Accessing native code from Java ME on Symbian

This article provides a framework for accessing native services on the device from Java ME.

Introduction

The J2ME Mobile Information Devices Profile (MIDP) has become ubiquitous on mobile phones, with MIDP having shipped in hundreds of millions of units. The Mobile Information Device Profile provides a standard API tailored to the requirements of cell phones and other low-end mobile devices. MIDP typically sits on top of the Connected Limited Device Configuration (CLDC), which is in essence a stripped down Java runtime environment designed to be compatible with the highly constrained hardware resources available on a typical mobile phone. The widespread adoption of CLDC/MIDP by handset manufacturers has established it as the de-facto Java Platform for cell phones and similar mobile devices.

The brief of the CLDC and MIDP experts group was to provide a complete Java Application Environment with a ROM footprint less than 500 kB. To comply with these requirements required some tough architectural decisions, particularly in the realm of security. For instance the memory requirements of the Java 2 Standard Edition (J2SE) security classes alone exceed the total memory budget available to J2ME CLDC/MIDP. Hence the J2ME CLDC specification dictates a much simpler "sandbox" security model.

A MIDP application (MIDlet) therefore runs in a closed environment and is only able to access a predefined set of classes and libraries supported by the device (“system classes” i.e. CLDC, MIDP, J2ME optional packages + any manufacturer-specific classes e.g. Nokia UI). The set of native functions available to the virtual machine is restricted only to such native functions required by the “system classes” defined above. To enforce this restriction a CLDC VM has a built in class loader which can only load classes from the predefined set of system classes or the application (MIDlet) JAR file. To avoid this restriction being bypassed, unlike a J2SE VM, the class loader can not be replaced, overridden or reconfigured by the user. Similarly there is no support for loading arbitrary native libraries via `System.loadLibrary` and consequently no support for the Java Native Interface (JNI) API.

CLDC/MIDP specifications provided a “lowest common denominator“Java platform targeted at the capabilities of the average cell phone at the time the specification was defined. As such the capabilities offered by the MIDP API were highly restrictive. It was recognized that more sophisticated high-end phones and smart phones had greater hardware resources available and could support a more extensive API. To ensure flexibility and satisfy the demands of the higher-end devices, the Java Community defined (and continues to define) optional J2ME APIs that can sit on top of the CLDC/MIDP stack, providing additional functionality above and beyond that offered by CLDC/MIDP. These are the so-called optional packages referred to earlier. Typical examples are the Wireless Messaging API (JSR 120) which gives MIDlets the ability to send and receive SMS messages and the Mobile Media API (JSR 135) which provides multimedia services to MIDlets. However, as we saw earlier these optional packages may only be provided as “system classes”. In other words they are part of the predefined set of system classes that have to be built into the device when it ships.

In short it is not possible for the user/developer to extend the range of libraries (APIs) supported by a CLDC/MIDP phone beyond the pre-defined set that was built in to the device1 short of re-flashing the phone. So although the modern CLDC/MIDP smart phone typically provide a rich API set, this remains a closed set. If the supported API does not include some crucial functionality required by a specific application, in the absence of JNI what can the developer do? In this paper we will show how on Symbian OS the MIDP developer can use standard communication protocols to access native services running on the host phone and access data not otherwise accessible to a Java MIDlet.

First we shall take a look at the pre-requisites of the native services framework and then look at the components required to realize it. After that we look at a concrete example of the framework, allowing a MIDlet to access a list of running processes on a Symbian phone. Then we move on to show how we build and customize the framework. We wrap up the article by looking at possible use cases for the framework.

Fundamentals of the MIDP native services framework

Just as a MIDlet can communicate with applications running on remote hosts using standard communications protocols, so a MIDlet running on Symbian OS can communicate with another application running on the same host using the local loop back address (127.0.0.1). If the peer application is a Symbian OS C++ application then of course the C++ application has the full native API set at its disposal. By agreeing a common protocol a Java MIDlet can make data requests of the native application which can then reply with a response in the agreed data format.

Of course this framework is not unique to Symbian, it could be offered by any open, multi-tasking OS providing that it satisfies a few criteria:

- Full multi-tasking capabilities

Obviously the OS must be able to support multiple running processes

- Open platform

The OS must be open to the installation of native applications, in addition to Java MIDlets.

- Support for suitable common communications supported by both native and MIDP applications, sockets being the most natural.
- Local loop back support and resolution of "localhost" or 127.0.0.1 IP address.

Indeed to illustrate this point the framework has also successfully run on Windows Mobile™ devices. Even so the above criteria rule out the vast majority of phones, since most devices support proprietary non-open operating systems and only offer the capability to install MIDlets, not native applications.

On Symbian OS the only consideration as to the applicability of the framework is support for client sockets within MIDP. Under the MIDP 1.0 specification support for client sockets was optional, and consequently not all Symbian OS MIDP 1.0 phones include such support. However, the MIDP 2.0 specification makes support for client sockets a recommended practice, and as a consequence all Symbian OS MIDP 2.0 devices include it.

Components

What is required on the native side is a so-called daemon. This is an EXE program that is always resident and processes the Java request.

It is assumed here that, owing to the nature of the framework, modifying an existing daemon will be the most frequent practice. You do not need strong Symbian C++ in this case and hence time-to-market is very fast. So setting the whole thing up the first time is the most difficult. Whether you want to understand the following standard components of the framework is entirely up to you. If all you want is being able to amend existing daemons, then usually you only have to change the native method.

EXE file(Daemon.cpp)

```
void DaemonMainL()
  {
  TInt KTestPort=8100;
  TInetAddr addr(KInetAddrLoop, KTestPort);
  RSocketServ socketServ;
  RSocket blank;
  RSocket listener;
  User::LeaveIfError(socketServ.Connect());
  CleanupClosePushL(socketServ);//ensure socket serv session closes on a leave
  User::LeaveIfError(listener.Open(socketServ, KAfInet,KSockStream, KProtocolInetTcp));
  User::LeaveIfError(listener.Bind(addr);
  User::LeaveIfError(listener.Listen(1));
  TRequestStatus status;
  TSockXfrLength dummyLength;
  TBool running = ETrue;
  _LIT8(KBadCommand, "Bad command");
  _LIT8(KClosing, "Server closing");
  TBuf8<256> buffer;
  while(running)
    {
    blank.Open(socketServ);
    listener.Accept(blank, status);
    User::WaitForRequest(status);
    if(status != KErrNone) User::Leave(KErrGeneral);
      blank.RecvOneOrMore(buffer, 0, status, dummyLength);
    User::WaitForRequest(status);
    if(status != KErrNone) User::Leave(KErrGeneral);
    if( (buffer[0] == 'p') )
      {
      ProcessList(&buffer);
      }
    else if( (buffer[0] == 'v') )
      {
      Version(&buffer);
      }
    else if( (buffer[0] == 'c') )
      {
      running = EFalse;
      buffer.Copy(KClosing);
      }
    else
      {
      buffer.Copy(KBadCommand);
      }
    blank.Write(buffer, status);
    User::WaitForRequest(status);
    if(status != KErrNone) User::Leave(KErrGeneral);
    blank.Close();
  }//end of while loop
  CleanupStack::Pop(&socketServ);
  socketServ.Close(); //closes all sockets created by this session
}
```

This is essentially standard fare for setting up a server socket. In this simple case study we have a dedicated daemon servicing a MIDlet synchronously, and as the daemon has no user interface, no advantage is gained by using the more common Symbian OS asynchronous Active Scheduler/Active object (AS/AO) paradigm. To generalize the framework to service multiple MIDlets would require using the AS/AO paradigm as part of the Symbian OS server framework (for more information on creating servers see for instance Harrison).

The DaemonMainL function opens a session with the Socket Server. A listening socket (listener) is then created, which listens on port 8100. Incoming connections are handed off to a second blank socket (blank), which receives the incoming request and writes the response. In the code sample above if the character 'p' is received the ProcessList method is called, which writes a list of the running processes to the socket.

```
void ProcessList(TDes8* aBuffer)
{
 _LIT8(KSpace, " ");
 TBuf<256> res;
 TFindProcess findP;
 aBuffer->Zero();

 for(TInt i = 0; i < 5; i++)
 {
  findP.Next(res);
  //check that we are not overflowing aBuffer
  if( (aBuffer->Size() + res.Size()) < 256 )
  {
   aBuffer->Append(res);
   aBuffer->Append(KSpace);
  }
  else
  {
   break;
  }
 }
}
```

Start On Boot

It is possible to incorporate "start on boot" behavior to ensure that the daemon is resident all of the time. Otherwise it would be closed down on every reboot and would have to be powered up manually by the user. There are several options to facilitate Start-On-Boot behavior, for example check out the **Start on Boot Registration API** from Symbian Developer Network or EZBoot from NewLC.

Coding the Java Side

With the native side in place, on the Java side you could order the daemon to yield a process list as follows:

```
conn = (StreamConnection)Connector.open("127.0.0.1:8100");
is = conn.openInputStream();
os = conn.openOutputStream();
os.write("p".toBytes());
os.flush();

byte[] data = new byte[maxlen];
int actlen = is.read(data);
```

The result (the process list) should then be in the data byte array. So generally speaking the steps you have to perform are:

- Connect to localhost
- Open an output stream and if you want to receive data too, an input stream
- Write the command to the socket and flush it
- Read from the socket if your operation expects an answer
- Process the data received from the socket

Code Example

Download the example NatWare_1.1.zip

Outside References

- J2ME API Bridge Interface