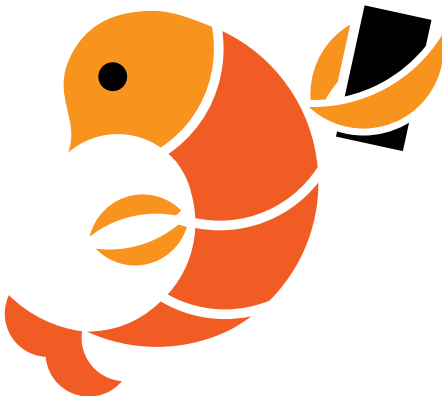
SCAMPI APPLICATION DEVELOPER GUIDE

SPACETIME NETWORKS OY



July 2014 – version 0.2

ABSTRACT

This guide describes how to develop applications for the Scampi Opportunistic Networking Middleware.

CONTENTS

1 QUICK START GUIDE

This chapter includes a quick start guide to building your first Scampi application. This guide focuses on the API use and does not explain how to configure and run a Scampi middleware instance, but assumes that one is running. The guide also does not explain how to create and configure a new project in any particular IDE, and assumes the reader is able to set up a simple Java project with the Scampi library JAR included.

The next section gives a full Hello World world example that publishes are receives messages using the Scampi platform. The following sections explain how the example was built and how it works. However, most details and advanced functionality is omitted and can be found in the following chapters.

1.1 HELLO WORLD!

The following application publishes and receives Hello World messages using the Scampi middleware:

```
1  package example;
   import fi.tkk.netlab.dtn.scampi.applib.*;

   public class ScampiHelloWorld {
     static private final AppLib APP_LIB
6        = AppLib.builder().build();

     public static void main( String[] args )
     throws InterruptedException {
       // Setup
11     APP_LIB.start();
       APP_LIB.addLifecycleListener( new LifeCyclePrinter() );
       APP_LIB.connect();

       // Subscribe to a service
16     APP_LIB.addMessageReceivedCallback( new MessagePrinter() );
       APP_LIB.subscribe( "Hello Service" );

       // Publish a message
       APP_LIB.publish( getMessage( "Hello World!" ),
21                      "Hello Service" );
     }

     private static SCAMPIMessage getMessage( String text ) {
       SCAMPIMessage message = SCAMPIMessage.builder()
26                             .appTag( "Hello" )
```

```
                                .build();
    message.putString( "text", text );
    return message;
  }

  private static final class LifeCyclePrinter
  implements AppLibLifecycleListener {

    @Override
    public void onConnected( String scampiId ) {
      System.out.println( "> onConnected: " + scampiId );
    }

    @Override
    public void onDisconnected() {
      System.out.println( "> onDisconnected" );
    }

    @Override
    public void onConnectFailed() {
      System.out.println( "> onConnectFailed" );
    }

    @Override
    public void onStopped() {
      System.out.println( "> onStopped" );
    }
  }

  private static final class MessagePrinter
  implements MessageReceivedCallback {

    @Override
    public void messageReceived( SCAMPIMessage message,
                                 String service ) {
      try {
        if ( message.hasString( "text" ) ) {
          System.out.println( "> messageReceived: "
                              + message.getString( "text" ) );
        }
      } finally {
        message.close();
      }
    }
  }
}
```

1.2 SETUP AND LIFECYCLE MANAGEMENT

Scampi applications use the services provided by the Scampi middleware instance through the Application Library (`AppLib`). AppLib is

provided as Java Archive (JAR) and comes in a deployment variant `AppLib.jar` and a development variant `AppLib-dev.jar` that includes the full source code to the Application Library.

The first step is to create a new Java project in your chosen IDE and include the AppLib.jar into the project. AppLib supports JavaSE 1.6 and newer, as well as Android.

In IntelliJ select File -> Project Structure...-> Libraries

The main class provided by AppLib.jar is `AppLib`. It serves as the endpoint of the API link and exposes the API for the application to communicate with the Scampi middleware instance. AppLib class has its own lifecycle and is designed so that a single instance can be used over the whole application lifetime, independent of the middleware lifecycle.

The API link is a local TCP link between the application process and the Scampi middleware process.

To start, create an `AppLib` instance and store a long lived reference to it, e.g., static variable:

```
package example;
import fi.tkk.netlab.dtn.scampi.applib.AppLib;

public class ScampiHelloWorld {
  static private final AppLib APP_LIB
      = AppLib.builder().build();
}
```

Creating an `AppLib` instance only reserves memory and initializes the state, no threads are created and no connection is attempted to the middleware. In order to start the `AppLib` instance, the `start()` method must be called. This starts internal threads and puts the `AppLib` into *idle* state. Note that it is not guaranteed that there will be any internal threads running, so the following example might terminate immediately:

Details of the AppLib states are explained in the next chapter.

```
public static void main( String[] args ) {
  APP_LIB.start();
}
```

The next step is to connect to a locally running Scampi middleware instance. This is done with the `connect()` method. Note that on Android `connect()` must not be called from the main thread since it tries to connect to the middleware using a socket, which will result in a `NetworkOnMainThreadException`.

```
public static void main( String[] args ) {
  APP_LIB.start();
  APP_LIB.connect();
}
```

If there is a local middleware instance running, the `AppLib` will now be connected to it. If we want to know whether the connection was successful, we can attach an `AppLibLifecycleListener` to the `AppLib`. The interface defines methods invoked during transitions of the `AppLib` lifecycle:

```
private static final class LifeCyclePrinter
implements AppLibLifecycleListener {

  @Override
  public void onConnected( String scampiId ) {
    System.out.println( "> onConnected: " + scampiId );
  }

  @Override
  public void onDisconnected() {
    System.out.println( "> onDisconnected" );
  }

  @Override
  public void onConnectFailed() {
    System.out.println( "> onConnectFailed" );
  }

  @Override
  public void onStopped() {
    System.out.println( "> onStopped" );
  }
}
```

Attaching a lifecycle listener:

```
public static void main( String[] args )
throws InterruptedException {
  APP_LIB.start();
  APP_LIB.addLifecycleListener( new LifeCyclePrinter() );
  APP_LIB.connect();
}
```

Starting the application now without a local Scampi middleware instance running will result in the following output:

```
> onConnectFailed
```

Starting the application when there is a running Scampi middleware instance that later shuts down results in the following output:

```
> onConnected: RSASHA256-4D62E1168CB48708E9F887D6182...
> onDisconnected
```

To shut down the `AppLib` instance and all of its running threads, we can call `stop()`. After calling `stop()` the instance cannot be used ever again and must be discarded.

```
APP_LIB.stop();
```

1.3 PUBLISH/SUBSCRIBE MESSAGING

The main function of the Scampi middleware is to publish messages into the opportunistic network and to receive messages published by other users. This is done via the *publish/subscribe* API exposed by the

`AppLib`. The `publish()` method takes as parameters a `SCAMPIMessage` to publish, and a service identifier to which to publish the message.

Messages in the Scampi system are key-value maps, where keys are strings and values integers, floats, strings or binary objects. A message containing a single string with the key "text" can be created as follows:

```
private static SCAMPIMessage getMessage( String text ) {
  SCAMPIMessage message = SCAMPIMessage.builder()
                              .appTag( "Hello" )
                              .build();
  message.putString( "text", text );
  return message;
}
```

The message creation uses the same AppTag ("Hello") for all messages. This causes the middleware to replace the old hello message with the new one any time the application is invoked. Not specifying the same AppTag would cause every invocation of the application to generate another message in the middleware. This way there is only one copy of the message in the middleware instead of multiple copies.

The `AppLib` API methods can be called at any time after calling `start()` and before calling `stop()`, regardless of the lifecycle state. If the API is not connected at the time of a method call, e.g., `publish()`, the call is buffered and delivered as soon as the API connection is established.

publish() will block if the internal buffer gets full, the buffer holds 2,147,483,647 calls.

```
public static void main( String[] args )
throws InterruptedException {
  // Setup
  APP_LIB.start();
  APP_LIB.addLifecycleListener( new LifeCyclePrinter() );
  APP_LIB.connect();

  // Publish a message
  APP_LIB.publish( getMessage( "Hello World!" ),
                   "Hello Service" );
}
```

To receive messages, the application can `subscribe()` to a service. Two steps that must be taken: 1) Add a `MessageReceivedCallback` to be executed when `AppLib` receives a new message from the local middleware instance, and 2) subscribe to a service so that the middleware instance starts delivering messages to the application. The application must `close()` received messages after it is done using them, in order to release any backing resources held by the message. Callback can be defined as follows:

Large binary values are stored in files, close() will delete the files.

```
private static final class MessagePrinter
implements MessageReceivedCallback {
  @Override
  public void messageReceived( SCAMPIMessage message,
                               String service ) {
    try {
      if ( message.hasString( "text" ) ) {
        System.out.println( "> messageReceived: "
                          + message.getString( "text" ) );
      }
    } finally {
      message.close();
    }
  }
}
```

The callback is added and subscription activated as follows:

```
public static void main( String[] args )
throws InterruptedException {
  // Setup
  APP_LIB.start();
  APP_LIB.addLifecycleListener( new LifeCyclePrinter() );
  APP_LIB.connect();

  // Subscribe to a service
  APP_LIB.addMessageReceivedCallback( new MessagePrinter() );
  APP_LIB.subscribe( "Hello Service" );

  // Publish a message
  APP_LIB.publish( getMessage( "Hello World!" ),
                   "Hello Service" );
}
```

The subscription causes the middleware to deliver to the application all messages published to the subscribed service, including any messages published by the application itself ("loopback"). Therefore the output of the program will be to following, assuming a local middleware instance is running:

```
> onConnected: RSASHA256-4D62E1168CB48708E9F887D6182...
> messageReceived: Hello World!
```

2 WORKING WITH THE SCAMPI API

2.1 APPLIB

The AppLib provides the inter-process communication between applications and the Scampi middleware.

2.1.1 *AppLib Lifecycle*

`AppLib` instances are meant to persist the whole lifetime of an application. Since the Scampi middleware runs in a separate process from the application, their lifecycles are not coupled. In other words, the middleware instance may shut down and start up multiple times during the application lifetime. `AppLib` manages this through two mechanisms: 1) application controllable and observable lifecycle, 2) transparent buffering of API calls that cannot be fulfilled until the middleware instance is connected.

The `AppLib` lifecycle is showin in Figure 1. The figure shows the `AppLib` API calls that can be used to control the lifecycle state (black dashed arrows in the figure), and the callbacks that can be used to observe changes in the state (grey arrows). To get callbacks, the application must add an `AppLibLifecycleListener` to the `AppLib` instance using the `addLifecycleListener()` method. The listener methods are invoked from a dedicated, single thread and will not block the `AppLib` (although it will block other listeners). Therefore there is no guarantee what the lifecycle state of the `AppLib` will be at the time of invocation, since the state may have changed while the listeners are being called.

After construction, the `AppLib` instance will be in the `NEW` state. In this state only basic initialization and memory reservations have been done. API calls that interact with the middleware, e.g., `publish()`, `subscribe()`, will result in `IllegalStateException`. However, local API calls such as adding callbacks and listeners can be used. To activate the `AppLib`, the application must call `start()`. This moves the `AppLib` into the `IDLE` state, possibly starting internal threads, and makes the full API available.

To connect to the local Scampi middleware instance, the application can call the `connect()` method. This will attempt to open a local TCP Scampi API link to the middleware instance. The `connect()` method opens and tries to connect a socket, which can potentially block (or cause `NetworkOnMainThreadException` on Android) so it should be called from a worker thread or pool. Depending on whether the connection is successful, the `AppLib` instances moves to either the

CONNECTED state or back to the IDLE state. These transitions invoke the onConnected and onConnectFailed callbacks respectively. Typical application might respond to an onConnectFailed by scheduling a new connect() attempt at some time later, retrying until a middleware instance is available.

The middleware instance can be completely reset and reinstalled during the application runtime.

In the CONNECTED state the application is actively interacting with the middleware instance. On entering the CONNECTED state, the middleware will have a fully fresh state for the API client (i.e., the application), even if the same AppLib instance has been previously connected to it. Therefore the application should expect to receive copies of the same messages it has received previously, and must implement duplicate filtering by itself if required.

To stop the AppLib the application can call the stop() method. This will stop any internal threads and invoke the onStopped callback once done. Once stop() has been called, the AppLib instance will move to the STOPPING state, followed by the TERMINATED state. In these states the AppLib API cannot be used and will result in IllegalStateException if attempted. Further, the AppLib instance cannot be reused after this point and must be discarded.

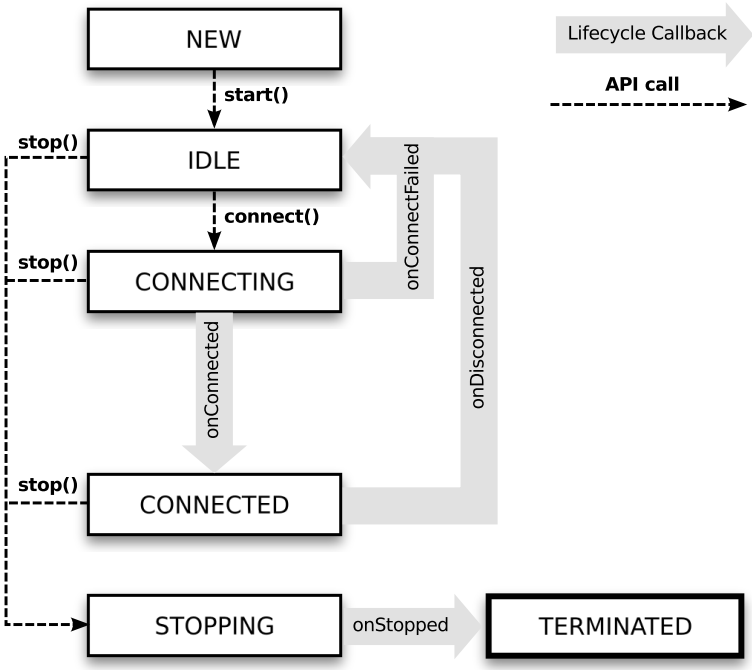


Figure 1: AppLib lifecycle.

2.1.2 *Publish/Subscribe*

The main functionality of the Scampi platform is to enable applications to exchange messages. In classical, well-connected networks applications typically send unicast messages directly between each

other. However, the unicast model is not well suited for opportunistic networks composed of chance encounters between users. Although routing algorithms for unicast routing in these networks do exist (and some are supported by the Scampi platform), the network is fundamentally better suited for applications that seek to broadcast content to many users around them. To support this model of communicating, Scampi provides a *publish/subscribe* API to the applications through the `publish()` and `subscribe()` methods.

To send a message, the application specifies the message to be published and a service identifier in the call to `publish()`. The service identifier is an opaque string with no semantics. If the application wants to know when the publish operation has succeeded, it can supply a callback in the `publish()` method. In general the application should always supply a callback that closes the published message after the operation has succeeded.

To receive messages, the application subscribes to a service by providing the service identifier to the `subscribe()` method. Typically the application will pick its own service identifier (or a set of identifiers) that all instances publish and subscribe to. However, it is possible to create services that many different applications subscribe to. For example, one could create a jukebox application that periodically publishes messages containing music files, to which any application that wants to receive music can subscribe to (e.g., music players or games that want to play background music).

To receive the messages following a subscription, the application must install a `MessageReceivedCallback` in the `AppLib` before activating the subscription. The callback can be set to receive messages from a specific service or all messages from all subscriptions. Calling `subscribe()` for an already subscribed service will result in all the messages for that service to be handed to the callbacks again.

2.1.3 *Peer Discovery*

Besides sending and receiving messages, the Scampi platform also allows applications to discover users around them. This is achieved through the peer discovery system. The application can install a `HostDiscoveryCallback` to receive updates about the discovered peers. The discovery information includes the node identity (opaque string), time of discovery, location of discovery (if known), and the number of hops away that the discovery was made.

2.2 SCAMPIMESSAGE

The basic data unit for communicating in a network composed of Scampi nodes is the `SCAMPIMessage`. This is comparable to byte streams in TCP or datagrams in UDP. However, unlike TCP and UDP that sim-

ply move bytes, Scampi messages provide structured and typed content, which in many cases can save the application from having to do its own serialization. In particular, the application sees the message contents as a key-value map of integer, float, string, and binary values (see Section 2.2.2). Applications can also control certain aspects of how the messages are handled by the middleware through message settings (see Section 2.2.1). Finally, metadata can be attached to the messages to provide generic descriptions of the content that is understood by the middleware and other applications (see Section 2.2.3).

2.2.1 *Message Settings*

Message settings control the handling of the message by the middleware. The application can set the message *lifetime*, replace old messages with newer ones by setting the `AppTag`, and ask the middleware to hold a permanent copy of the message by setting its *persistence*.

The settings should be set during message construction, using the builder mechanism:

```
SCAMPIMessage msg = SCAMPIMessage.builder()
                        .lifetime( 1, TimeUnit.DAYS )
3                       .persistent( false )
                        .appTag( "my message" )
                        .build();
```

Message Lifetime

Scampi middleware can be configured to learn the correct time from the peers that it meets even if the local device does not have a synchronized clock.

Every Scampi Message has a lifetime after which it is deleted from the network. The lifetime can be set using the `setLifetime()` method. By default the lifetime of messages is 12 hours. It should be noted that the lifetime is converted to an absolute deletion time, which is used by other nodes in the network to delete their message copies. This means that if the platform that the application is running on has incorrect time, the actual lifetime of the message in the network will not match the provided lifetime.

AppTag Message Identifier

Each message is identified by an `AppTag`, which is a local identifier for the message shared by the application and the middleware. The primary use of the `AppTag` is to identify different Scampi Message versions that are logically the same message. In particular, publishing a message through the `publish()` method will replace an existing message with the same `AppTag` in the middleware. This is useful for messages that the application wants to update over its lifetime without spreading multiple different versions into the network at the same time. For example, a message that contains the user's social networking profile should be marked with the same `AppTag`, so that when the

application updates some part of the profile and re-publishes it, the middleware will replace the old version with the new. `AppTags` are opaque string with no semantics, and are set by passing a string to the `SCAMPIMessage` constructor. If no `AppTag` is provided, the system will pick a random one.

Message Persistence

Messages published into the Scampi middleware will be placed in a space-limited cache on the disk. Once the cache fills up, the oldest messages will start to get deleted to make room for new ones even if their lifetime has not expired yet. Messages published by local applications are put into a separate cache from the messages received from the network, meaning that received messages cannot push the locally generated message out of the cache.

However, local applications can still push other applications' messages out of the cache. For some types of messages this is the desired behaviour, but for other types of content the application may want to ensure that the message will persist in the middleware and cannot be pushed out even by other local applications. To achieve this, the application can mark a message as *persistent* by using the `setPersistent()` method. Messages marked as persistent are placed into their own cache, which is typically configured to never delete its messages. Applications should take care not to over flood the persistent cache and use it conservatively for messages for which it makes sense (e.g., a message containing the user's profile).

2.2.2 *Message Content*

Scampi Message content is a typed key-value map of content items. Keys are strings while values are integers, floats, strings, and binaries. The application adds content items by calling the `putInteger()`, `putFloat()`, `putString()`, and `putBinary()` methods. Keys are unique and calling a put method overwrites any existing content item with the same key, regardless of the type. To retrieve a value, the application can call `getInteger()`, `getFloat()`, `getString()`, and `getBinary()` methods. These will either return the value or throw an `ApiException` if a content item with the given key and correct type doesn't exist. The existence of a content item can be queried using the `has*()` methods.

Binary Content

Scampi Messages can contain arbitrarily large binary values in the content. When constructing a message, the application can add a binary value either as a `File` or as a `byte[]` buffer via the two different variants of `putBinary()`. For binary values small enough to be held in memory, the `byte[]` buffer variant is recommended, while for large

binary values a backing file is more efficient. In both cases the passed parameter becomes the backing resource for the message, and must not be modified before the message has been discarded. When receiving a message with a binary content item from the middleware, it may be backed by either a file or a memory buffer, as decided by the middleware.

Because the size of binary parts can vary greatly, there are multiple ways to access them: `getBinary()`, `getBinaryBuffer()`, `copyBinary`, and `moveBinary()`. To decide which method to use, the application can call `getBinarySize()` to learn the size of the binary value. The first option, `getBinary()`, will return a direct `InputStream` to the binary value's backing resources (i.e., a file or memory buffer). This can be used when the application just wants to parse the content, without necessarily wanting to persist it. The second option, `getBinaryBuffer()`, makes an in-memory copy of the binary value and returns it. This should be used only with small binary values that the application wants to process directly from a memory buffer and keep around for an extended period. The third option, `copyBinary()` can be used to copy the contents of a binary value into a file. This can be used if the application wants to make a persistent copy of the binary value on disk.

The application has exclusive ownership of all the backing resources of the messages received from the middleware. This has two consequences in the case of a file backed binary value: 1) the application can choose to move the file to its own directory for later use (instead of needing to make a copy), and 2) the application must delete the backing files after it is done with the message by calling `close()`. To move a backing file received as a binary content item in a Scampi Message, the application can call `moveBinary()`. This causes the backing file to be moved to the specified location, where the application has direct access to it (the result will be the same even if the binary is backed by a memory buffer instead of a file). Behind the scenes, the Application Library attempts to perform this move via the most efficient mechanism possible, e.g., by moving the existing backing file to the specified location. This method should be used with large binary values that the application wants to persist on disk. By using `moveBinary()` instead of the other variants on large binary values, the application can often avoid an expensive filesystem copy operation and do a cheap rename operation instead.

2.2.3 *Metadata*

While applications use the message content to communicate data between them, in some cases the application may want to communicate something about the message to the middleware itself (or in some cases to other applications). This is achieved by attaching meta-

data to the message using the `addMetadata*()` methods. Metadata is namespaced and typed key-value pairs. There are variants to add integers, floats and strings as metadata. Metadata can be read using the `getMetadata*()` methods.

Currently there are no metadata specifications that the middleware understands, but these can be defined in separate specifications later. Applications should therefore avoid using the metadata and prefer adding content items instead, even though the metadata can be read and written by the applications.

2.3 ANDROID HTML5 API

In addition to the native Java API described in the previous sections, Scampi provides a JavaScript API for use with the Android `WebView` component. This allows HTML5 applications to use the services provided by the middleware. The support is provided by the `ScampiAndroidHtml5.jar` package, which includes the `AppLib` library.

2.3.1 *Setup*

JavaScript APIs can be attached to the Android `WebView` components through the `addJavascriptInterface()` method. The Scampi API is provided by the `ScampiJavaScriptInterface` class, which can be attached to a `WebView`. Each instance of the API contains an `AppLib` instance that is used to communicate with the middleware. In order to function, the application must have networking and external storage permissions set in the Android manifest.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.
    WRITE_EXTERNAL_STORAGE"/>
```

API instances are created by using the `builder()` method, which has two mandatory parameters: 1) the `WebView` instance to which the instance will be attached, and 2) a directory where to store files received as Scampi message content parts. Optionally, the builder can be used to specify the port that the `AppLib` instance should try to connect to, and the retry interval in case the connection fails.

The builder creates an `InterfaceInstance` that contains the actual API object to be passed to the `WebView` (`javaScriptInterface`), and a `controller` that is used to `start()` and `stop()` the API object (i.e., the `AppLib` enclosed by the API). This split is needed because Android exposes all public methods of the API object through JavaScript, while the controller interface should only be accessible from Java.

```
private void setupWebView( WebView webView,
                           File fileStorage ) {
  // Create Scampi JavaScript interface
  ScampiJavaScriptInterface.InterfaceInstance intf
      = ScampiJavaScriptInterface
          .builder( webView, fileStorage )
          .build();

  // Used to start and stop the JavaScript interface
  // (which uses AppLib internally)
  this.javaScriptController = intf.controller;
  this.javaScriptController.start();

  // Add the interface to the WebView
  webView.addJavascriptInterface(
      intf.javaScriptInterface, "Scampi" );
}
```

It is important to use the controller to stop the API instance when the attached `WebView` is disposed. Otherwise the active `AppLib` instance enclosed by the API will remain running. This can be done, e.g., in the Android `onDestroy()` lifecycle callback.

```
@Override
public void onDestroy() {
  this.javaScriptController.stop();
  super.onDestroy();
}
```

In addition, Android requires some further configuration for the `WebView` component in order to run HTML5 applications correctly (these are not specific to Scampi):

```
// Setup the web view
WebSettings settings = this.webView.getSettings();
settings.setJavaScriptEnabled( true );
settings.setUserAgentString( "ScampiHelloWorld" );
settings.setDomStorageEnabled( true );
```

2.3.2 *JavaScript API*

The API exposes two JavaScript functions: `publish()` and `subscribe()`. These are roughly equivalent to the `AppLib` `publish()` and `subscribe()` methods.

The `publish()` method has two arguments, *content* and *settings*. Both of these are stringified JSON objects. The content object specifies the content items that become the Scampi Message content (see 2.2.2). The JSON property values can be strings or numbers; booleans are not allowed (Scampi Message content has no boolean type).

The settings object specifies the optional message settings (see 2.2.1):

1. "appTag": String that defines the message AppTag.

2. "lifetime": Number that defines the message lifetime in seconds.
3. "persistent": Boolean that defines the persistence of the message.
4. "fileProperties": Array of content property names that are file paths.

In addition, it is possible to attach files as the message content. This is done by specifying the path to a file as a string property in the content JSON object. Normally the path would be treated as any other string content item, but the application can identify it as a file path by adding its name to the `fileProperties` array in the settings object. This causes Scampi to read the file from the given path and attach the file contents to the message.

```
function publish() {
  var content = {
    text: "Hello World!"
  }

  var settings = {
    lifetime: 600,
    appTag: "hello-message"
  }

  Scampi.publish( JSON.stringify( content ),
                  JSON.stringify( settings ),
                  "hello-service" );
}
```

To receive messages, the application can call the `subscribe()` method.

```
window.onload = function() {
  Scampi.subscribe( "hello-service", "receive" );
}
```

This method takes two string arguments: name of the *service* to subscribe to, and the name of a *callback* to invoke when a matching message is received. The callback gets two arguments: the stringified JSON *content* and *settings* objects.

```
function receive( content, settings ) {
  var contentModel = JSON.parse( content );

  // Container element for messages
  var recvField = document.getElementById( "received" );
  if ( !recvField ) {
    recvField = document.createElement( "div" );
    recvField.id = "received";
    recvField.innerHTML = "<h1>Received</h1>";
    document.body.appendChild( recvField );
  }

  // Message element
```

```
  var pre = document.createElement( "pre" );
  var text = document.createTextNode( contentModel[ "text" ] );
  pre.appendChild( text );
  recvField.appendChild( pre );
}
```

2.3.3 *Security Considerations*

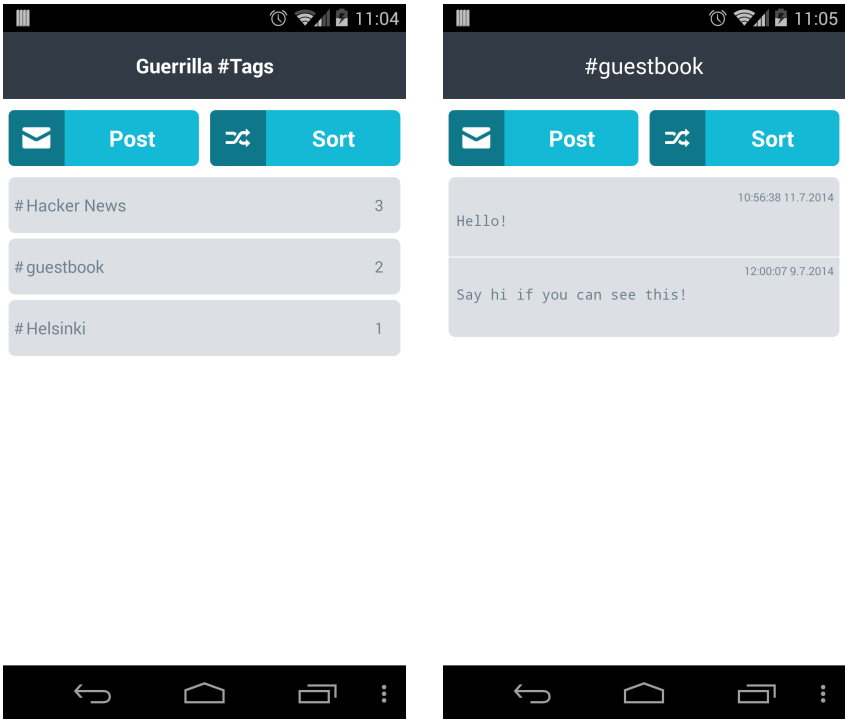
In Android earlier than 4.2 and target API level 17, the framework exposes *all* public methods of the interface object, including all inherited methods, to the JavaScript running in the `WebView`. This includes `Object.getClass().forName()`, which allows malicious JavaScript to call any method of any class loaded by the Android runtime. Therefore untrusted JavaScript should never be run in a `WebView` that has loaded any JavaScript interface, including the Scampi API.

3 EXAMPLE APPLICATIONS

This chapter introduces the example applications that are included in the Scampi Developer kit.

3.1 GUERRILLA TAGS

GuerrillaTags is an anonymous message board application that runs on Android devices. It allows users to post messages tagged with a topic ("Tag"). The messages are organized as boards using the Tags. Each application instance maintains a database of messages that the user has posted and that it has received from the SCAMPI middleware. Messages posted by the user are passed to the SCAMPI middleware, which spreads them opportunistically to other nodes.



(a) View of tags. (b) View of messages.

Figure 2: Guerrilla Tags interface.

Figure 2 shows the user interface of the GuerrillaTags Android application. On the left side, all the boards are shown with their Tags or titles. The user can post a new message to an existing board and to a new board. When the user clicks on a board, all the messages sent to that board displayed as shown on the right side of the figure.

3.1.1 *Application Architecture*

The application is composed of three main parts: 1) *Activities* that make up the user interface of the application, 2) *database* where the messages are stored, and 3) *AppLibService* that handles the Scampi API interactions

The user interface populates its views exclusively from the database through the *MsgDatabaseHelper*, including live updates through a callback mechanism that is invoked when a new message gets inserted into the database. The *AppLibService* inserts received messages into the database, with the database taking care of filtering out duplicates that would otherwise occur when the application receives copies of the same message multiple times over time (e.g., when restarting the application).

The *AppLibService* exposes and API for the user interface to publish messages directly, but also queries the database occasionally for unpublished messages (e.g., when the middleware connects). It is also possible to structure the architecture in a way that the *AppLibService* installs itself as a listener to a database service, meaning that the user interface only interacts with the database and never with the *AppLibService* directly. This approach is used in the *Guerrilla Pics* example application, for example.

3.1.2 *Scampi API Usage*

The Scampi API usage is handled by the *AppLibService*, which encapsulates all logic for interacting with the middleware.

The *AppLib* lifecycle is tied to the Android *Service* lifecycle using the Android *onCreate* and *onDestroy* callbacks. The *AppLib* instance is created and *start()*ed in the former, and *stop()*ed in the latter.

```
@Override
public void onCreate() {
  super.onCreate();

  // Create timer
  this.scheduledExecutor
      = Executors.newSingleThreadScheduledExecutor();

  // Create database helper
  this.db = new MsgDatabaseHelper( this );

  // Create AppLib and try to connect.
  this.appLib = this.getAppLib();
  this.appLib.start();
  this.scheduleConnect( 0, TimeUnit.MILLISECONDS );
}

@Override
```

```
public void onDestroy() {
  this.scheduledExecutor.shutdownNow();
  this.appLib.stop();
  super.onDestroy();
}
```

The *AppLib* lifecycle callbacks are used to query and publish all "unrouted" messages (message that have not been passed to or received from the middleware) when the *AppLib* connects through the *onConnected* callback. The *AppLib onDisconnected* and *onConnectFailed()* callbacks are used to schedule an executor task to call *AppLib connect()* some time later in order to try to re-establish the API connection (*scheduleConnect()*). It's important to note that the same method is used initially in the *Service onCreate()* callback, since calling *AppLib connect()* from the main thread would result in a *NetworkOnMainThreadException*.

```
@Override
public void onConnected( String scampiId ) {
  this.publishUnrouted();
}

@Override
public void onDisconnected() {
  this.scheduleConnect( RECONNECT_PERIOD,
                        TimeUnit.MILLISECONDS );
}

@Override
public void onConnectFailed() {
  this.scheduleConnect( RECONNECT_PERIOD,
                        TimeUnit.MILLISECONDS );
}

@Override
public void onStopped() {
  // Terminal state
}
```

The Android *Service* lets activities and other service to bind to it using a *Binder*, and exposes a single API call *publish()* that lets the other components publish messages to the middleware. The main user of this API is the *NewPostActivity* which invokes it when the user has entered a new message.

The *publish()* method composes a *SCAMPIMessage* using *getScampiMessage()* that includes four content items:

1. Tag (*String*): The tag that the message belongs to.
2. Message content (*String*): The message content.
3. Timestamp (*Integer*): the UNIX timestamp when the message was created.

4. Unique ID (*Integer*): random integer that uniquely identifies the message in conjunction with the timestamp.

```
private SCAMPIMessage getScampiMessage(
      String tag, String content,
      long timestamp, long uniqueid ) {
  SCAMPIMessage msg = SCAMPIMessage.builder()
                        .lifetime( MSG_LIFETIME )
                        .build();
  msg.putString( MSG_TAG_FIELD, tag );
  msg.putString( MSG_CONTENT_FIELD, content );
  msg.putInteger( MSG_TIMESTAMP_FIELD, timestamp );
  msg.putInteger( MSG_UNIQUE_ID_FIELD, uniqueid );
  return msg;
}
```

The message is then passed to the *AppLib* using the *AppLib.publish()* method.

When receiving messages, the middleware first calls the *messageReceived()* callback of *AppLibService*, which calls *handleIncomingMessage()*. This method then tries to read the above four fields from the received message, and if successful inserts the received message into the database by using the *MsgDatabaseHelper.insertMessage()* method.