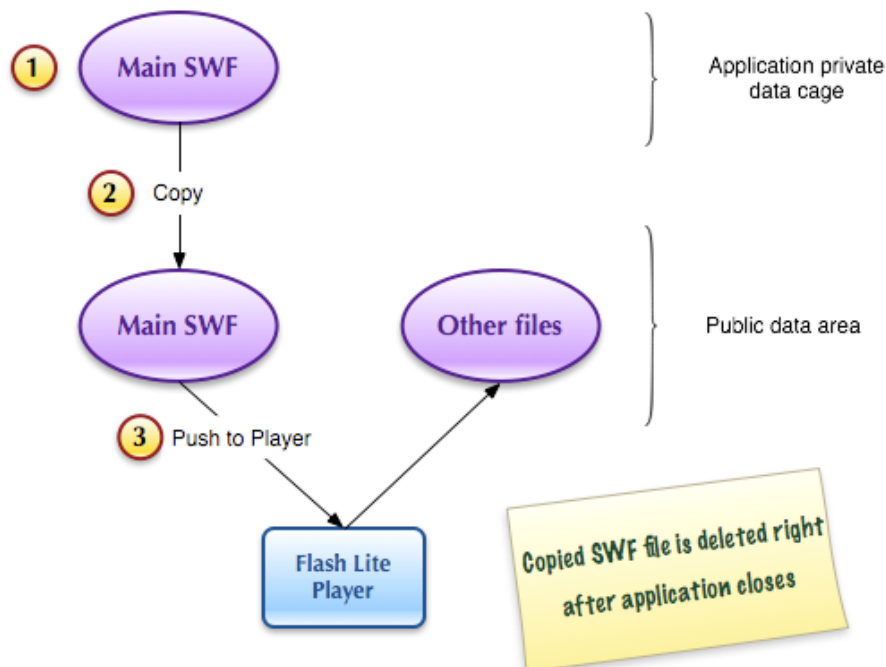
<sup>9</sup> See the following wiki article for details:  
[wiki.forum.nokia.com/index.php/Using\\_TC\\_TrustCenter\\_Publisher\\_ID](http://wiki.forum.nokia.com/index.php/Using_TC_TrustCenter_Publisher_ID).

- the resultant .key and .cer files can be entered into KunerLite to sign the SIS file for the Express Signed procedure (the generated SIS file can be sent to [www.symbiansigned.com](http://www.symbiansigned.com))
- certificate files can be used for Offline signing as well (once again, needed only for testing if plug-ins require signing).

## 4.5 Installation and Security

All Symbian applications must conform to the platform security requirements of Symbian OS,<sup>10</sup> which applies also for Flash Lite applications using KunerLite or any other third-party C++ or Python-based framework.

All application data files should be located in the private data cage area of the file system. However, the main SWF file for Flash Lite cannot be there because the Flash Lite player is unable to access files other than its own. This issue is handled either by keeping Flash Lite files in a public directory, or with other methods such as KunerLite Simple Protection (see Figure 6).



**Figure 6: KunerLite Simple Protection mechanism**

The biggest and most concrete risk for a developer is still the unpacking of unsigned SIS files (via third-party tools) and copying all of the content. However, files cannot be re-placed into a new SIS file if the original is signed with a Symbian Signed Developer Certificate since it is not allowed for a developer to sign binaries that use another developer's UIDs. As a result, KunerLite plug-ins must be licensed versions, where the UIDs used by KunerLite plug-ins all belong to the developer.

<sup>10</sup> More information about platform security on Symbian OS can be found in *Symbian OS Platform Security: Software Development Using the Symbian OS Security Architecture*, the Symbian Press book written by Craig Heath ([developer.symbian.com/main/documentation/books/books\\_files/sops/index.jsp](http://developer.symbian.com/main/documentation/books/books_files/sops/index.jsp)).

Another security issue for Flash Lite and Symbian C++ applications is HTTP communication. Since HTTP is used, all common HTTP security issues apply. The following are some common security issues and their solutions:

- **Eavesdropping and malicious HTTP attacks.**  
KunerLite reserves an HTTP Socket port and the operating system will not allow the use of the same socket port by two applications.
- **Replacement and hacking of application executables on the device.**  
Symbian OS avoids this problem with data caging.
- **Listening to or issuing HTTP requests from another Flash Lite (SWF) file.**  
KunerLite blocks that by using certain license keys in `LoadVariables()` calls and also by allowing connections only from the Flash Lite Player.

These security measures make it difficult to reproduce or exploit your application, although they wouldn't stop a determined hacker. If you have high value content it will need additional protection.

#### 4.5.1 Flash Lite Security Sandbox

The Flash Lite security sandbox is implemented for Flash Lite Player 2 and above. Although this isn't a significant problem for 2.x players (as only a few `fscommand` calls are affected), it can be for 3.x players.

Flash Lite Player 3.x denies all HTTP calls to localhost (127.0.0.1) by default, which makes communication with the local server impossible. Thankfully, Adobe and Nokia have provided a workaround, which involves installing the application in a 'trusted' folder, as described in Section 3.2.2.

#### 4.6 Extending KunerLite with Symbian C++

Currently it is not possible for third parties to write their own KunerLite plug-ins due to various commercial and security issues. Instead, an alternative mechanism for extending KunerLite is provided: KunerLite supports the launching of any external Symbian application (via its System plug-in).

If a developer implements an application with Symbian C++, the system plug-in commands `open` and `openname` can be used to launch it (only the `openname` command supports non-GUI applications).

Flash Lite can then communicate with the launched application using the KunerLite File plug-in.

##### An example of the communication procedure:

- 1 The application writes its desired custom command into a separate file using the File plug-in's `write` command, which also creates the file if it doesn't exist.
- 2 Flash Lite calls the System plug-in's `open` or `openname` command to launch the custom application.
- 3 The launched custom application reads the desired command from the file and executes it.
- 4 The custom application writes a variable (for example, `customVar=12&`) as a response into a file and closes.

- 5 The Flash Lite application can check when the response file exists using the File plug-in's `exists` command. (It is important not to use the file directly as an argument at this point because the player will give an error note if file doesn't exist yet.)
- 6 The response can be read from the file using the `LoadVariables()` method with the file as an argument.
- 7 The application can delete the files using the File plug-in's `delete` command.

Most use cases can be implemented using this method.

## 5 Alternatives

At the time of writing this paper, one of the alternative frameworks for extending Flash Lite with Symbian C++ mentioned in Section 4.1, Janus, has just been released as an open source project.<sup>11</sup> The functionality offered is not as comprehensive as that available in KureriLite, there are currently only six native functions supported, and the product has not evolved for several months. However, the open source nature of the project means that you can extend and customize it for your own purposes. If you have a use case that goes beyond the existing plug-ins and extension mechanisms for KureriLite, or you need to target versions of S60 before 3rd Edition, then it is worth exploring further. The Janus Symbian Engine is available under either the permissive BSD or GNU LGPL licenses, so it should be usable in almost any project. If the implementation or licensing is not suitable for both KureriLite and Janus, the only remaining option for extending Flash Lite with Symbian C++ at present is to create your own solution. Fortunately, many of the benefits of native Symbian C++ extensions are possible using other languages as well.

Looking for an easy application start-up method and a way to extend their Flash applications, developers have combined Flash Lite with several other mobile languages. This concept was first introduced in the Flyer Framework project that brought the power of Python for S60 and Flash Lite to the mobile ecosystem. Those wanting a Flash UI and rapid application development with an easier learning curve for extensions might consider Python for S60 rather than Symbian C++. That combination will be discussed in the next paper in this series.<sup>12</sup> However, Python for S60 is, as the name suggests, limited to S60 devices. The Jarpa project described above was created to provide a solution supporting a wider range of devices. However, if your requirements are not satisfied by the frameworks mentioned so far then another option, SWF2Go by Faisal Iqbal ([www.swf2go.com/](http://www.swf2go.com/)), has been released during the writing of this paper. It has support for extending Flash Lite with either Python for S60 or C#, via the Net60 runtime developed by Red Five Labs ([www.redfive.com/](http://www.redfive.com/)).

## 6 Resources

Further information about extending Flash Lite with Symbian C++ is available:

- Flash Viewer Framework API  
[www.forum.nokia.com/document/Cpp\\_Developers\\_Library/GUID-96C272CA-2BED-4352-AE7C-E692B193EC06/html/Flash\\_Viewer\\_Framework\\_API4.html](http://www.forum.nokia.com/document/Cpp_Developers_Library/GUID-96C272CA-2BED-4352-AE7C-E692B193EC06/html/Flash_Viewer_Framework_API4.html)

---

<sup>11</sup> See [www.janus-flash.com](http://www.janus-flash.com) for details.

<sup>12</sup> The third paper can be downloaded from [developer.symbian.com/main/downloads/papers/Multi\\_Language\\_Programming\\_Part3.pdf](http://developer.symbian.com/main/downloads/papers/Multi_Language_Programming_Part3.pdf).

- How to make Flash launcher with Symbian C++  
[wiki.forum.nokia.com/index.php/How\\_to\\_make\\_Flash\\_launcher\\_with\\_Symbian\\_C++](http://wiki.forum.nokia.com/index.php/How_to_make_Flash_launcher_with_Symbian_C++)
- How to Prototype Applications with Flash (see 'Extending the capabilities of Flash')  
[www.forum.nokia.com/info/sw.nokia.com/id/1cc39e54-2309-483c-9d0e-a04fab9740f3/S60\\_Platform\\_How\\_to\\_Prototype\\_Applications\\_with\\_Flash\\_v1\\_0\\_en.pdf.html](http://www.forum.nokia.com/info/sw.nokia.com/id/1cc39e54-2309-483c-9d0e-a04fab9740f3/S60_Platform_How_to_Prototype_Applications_with_Flash_v1_0_en.pdf.html)

KuneriLite-related resources on the Internet are found at:

- [www.kuneri.net](http://www.kuneri.net)
- [wiki.kuneri.net](http://wiki.kuneri.net)
- [forum.kuneri.net](http://forum.kuneri.net)

The following are links to complete tutorials by third-party developers who have created applications with KuneriLite:

- Tutorial: FliKun - Flickr photo uploader  
[www.jappit.com/blog/2008/06/25/flikun-kuneri-lite-based-flashlite-photo-uploader-for-flickr/](http://www.jappit.com/blog/2008/06/25/flikun-kuneri-lite-based-flashlite-photo-uploader-for-flickr/)
- Tutorial: Instructions to display GPS position using Google Maps  
[wiki.forum.nokia.com/index.php/Displaying\\_GPS\\_position\\_using\\_Google\\_Maps\\_images\\_in\\_FlashLite](http://wiki.forum.nokia.com/index.php/Displaying_GPS_position_using_Google_Maps_images_in_FlashLite)
- Tutorial: How to create a Bluetooth chat application  
[wiki.forum.nokia.com/index.php/How\\_to\\_create\\_a\\_chat\\_by\\_Bluetooth\\_using\\_Kuneri\\_Lite](http://wiki.forum.nokia.com/index.php/How_to_create_a_chat_by_Bluetooth_using_Kuneri_Lite)
- Tutorial: How to create your first Flash Lite ringing tone with KuneriLite  
[www.jappit.com/blog/2008/10/08/create-your-first-flash-lite-ringtone-with-kuneri-lite/](http://www.jappit.com/blog/2008/10/08/create-your-first-flash-lite-ringtone-with-kuneri-lite/)

KISS60 ([www.kiss60.com/](http://www.kiss60.com/)) is also a great example of how Flash Lite can provide a great look and feel as an S60 application.

Figure 7 shows some screenshots from an application created with KuneriLite as an example of the rich interfaces you can create with Flash Lite.

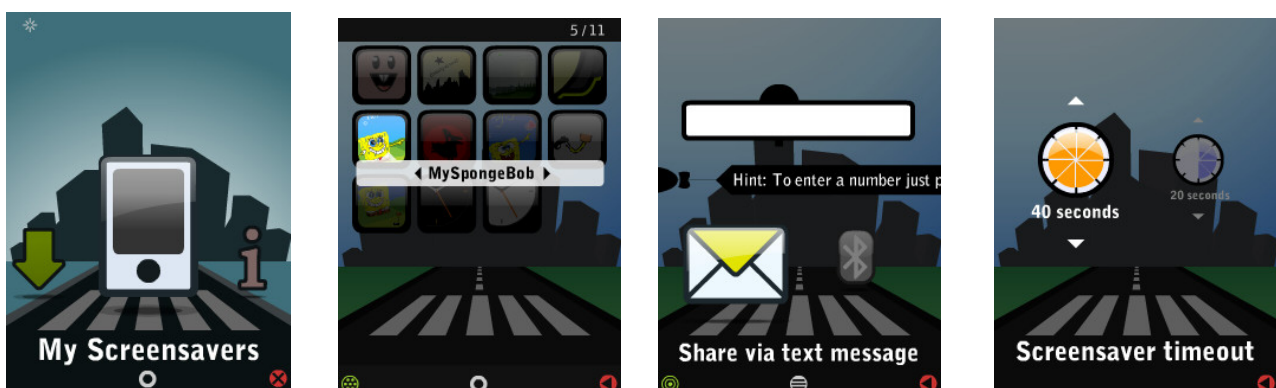


Figure 7: Screenshots of a KuneriLite application

WriteFile from Leonardo Risuleo is available from his blog: [www.scriptamamentgroup.net/byte/?p=144](http://www.scriptamamentgroup.net/byte/?p=144).

Janus Symbian Engine ([www.janus-flash.com/products-janus-symbian.html](http://www.janus-flash.com/products-janus-symbian.html)) can be downloaded from SourceForge, at [sourceforge.net/projects/janussymbianeng/](http://sourceforge.net/projects/janussymbianeng/).

## 7 Conclusion

In this paper we have examined some popular methods for extending the capabilities of Flash Lite with Java ME or Symbian C++. Conversely, these could also be considered as ways of adding a Flash Lite UI to a Symbian C++ or Java ME application. This brings the advantage of tools that graphic designers and UI designers are more comfortable with to the UI development process.

If you are looking for maximum device coverage then Jarpa is an attractive option for Flash Lite application distribution. However, extending Flash Lite capabilities with Jarpa is somewhat restricted. There is a requirement for MIDlet signing to avoid constant security dialogs while the runtimes communicate and also the full functionality is only available on devices that run a multi-tasking operating system like Symbian OS, due to the requirement for a background server process. If your target market is restricted to Symbian OS, then further narrowing it to S60 to take advantage of the built-in native features and performance of KureriLite isn't a big step.

In addition to commercial application development, these language combinations could also be very useful for rapid prototyping. However, on S60 the most popular rapid prototyping language is Python. The good news is that you can also combine Flash Lite with Python and even extend that with native C++ code where the required functionality isn't already available. These are the language combinations that we'll be discussing in the third paper in this series, to be published soon.