symbian Developer Network

Symbian OS IMS APIs

Symsource Ltd.

Published by the Symbian Developer Network

1 INTRODUCTION 2

2 A TOUR OF THE SIP API 2

2.1 CLASSIFYING THE CLASSES 2

2.2 SESSIONS AND SUBSESSIONS: CSIP AND CSIPCONNECTION 3

2.3 PROFILES AND REGISTRATION 5

2.4 DIALOGS AND TRANSACTIONS 6

2.5 RESPONSE, REQUEST AND MESSAGE ELEMENTS 7

2.6 SDP 7

2.7 ECOM CLIENT RESOLUTION 7

2.8 ADDRESS MANAGEMENT 8

2.9 HTTP DIGESTS 8

2.10 REFRESHING REQUESTS 8

2.11 HEADERS 8

3 USING THE SIP STACK 8

3.1 CONNECTING TO THE STACK AND ENABLING A PROFILE 9

3.2 MAKING A CALL 9

3.3 RECEIVING A CALL 12

3.4 SUBSCRIBING TO AN EVENT 13

3.5 ACTING AS A NOTIFIER 15

4 ARCHITECTURAL PATTERNS 17

4.1 WHAT PROBLEMS DO WE NEED TO SOLVE? 17

4.2 GENERAL ARCHITECTURE 18

4.3 ERROR HANDLING 29

4.4 MESSAGE CONTENTS 30

4.5 THE CLIENT RESOLVER PLUG-IN 33

5 CONCLUSION 34

6 BIOGRAPHY 34

1 Introduction

This paper forms the second part of a two-part series about IMS on Symbian. The first paper provided an overview of IMS; this paper will cover what Symbian has to offer in more detail, including a tour of the APIs, a section on how to use them, and an outline of some useful patterns.

2 A tour of the SIP API

This section provides an overview of the classes that the Symbian API offers the developer. It is an accompaniment to the existing Symbian documentation, expanding on some parts of the documentation and adding commentary elsewhere.

The Symbian SIP documentation is available online on the Symbian website, both in reference form[1] and as part of the guide to multimedia protocols.[2]

Please note that the reference documents are for Symbian OS v9.3: the v9.2 versions are currently unavailable on the website.

2.1 Classifying the classes

At first glance, the sheer number of classes listed in the reference may appear overwhelming. However, they can be more manageable by applying a little organization. To this end, the SIP classes are reclassified according to use below. In the following sub-sections, we will run through each group of classes in turn.

Session classes and callbacks:

```
CSIP
MSIPObserver
CSIPConnection
MSIPConnectionObserver
```

Dialogs and their associations:

```
CSIPDialog
CSIPDialogAssocBase
CSIPInviteDialogAssoc
CSIPReferDialogAssoc
CSIPNotifyDialogAssoc
CSIPSubscribeDialogAssoc
```

Transactions:

```
CSIPTransactionBase
CSIPClientTransaction
CSIPServerTransaction
```

Message elements:

```
CSIPRequestElements
CSIPResponseElements
CSIPMessageElements
```

---

[1] See this at http://www.symbian.com/developer/techlib/v9.3docs/doc_source/reference/reference-cpp/SIP_COM/index.html#SIP_COM%2etoc

[2] See this at http://www.symbian.com/developer/techlib/v9.3docs/doc_source/guide/Multimedia-Protocols-subsystem-guide/SIP/index.html#MultiMediaProtocols%2esip%2eindex

symbian

Profiles and registration:

CSIPProfile
CSIPProfileRegistry
CSIPProfileRegistryBase
CSIPManagedProfile
CSIPManagedProfileRegistry
TSIPProfileTypeInfo
MSIPConcreteProfileObserver
MSIPProfileRegistryObserver
MSIPRegistrationContext

HTTP Authentication:

CSIPHttpDigest
MSIPHttpDigestChallengeObserver
MSIPHttpDigestChallengeObserver2

Refreshing:

CSIPRefresh

ECOM client resolution:

CSIPResolvedClient

Address handling

CSIPAddress

SDP handling:

CSdpDocument
CSdpAttributeField
CSdpBandwidthField
CSdpConnectionField
CSdpFmtAttributeField
CSdpKeyField
CSdpMediaField
CSdpOriginField
CSdpRepeatField
CSdpTimeField
TSdpRtpmapValue
TSdpTypedTime

String constants:

SIPStrings
SdpCodecStringPool

SIP Headers:

CSIPAcceptContactHeader
CSIPAcceptEncodingHeader
CSIPAcceptHeader
CSIPAcceptLanguageHeader
CSIPAddressHeaderBase
CSIPAllowEventsHeader
CSIPAuthenticateHeaderBase
CSIPAllowHeader
CSIPAuthHeaderBase
CSIPCSeqHeader
CSIPCallIDHeader
CSIPContactHeader
CSIPContentDispositionHeader
CSIPContentEncodingHeader
CSIPContentTypeHeader
CSIPEventHeader
CSIPExpiresHeader
CSIPExtensionHeader
CSIPFromHeader
CSIPFromToHeaderBase
CSIPHeaderBase
CSIPPAssociatedURIHeader
CSIPParameterHeaderBase
CSIPProxyAuthenticateHeader
CSIPProxyRequireHeader
CSIPRAckHeader
CSIPRSeqHeader
CSIPReferToHeader
CSIPReplyToHeader
CSIPRequireHeader
CSIPRetryAfterHeader
CSIPRouteHeader
CSIPRouteHeaderBase
CSIPSecurityClientHeader
CSIPSecurityHeaderBase
CSIPSubscriptionStateHeader
CSIPSupportedHeader
CSIPTimestampHeader
CSIPToHeader
CSIPTokenHeaderBase
CSIPUnsignedIntHeaderBase
CSIPUnsupportedHeader
CSIPWWWAuthenticateHeader

2.2 Sessions and subsessions: CSIP and CSIPConnection

The SIP stack follows the Symbian client-server pattern: the functionality is implemented in server executables, which are accessed by applications via a client DLL. However, its client-side interface is unusual. Rather than using RSession2– and RSubSession2–derived classes, it instead employs

a set of CBase–derived classes. There is a fairly clean mapping between the classes and behaviors that an experienced Symbian developer might expect to find and those offered by the SIP client API.

Symbian OS SIP class	Notional client / server equivalent
CSIP	RSession2–derived session with the SIP server
CSIPConnection	RSubSession2–derived session with the SIP server
CSIP::NewL()	Open() on the RSession2–derived session
CSIPConnection::NewL()	Open() on the RSubSession2–derived session
delete CSIP instance	Close() on the RSession2–derived session.
delete CSIPConnection instance	Close() on the RSubSession2–derived session

From the table, you can see that the C classes behave as if they were smart wrappers around handles that combine the lifetimes of the handles with the lifetimes of the objects. In other C++ environments, with larger stacks and full exception support, this forms part of a powerful technique known as RAII (Resource Acquisition Is Initialization), which allows easy management of non-memory resources via stack-allocated objects.[3]

However, Symbian C classes aren't allocated on the stack, so the classes still have to be explicitly managed, much like the R class equivalents. This has some implications for the management of these classes, which we will discuss further in Section 4, on Architectural Patterns.

2.2.1 CSIP and MSIPObserver

CSIP, the root client-side session object for the SIP stack, does not expose much functionality apart from:

- methods for querying the SIP stack's support for security mechanisms and signal compressing
- a way of acquiring the appropriate CSIPConnection for a given IAP ID.

As discussed in the previous paper,[4] incoming SIP requests can potentially be received by a user-agent at any time. These incoming requests can be directed to your application through two interfaces. The MSIPObserver interface is used for handling requests on IAPs which do not yet have an associated CSIPConnection (as opposed to MSIPConnectionObserver, which handles requests on IAPs that have an associated CSIPConnection). This interface class, an implementation of which must be passed to the CSIP constructor, declares just two methods:

- IncomingRequest()
- TimedOut()

which are the bare minimum needed to handle connection-initiating requests.

---

[3] See section 14.4 of *The C++ Programming Language (3rd Edition)* by Bjarne Stroustrup pub. Addision Wesley 1997

[4] See http://developer.symbian.com/main/downloads/papers/IMS_Introduction_Part_1.pdf

2.2.2 CSIPConnection and MSIPConnectionObserver

CSIPConnection and MSIPConnectionObserver objects provide the main way of communicating with the SIP stack. CSIPConnection exposes the ability to send SIP requests. MSIPConnectionObserver defines the callback interface that an application needs to implement to receive notification of the incoming requests and responses on the connection. This callback interface is complex, and strategies for dealing with this complexity make up a large part of Section 4 on architectural patterns.

Each CSIPConnection object uses a single IAP, and there can be only one CSIPConnection per IAP per application. All SIP requests passed to a CSIPConnection instance are funnelled through the network connection associated with this IAP, and only incoming requests received through that network connection are passed through the MSIPConnectionObserver instance associated with the CSIPConnection. As such, your SIP application will be implementing *both* the MSIPConnectionObserver and the MSIPObserver interface.

2.3 Profiles and registration

The Symbian OS SIP API supports the notion of profiles. These are to the SIP stack what the access points are to the networking stack: collections of user configurable data that represent a particular connection end point. A profile consists of a SIP address of record (AoR), its security realm, its underlying IAP, SIP proxy and registrar connection details, and a profile type determining whether IETF-standard SIP or full-blown IMS registration should be used with the profile. A full description of these fields is not included in the documentation, but there is some information in the sipprofile.h and sipprofiletypinfo.h headers.

Access to SIP profiles is mediated and controlled by a server, the SIP Profile Server. Note that this is a separate server to the SIP Stack Server mentioned above, and can be used independently. Client-side access to the profile registry is through the classes CSIPProfileRegistry and CSIPManagedProfileRegistry, which operate on representations of profiles encapsulated in CSIPProfile and CSIPManagedProfile respectively. The 'plain' versions are essentially read-only; the 'managed' versions allow manipulation of the profiles programmatically and consequently requires the WriteDeviceData platform security capability from clients. In both cases, the registry class is a container for multiple instances of the profile, as well as offering some generic services, including the enabling of profiles.

'Enabling' a profile registers the contact information and address of record with the specified SIP registrar server, if the profile is not already registered. This may entail the establishment of a network connection if a connection with the appropriate IAP is not already established. But don't worry; the Symbian OS SIP server manages this process for you.

You can also programmatically register with a SIP server directly, either by using the CSIPRegistryBinding[5] class or by constructing and sending your own REGISTER request, using CSIPRequestElements and CSIPConnection::SendRequestL().

One important consequence of relying on profile 'Enabling' to register a profile is that it effectively ties the profile to a particular IAP, and therefore to a particular CSIPConnection. The point of registering is to establish a location at which you can be contacted and your IP address will be different for different access points, so sharing registrations across IAPs would not make much sense.

---

[5] This is currently not documented in the reference pages, but a brief guide to its usage can be found in the SIP Client API usage example.

2.4 Dialogs and Transactions

Below the level of the connection, we have transactions and dialogs. As outlined in the first paper in this series, a transaction consists of a single request followed by a number of responses, and a dialog is a stateful relationship between two participants in a SIP session. Both belong to individual connections, but it is possible for a transaction to be part of more than one dialog, and a dialog to encompass more than one transaction.

2.4.1 Transactions

There are two types of transactions: `CSIPServerTransaction` and `CSIPClientTransaction`, which both inherit from `CSIPTransactionBase`. The '`Server`' and '`Client`' parts of the name refer to the role of the stack in the transaction: if a transaction is incoming (i.e., we are operating in UAS mode), it will be of type `CSIPServerTransaction`; if it is outgoing, it will be of type `CSIPClientTransaction`.

Both types are created by the SIP stack, but in different ways. `CSIPServerTransaction` is created when an incoming message is received, and then passed back to the application via one of the callback methods. `CSIPClientTransaction` is created when the application calls one of the many `Send` methods on the dialog association, or the basic `SendRequestL()` on the connection. Ownership passes to the application in both cases, which again has a significant effect on the patterns of implementation you can employ.

`CSIPServerTransaction` allows you to access the full request information and to send a response. Likewise, `CSIPClientTransaction` allows you to access response information, but only that relating to the most recent response. Note that although the response and request sending methods take ownership of the data passed into them, there is no way of accessing that data once it is passed in. Therefore, if the application needs to access that data subsequently, it must make sure it has its own copy.

2.4.2 Dialogs and associations

Dialogs are represented by a single `CSIPDialog` class and a set of dialog association classes derived from `CSIPDialogAssocBase`.

The `CSIPDialog` class is created and owned by the SIP stack. Its main function, as you might expect, is to hold the state of a particular logical SIP dialog. For the most part, an application will only use it to query for various pieces of state information. The one exception to this is when the stack receives a request in-dialog and the application is expected to use the `CSIPDialog` to resolve the request to the appropriate dialog association.

Dialog associations are wrappers that abstract away some of the details of handling specific types of in-dialog interactions. Each carries the name of a particular SIP method, and offers facilities for sending requests and manipulating the state of the dialog.

There are four dialog associations, despite there only being explicit support for three dialog initiating methods (INVITE, SUBSCRIBE, and REFER) in the stack. This seeming mismatch is a result of some commonalities and asymmetries in the SIP protocol, which may not be immediately obvious.

- SUBSCRIBE and REFER both result in in-dialog NOTIFY requests being sent back to the original subscriber and referrer. The behavior of the notifying side with respect to the subscribing/referring side is essentially the same in both cases, so the stack wraps this functionality up in a single `CSIPNotifyDialogAssociation`.

symbian

- However, there are significant differences on the subscriber/referrer side, so there are separate classes for subscription and referring: `CSIPSubscribeDialogAssociation` and `CSIPReferDialogAssociation`.
- In the case of INVITE requests, the behavior of the inviter and invitee does differ during initiation. But once the call is up and running, both sides of the call can perform the same actions, so just the one `CSIPInviteDialogAssociation` is provided.

In addition to these differences, a dialog can only have a single invite association, but multiple notify, subscribe, and refer associations. For a given dialog you can only ever have a given INVITE active at a time: sending another one starts the re-invitation process and the target is refreshed. However, for SUBSCRIBEs, REFERs, and NOTIFYs this is not true: a dialog can support subscriptions to multiple events within it. So the design of the associations reflects this, with each of the event-related associations including event information and a single association mapping on to a single event in a single dialog.

2.5 Response, request and message elements

The SIP API represents responses and requests as bags of data. Each gets its respective class (`CSIPResponseElements` and `CSIPRequestElements`), but the methods on them are fairly transparent getters and setters. The transaction classes do the actual work in transmitting and receiving the data.

Despite theoretically sharing a large amount of functionality, the two classes do not share a common base class. Instead, instances of both own a `CSIPMessageElements` object, which wraps all the non-obligatory elements of a SIP message. These non-obligatory elements include both the optional headers and the body of the message.

This splitting of functionality allows different levels of message specification to be used in different circumstances. Note how `SendRequestL()` on `CSIPConnection` takes `CSIPResponseElements`, yet its equivalents on the dialog associations only take `CSIPMessageElements`, the compulsory headers being automatically supplied by the associations.

2.6 SDP

The SDP classes are arguably the simplest of all: a container `CSdpDocument` class, and a set of field classes to put in the container. Each field class neatly encapsulates a line of SDP data, with the `CSdpDocument` class offering full parsing and generation facilities for whole documents.

2.7 ECOM client resolution

The single `CSIPResolvedClient` class allows a developer to divert incoming out-of-dialog and dialog-initiating requests into their application. It provides an interface that, when implemented, should supply an XML document containing a set of matching criteria to the SIP stack, along with the same UID that is passed to an application's `CSIP` instance on construction. Then, when the SIP stack detects an incoming message that matches the criteria, it knows to divert that message to the `CSIP` instance with that UID.

This is an essential, but somewhat separate, part of using the SIP stack: as a result, we have included a separate section on its use at the end of the document, rather than discussing it in the examples.

2.8 Address management

The CSIPAddress class offers some basic address manipulation functionality to help with handling the potentially complex SIP addresses that are found in several of the SIP headers.

2.9 HTTP digests

HTTP digest authentication is the standard way of performing authentication over SIP. The SIP stack supports this in the form of a single class for manipulating the digest itself (CSIPHttpDigest), and a pair of callback interfaces (MSIPHttpDigestChallengeObserver and MSIPHttpDigestChallengeObserver2) for notifying the application of server responses to the digest. Which observer you implement depends on the nature of the challenges you expect to handle.

2.10 Refreshing requests

Some SIP requests that alter state on remote servers (e.g., SUBSCRIBE and REGISTER) are time limited, containing explicit expiry times. This means that to maintain the remote state, they have to be periodically refreshed. The CSIPRefresh class provides a generic way of doing this, either automatically, when passed as an argument to a dialog association send method or to CSIPRegistryBinding, or manually, by calling its UpdateL() method.

2.11 Headers

Finally, there are an enormous number of header classes, each one representing a single type of header. These wrap various types in different ways, depending on the type of the header, offering getters and setters as appropriate.

3 Using the SIP stack

If you are writing a SIP-using application, you typically have the following requirements:

1. the ability to initiate outbound calls (using calls in the broadest sense)
2. notification of incoming calls, and the ability to selectively accept and reject them.

In addition, if you are to make use of presence information, such as buddy lists, you are also likely to want:

3. to subscribe to the status of buddies
4. the ability to notify others of your status.

How can these user requirements be mapped on to the Symbian API? The SIP example in the Nokia S60 SDK[6] offers one way of doing the first two; part of the aim of Section 4 of this paper is to suggest an alternative basic architecture that addresses some of the difficulties in using the API.

First of all, we will go through a few examples of using the naked API as naively as possible to fulfil requirements 1 to 4. Not only will these be useful in themselves, they will also serve as motivators and reference points for the discussion of architecture that follows. Please note that they are only intended as basic demonstrations of API usage, not as suggested best practice. Indeed, some

[6] This can be found under /S60Ex/SIPExample in the Nokia Series60 v3.x SDKs.

parts of them are intentionally unsafe so as to highlight potential pitfalls - use them in your own code at your own risk!

For a contrasting set of examples that also cover some extra functionality see the 'How To Use The SIP Client API' section of the Symbian documentation.[7]

3.1 Connecting to the stack and enabling a profile

All the following examples assume that we already have a CSIP instance and a CProfileRegistry up and running. To do this, we need the following lines of code somewhere in our application.

```
CSIP* pGlobalSip = CSIP::NewL(KAppUid, globalSipObserver);
CSIPProfileRegistry* pProfileReg = CSIPProfileRegistry::NewL(*pGlobalSip,
                                                             globalRegObserver);
```

The KAppUid is the unique identifier of the CSIP object you are creating. This must be same UID as that returned to the SIP stack by the equivalent client resolution plug-in. The application UID is often used, as it is already effectively reserved, but it can be any genuinely unique UID.

The two observers, globalSipObserver and globalRegObserver, are implementations of CSIPObserver and CSIPProfileRegistryObserver respectively.

The examples also assume that we have already opened a CSIP connection and enabled a profile in a ConstructL() that looks something like this:

```
void CCallMaker::ConstructL(CSIP& aSipSession, TInt aIapId,
                            CSIPProfileRegistry& aProfileReg)
    {
    ipConnection = CSIPConnection::NewL(aSipSession,aIapId,*this);
    ipProfile = GetAppropriateProfileL(aProfileReg,aIapId);
    aProfileReg.EnableL(*ipProfile,*this);
    }
```

The implementation details of GetAppropriateProfileL() are application-specific, and we talk more about selecting a profile in Section 4. For now, we can assume that it returns the first profile it finds with the correct IAP.

3.2 Making a call

The simplest way of making a call is to implement a class that looks something like the CCallMaker definition below, creating the connection and enabling an appropriate profile on construction.

```
class CCallMaker : public CBase, public MSIPConnectionObserver
    {
public:
    void MakeCallL(CUri8* apDestination);

    virtual void IncomingResponse (CSIPClientTransaction&, CSIPDialogAssocBase&);
    virtual void IncomingResponse (CSIPClientTransaction&, CSIPInviteDialogAssoc*);
    virtual void InviteCompleted (CSIPClientTransaction&);

    // other overrides removed for clarity
```

[7] See http://www.symbian.com/developer/techlib/v9.3docs/doc_source/guide/Multimedia-Protocols-subsystem-guide/SIP/SIPClientAPI/SIPHowToUse.guide.html#mmprotocols%2esipapi%2ehow%2dto%2duse

```
private:
    CSIPClientTransaction* ipInviteTrans;
    CSIPInviteDialogAssoc* ipInviteAssoc;

    CSIPConnection* ipConnection;
    CSIPProfile* ipProfile;
    };
```

To start a call, we need to create a dialog association using the connection, the destination address, and our already enabled profile. We then call `SendInviteL()` on the association, and store the resulting transaction for later.

```
void CCallMaker::MakeCallL(CUri8* apDestination)
    {
    ipInviteAssoc = CSIPInviteDialogAssoc::NewL(*ipConnection, apDestination,
                                                *ipProfile);
    ipInviteTrans = ipInviteAssoc->SendInviteL(InviteReqElemsLC());
    CleanupStack::Pop();
    }
```

The pattern of calling `LC` functions returning request, response, or message elements in the arguments of `SendXxx()` functions, and then immediately popping returned elements off the stack once the `SendXxxL()` has returned, is used throughout these examples. It's the neatest way of safely encapsulating the element generation without littering the code with unnecessary temporaries and explicit pushes and pops on either side of ownership-taking functions.[8] It also means that we can concentrate on the structure of the example code, rather than worrying about the precise details of what might go into a message all the time.

Once the call has been started, we wait for a response. When one comes in, the implementation of the `CSIPDialogAssocBase` overload of `IncomingResponse()` will be called. To handle this response, we first check that the incoming transaction is the same as the one we stored and then get the status code. This is most likely to be a 1xx or a 4xx. In the former case, we are only interested in a 180 response for now, which signals the call is ringing at the destination, and we should alert the user. In the latter case, we simply flag up the failure of the call.

```
void CCallMaker::IncomingResponse (CSIPClientTransaction& aTransaction,
                                   CSIPDialogAssocBase& aAssoc)
    {
    if(ipInviteTrans && *ipInviteTrans == aTransaction)
        {
        // we have a response!
        const CSIPResponseElements* pRespElems = ipInviteTrans->ResponseElements();
        TUint statusCode = pRespElems->StatusCode();
        if(statusCode == 180)
            {
            DestinationRinging(pRespElems);
            }
        else if (statusCode == 200)
            {
            // ok, so send ack
            TRAPD(err,ipInviteAssoc->SendAckL(*ipInviteTrans));
            DestinationAccepted(pRespElems);
```

[8] This technique is only suitable for functions that take ownership just before returning. If a function takes ownership at the beginning, before any leaving calls are executed, the object passed in should not left be on the cleanup stack.

```
            }
        else if (statusCode >= 400)
            {
            CallFailed(pRespElems);
            }
        else
            {
            // handle all other responses here
            }
        }
    }
```

Assuming that the user at the other end picks up, `IncomingResponse()` will get called again with a 200 code. When this happens we send an ACK back, and flag to our own application that the call has been accepted by the destination.

To properly handle INVITE calls we need to implement two further methods: the `CSIPDialogAssoc` overload of `IncomingResponse()`, and `InviteCompleted()`.

Both these are necessary to support the notion of forking. A fork can occur when an intermediate proxy knows of two or more locations at which the invitee can be found, and so forwards the request to all of them. If more than one of these responds positively, the call is said to have forked and, because of the multiple locations involved, from then on it consists of more than one dialog.

As part of its support for INVITE requests,[9] the SIP stack automatically creates dialogs and dialog associations for each fork, as it is made. These dialog associations get passed in to the application as part of the `IncomingResponse()` call, and need to be handled by the application. For the moment, we are going to choose to reject all responses after the first one by immediately deleting any dialog association.

```
void CCallMaker::IncomingResponse(CSIPClientTransaction& aTransaction,
                                  CSIPInviteDialogAssoc* apAssoc)
    {
    delete apAssoc;
    }
```

`InviteCompleted()` is called when the system is finally done with the transaction created by the application as part of the initial send. In non-INVITE dialogs, the transaction is usually deleted after the final response to it is received, but this cannot be done in INVITES because the response-less ACK needs to reuse the transaction information and the INVITE might still fork, even after we have sent the ACK. So instead we delete the transaction in `InviteCompleted()`:

```
void CCallMaker::InviteCompleted (CSIPClientTransaction& aTransaction)
    {
    if(ipInviteTrans && *ipInviteTrans == aTransaction)
        {
        delete ipInviteTrans;
        ipInviteTrans = NULL;
        }
    }
```

---

[9] Note that forks can theoretically also occur in SUBSCRIBE requests but they are rare, and the SIP stack offers no explicit support for them.

symbian

3.3 Receiving a call

To receive an incoming call, we first need to have defined a `CSIPResolvedClient` plug-in. The rest of the reception process is generally independent of the resolution details, which are covered in depth below, so for the sake of simplicity we shall start at the point where the application is activated by the SIP stack.

We again need to implement a class derived from `MSIPConnectionObserver`, and pass it the `CSIPConnection` object as part of the construction process.

```
class COurApp : public CBase, MSIPConnectionObserver
    {
public:
    virtual void IncomingRequest(CSIPServerTransaction*);
    virtual void IncomingRequest(CSIPServerTransaction*, CSIPDialog&);
    // other methods deleted for clarity

    void UserHasRespondedL(TBool aUserAccepts);

private:
    CSIPServerTransaction* ipTransaction;
    CSIPInviteDialogAssoc* ipInviteAssoc;

    CSIPProfile* ipProfile;
    };
```

If the incoming call is on our connection, the SIP stack will call the simple `IncomingRequest()` callback. Once we have taken ownership of the transaction, the implementation of this callback has three stages. First, we check that the transaction is actually an INVITE, sending a rejection response and then deleting the transaction if it is not. Secondly, we check whether the call is one our application is at least theoretically interested in. If it isn't, we again send a rejection response and delete the transaction. Thirdly, now that we know we can accept the call if the user picks up, we initialize the dialog association, start the application ringing, and send back a ringing response.

```
void CCallReceiver::IncomingRequest(CSIPServerTransaction* apTransaction)
    {
    ipTransaction = apTransaction;

    // Only interested in invites
    if(ipTransaction->Type() !=
        SIPStrings::StringF(SipStrConsts::EInvite))
        {
        TRAPD(dummy, ipTransaction->SendResponseL(UnsupportedRespElemsLC());
                                                  CleanupStack::Pop(););
        delete ipTransaction;
        return;
        }

    if(!AcceptCallOnPrinciple(ipTransaction->RequestElements()))
        {
        // rejection
        TRAP_IGNORE(ipTransaction->SendResponseL(RejectionRespElemsLC());
                                                 CleanupStack::Pop(););
        delete ipTransaction;
        return;
        }
```

```
    // theoretical acceptance
    TRAP_IGNORE(ipInviteAssoc = CSIPInviteDialogAssoc::NewL(*apTransaction));

    StartRinging();
    TRAP_IGNORE(ipTransaction-> SendResponseL(RingingRespElemsLC());
    CleanupStack::Pop(););

    // Now we wait...
    }
```

We now have to wait for our user to pick up the call or reject it. If the user rejects the call, we send a rejection response and delete the transaction and dialog association. If, on the other hand, the user accepts the call, we send a 200 OK response, delete the transaction and wait for the expected ACK.

```
void CCallReceiver::UserHasRespondedL(TBool aUserAccepts)
    {
    if(!aUserAccepts)
        {
        ipTransaction->SendResponseL(UserRejectRespElemsLC());
        CleanupStack::Pop();

        delete ipTransaction;
        delete ipInviteAssoc;
        return;
        }

    ipTransaction->SendResponseL(OkRespElemsLC());
    CleanupStack::Pop();

    delete ipTransaction;
    }
```

Because the ACK request will be part of a dialog, we will be notified via the `CSIPDialog` overload of the `IncomingRequest()` callback. It's our responsibility to resolve the dialog to the correct association. This can be quite a tricky process, and we go into it further in Section 4. For now, we simply check that the dialog is the one we are using, and signal to the application to expect an incoming media session.

```
void CCallReceiver::IncomingRequest(CSIPServerTransaction* apTransaction,
                                    CSIPDialog& aDialog)
    {
    if(ipInviteAssoc->Dialog() == aDialog)
        {
        // no need to send response
        WaitForIncomingMediaSession(apTransaction->RequestElements());
        }

    delete apTransaction;
    }
```

3.4 Subscribing to an event

To subscribe to a single event, we again need to implement an instance of `MSIPConnectionObserver`. This time, it looks like this (with the uninteresting methods removed):

```
class CSubscriber : public CBase, public MSIPConnectionObserver
    {
public:
    virtual void IncomingResponse(CSIPClientTransaction&);
    virtual void IncomingRequest(CSIPServerTransaction*, CSIPDialog&);

    // other overrides deleted for clarity
    void SubscribeToEventL(CSubsEventInfo*);
    void CancelSubscriptionL();

private:
    CSIPClientTransaction* ipSubscribeTrans;
    CSIPClientTransaction* ipCancelTrans;
    CSIPSubscribeDialogAssoc* ipSubscribeAssoc;

    CSIPConnection* ipConnection;
    CSIPProfile* ipProfile;
    };
```

To start the subscription, we create a `CSIPSubscribeDialogAssoc` with enough information to identify the server and event we are subscribing to. Then we send the request using the `SendSubscribeL()` method, and wait for the response to come in.

```
void CSubscriber::SubscribeToEventL(CSubsEventInfo* apEventInfo)
    {
    ipSubscribeAssoc = CSIPSubscribeDialogAssoc::NewL(*ipConnection,
                                                  apEventInfo->RemoteUri(),
                                                  *ipProfile,
                                                  apEventInfo->Event());
    ipSubscribeTrans = ipSubscribeAssoc->SendSubscribeL();
    }
```

We receive the response via the transaction-only overload of `IncomingResponse()`, in which we check that it's the right transaction and pass the response to the rest of the application for further processing.

```
void CSubscriber::IncomingResponse(CSIPClientTransaction& aTrans)
    {
    if(ipSubscribeTrans && aTrans == *ipSubscribeTrans)
        {
        HandleSubscriptionResponse(ipSubscribeTrans->ResponseElements());

        delete ipSubscribeTrans;
        ipSubscribeTrans = NULL;
        }
    else if(ipCancelTrans && aTrans == *ipCancelTrans)
        {
        HandleCancelResponse(ipCancelTrans->ResponseElements());

        delete ipCancelTrans;
        ipCancelTrans = NULL;
        }
    else
        {
        // eek!
        }
    }
```

When the subscribed-to event is triggered, a NOTIFY request is sent out. We receive this in the CSIPDialog overload of IncomingRequest(), checking that the request is of the expected type and that it belongs to the expected dialog, deleting the transaction immediately if either of these conditions is not met. If they are both met, we send out an OK response immediately, and then delete the transaction. Note that we are not checking that the request refers to the same event as our stored transaction, which would be necessary if we subscribed to more than a single event.

```
void CSubscriber::IncomingRequest(CSIPServerTransaction* apTrans,
                                  CSIPDialog& aDialog)
    {
    if(apTrans->Type() != SIPStrings::StringF(SipStrConsts::ENotify))
        {
        delete apTrans;
        return;
        }

    if(!(aDialog == ipSubscribeAssoc->Dialog()))
        {
        delete apTrans;
        return;
        }

    TRAPD(err, apTrans->SendResponseL(OkRespElemsLC()); CleanupStack::Pop(););

    HandleNotifyRequest(apTrans->RequestElements());

    delete apTrans;
    }
```

Finally, to cancel the subscription, we call SendUnsubscribeL() on the dialog association. The response to this is handled in the transaction-only overload of IncomingResponse(), as shown above, deleting the transaction and dialog associations immediately.

```
void CSubscriber::CancelSubscriptionL()
    {
    ipCancelTrans = ipSubscribeAssoc->SendUnsubscribeL();
    }
```

3.5 Acting as a notifier

In addition to the one function per CSIPConnection assumption shared with the other examples, this example makes the even bigger assumption that there will only be a single subscriber to a single event. In real situations you have to manage multiple subscriptions per event, and possibly multiple events per dialog, which in practise means managing multiple dialog associations.

```
class CNotifier : public CBase, public MSIPConnectionObserver
    {
public:
    virtual void IncomingRequest(CSIPServerTransaction*);
    virtual void IncomingRequest(CSIPServerTransaction*, CSIPDialog&);
    virtual void IncomingResponse(CSIPClientTransaction&, CSIPDialogAssocBase&);
    void EventHasOccurredL(CEventChanges*);
```


symbian

```
private:
    CSIPNotifyDialogAssoc* ipNotifyAssoc;
    CSIPClientTransaction* ipNotifyTrans;

    CSIPEventHeader* ipEventHeader;

    CSIPConnection* ipConnection;
    CSIPProfile* ipProfile;
    };
```

At the beginning of a subscription, assuming that no dialog has been established, we receive the SUBSCRIBE request in the transaction-only overload of `IncomingRequest()`. We check that it's a subscription to the event we support by comparing headers, and if it isn't we send a not supported response. If it is supported, we extract the expiry time and prime our expiration timer. Then we create the dialog association and issue an OK response.

```
void CNotifier::IncomingRequest(CSIPServerTransaction* apTrans)
    {
    CSIPEventHeader* pEventHeader =
        ExtractEventHeader(apTrans->RequestElements());
    if(!(pEventHeader && *pEventHeader == *ipEventHeader))
        {
        TRAP_IGNORE(apTrans->SendResponseL(NotSupportedRespElemsLC());
                    CleanupStack::Pop(););
        delete apTrans;
        return;
        }

    ResetExpiryTimer(ExpiryHeaderValue(apTrans));

    TRAP_IGNORE(ipNotifyAssoc = CSIPNotifyDialogAssoc::NewL(*apTrans,
                (CSIPEventHeader*)pEventHeader->CloneL(),
                 CSIPSubscriptionStateHeader::NewL(KNotifierActive)));

    TRAP_IGNORE(apTrans->SendResponseL(OkRespElemsLC()); CleanupStack::Pop(););

    delete apTrans;
    }
```

When an event occurs, we send a notify response back…

```
void CNotifier::EventHasOccurredL(CEventChanges* apEventChanges)
    {
    ipNotifyTrans = ipNotifyAssoc->SendNotifyL(EventReqElemsLC(apEventChanges));
    CleanupStack::Pop();
    }
```

…and wait for a response, deleting it on reception.

```
void CNotifier::IncomingResponse(CSIPClientTransaction& aTrans,
                                  CSIPDialogAssocBase& aDialogAssoc)
    {
    if(ipNotifyTrans && *ipNotifyTrans == aTrans)
        {
        // check for failure here and maybe delete association
        // as well if we do fail
        delete ipNotifyTrans;
        }
    }
```

All subsequent subscription requests on the dialog will come through the CSIPDialog overload of IncomingRequest(). This includes subscriptions to new events as well as refreshes of the original event, but if we only support a single event we can safely ignore anything new.

So, in the implementation, we first check the event header of the incoming message, rejecting it if it refers to a different event. If it is our event, we then check the value of the EXPIRES header. If it's equal to zero, it means that we should immediately expire the session, so we send an OK response and then delete our transaction and the dialog association. If it's not equal to zero, it means we should refresh the session, so we reset the timer and then only delete the transaction, not the association.

```
void CNotifier::IncomingRequest(CSIPServerTransaction* apTrans,CSIPDialog& aDialog)
    {
    CSIPEventHeader* pEventHeader = ExtractEventHeader(apTrans->RequestElements());
    if(! (pEventHeader && *pEventHeader == *ipEventHeader) )
        {
        TRAP_IGNORE(apTrans->SendResponseL(NotSupportedRespElemsLC());
                    CleanupStack::Pop(););
        delete apTrans;
        return;
        }

    TInt expires = ExpiryHeaderValue(apTrans);
    if(expires == 0)
        {
        TRAP_IGNORE(apTrans->SendResponseL(OkRespElemsLC());
                    CleanupStack::Pop(););
        delete apTrans;
        delete ipNotifyAssoc;
        }
    else
        {
        TRAP_IGNORE(apTrans->SendResponseL(OkRespElemsLC());
                    CleanupStack::Pop(););
        ResetExpiryTimer(expires);
        delete apTrans;
        }
    }
```

4 Architectural patterns

The aim of this section is to suggest an alternative architecture for implementing SIP applications to the one in the Nokia S60 SDK example and to demonstrate some useful generic patterns. Because of space restrictions and the large variation in the needs of developers, this will not consist of a full multi-purpose library. Rather, we will take a largely high level, patterns-based approach, only burrowing down to specific code examples where necessary.

4.1 What problems do we need to solve?

From the assumptions made before presenting the examples above, and from the examples themselves, it should be clear that a general architecture is going to have to deal with the following problems:

- maintenance of the state of each transaction and dialog
- interoperation of multiple simultaneous dialogs and transactions
- resolution of incoming requests and responses to the appropriate dialog associations
- memory management of transactions and dialog associations
- error handling
- request and response element generation.

The first four problems overlap somewhat (e.g., the resolution of incoming requests depends on both the dialogs and transactions that are active and the state of the system, maintenance of which is affected by the issues imposed by the memory management model), whereas the last two are largely orthogonal to the others. For this reason, we will cover possible solutions to the first four together, and then deal with the remaining two afterwards. Finally, we include a brief guide to using the client resolver plug-in.

4.2 General architecture

The assumption throughout this section is that the classes described are stand-ins for your own classes that will share some of the same structure. The relationships described below are only part of the picture: they provide an outline implementation, but you need to fill in the gaps yourself. We've purposefully left out any indication of the outside world beyond some descriptive function names to prevent obfuscation of the structure. In a real implementation, these functions would likely be methods on some class object, or members of a callback interface, or even chunks of code.

4.2.1 The problem with state

The most pressing concern is arguably the maintenance of state. In any system that interacts with the world in an asynchronous manner it is necessary for some state to be held between interactions. If your interactions are very simple, then a couple of global variables might be all you need, but if you have multiple simultaneous transactions, having a large collection of unorganized variables can rapidly become unwieldy, and lead to hard-to-maintain code. Some sort of wider organizing principle is required.

The State pattern, as described by Gamma et al.,[10] and used in the S60 example, provides one well-respected method of organization. The governing idea is that each individual state is represented by a single class, with the transitions being constructed by the engine at start up. This works well when you effectively have a large, possibly system-wide, state machine and the transitions between states are complicated. This is the case in the S60 example: its assumption that there will only be a single active transaction and dialog underway at a given time means that the whole engine can be treated as a single state machine.

However, this assumption will not always hold. In many situations it is likely that an application will need to gracefully handle multiple overlapping transactions and dialogs, if only to reject them. This does not fit well with having one monolithic state machine: if each transaction or dialog can logically possess its own state, it's non-trivial to combine these states into a single aggregated state. This suggests we need to consider ways of dividing up the state into more manageable chunks.

---

[10] *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides. Addison Wesley 1995

4.2.2 Splitting state: connections

The simplest place to begin this division is at the connection level. A given dialog or transaction will only ever occur over a single IAP, and there is a one-to-one mapping between connections and IAPs, so splitting state into per-connection pieces makes sense. In addition, the majority of events are raised at the connection level, as calls to implementations of `MSIPConnection` observer, so it also seems wise to make the split from a request/response routing perspective.

To achieve this connection-level division of labor we can use a single class, called `CSymConnection`,[11] which both owns an instance of `CSIPConnection` and inherits from `MSIPConnectionObserver`.

```
class CSymConnection : public CBase, public MSIPConnectionObserver
    {
public:
    static CSymConnection* NewL(CSIP* apSip, TInt aIapId);
    TInt Iap() const;
    // other members removed for clarity, including
    // MSIPConnectionObserver's pure virtuals

private:
    void ConstructL(CSIP* apSip, TInt aIapId);
    CSIPConnection* ipConnection;
    };
```

The creation of the `CSIPConnection` instance is done in the `ConstructL()`, ensuring that all future connection instances will be directed straight to our implementation.

```
void CSymConnection::ConstructL(CSIP* apSip, TInt aIapId)
    {
    ipConnection = CSIPConnection::NewL(*apSip, aIapId,*this);
    }
```

Having outlined a basic connection class, we can go in two directions: upwards, to look at what might own the connection objects, and downwards, to look at how the connection class handles its communications with dialogs and transactions.

4.2.3 Going up: CSymSession

Above the level of the connection class, we need a way of managing the lifetime of the connection objects.

A connection will normally be created as a result of one of two things happening:

1. an end user asks to start a new call on an IAP that doesn't yet have a connection associated with it
2. a request from the outside world comes in on another IAP without a `CSIPConnection` instance.

In both cases a `CSIP` object is needed to initialize the connection, and in the latter the application will be informed via the `MSIPObserver::IncomingRequest()` method. These two facts immediately suggest a similar solution to that used for connections: a `CSymSession` object that wraps the `CSIP` object and inherits from `MSIPObserver`:

---

[11] To easily differentiate between our classes and the built in ones, all further classes we define will have the form *X*Sym*Function*

```
class CSymSession : public CBase, public MSIPObserver
    {
public:
    static CSymSession* NewL(TUid aSipUid);

    CSymConnection& CreateConnectionL(TInt aIapId);
    void DestroyConnection(TInt aIapId);
    TBool ConnectionExists(TInt aIapId);
    CSymConnection& GetConnectionL(TInt aIapId);

    virtual void IncomingRequest(TUint aIapId, CSIPServerTransaction* aTrans);

    // other methods removed

private:
    void ConstructL(TUid aSipUid);

    RPointerArray<CSymConnection> iConnections;
    CSIP* ipSip;
    };
```

Note that the `CreateConnectionL()` returns a reference and that there is no way of accessing the `CSIP` object directly. This is intentional: to help simplify memory management, all creation and destruction of connections should be done via `CSymSession`. The lack of a `CSIP` means you can't call `CSymConnection::NewL()` and you can't delete via a reference unless you explicitly use the `&` operator. It's possible to enforce this further by making `CSymConnection::NewL()` private, and making `CSymSession` a friend, but to improve readability we have left this out.

The connection accessors are a compromise between convenience and safety. Depending on your application, you may be happy dropping `CreateConnectionL()` and having `GetConnectionL()` create a connection automatically if one does not exist for the given IAP, or you might decide that `ConnectionExists()` is unnecessary, and have `GetConnectionL()` leave if the supplied IAP does not map onto a connection.

The `IncomingRequest()` method only gets called if no appropriate connection object already exists, so it is safe to implement it by simply creating a connection and forwarding the transaction on to the single transaction overload of `CSymConnection::IncomingRequest()`, like so:

```
void CSymSession::IncomingRequest(TInt aIap, CSIPServerTransaction* apTrans)
    {
    TRAPD(err, CreateConnectionL(aIap).IncomingRequest(apTransaction));
    }
```

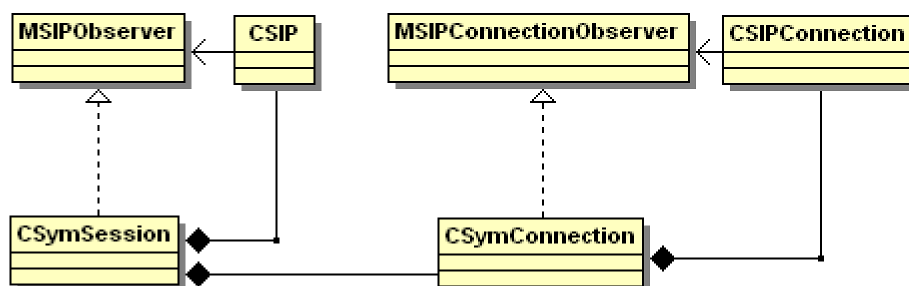

Figure 1: Simplified class diagram of the session and connection classes.

4.2.4 A brief note on error handling

You may have noticed that all the callback functions are non-leaving. This means that all leaving calls made inside them have to be trapped. This is tedious and can significantly lengthen and obfuscate the code. Two ways around this are discussed in the section on error handling below, but for now we are going to assume that all the `IncomingXxx()` functions are implemented in such a way that they immediately wrap `IncomingXxxL()` functions and sink the leaves to some global error handler.

4.2.5 Below the connection: free transactions and call bundles

We now have two classes, one representing the entire SIP session, and another a given connection. But these are not very useful by themselves. Splitting the state into individual connections is fine, but it still leaves us with a mass of transaction and dialog state to handle at the connection level. We need to look at how we can break the state (and functionality) down further and, as a corollary, how to route all the incoming requests and responses to these smaller chunks. To help with this, we now introduce two new pieces of terminology:

- Free transactions: these are transactions that are not part of a dialog, and never will be. They are characterized by being simple request and response pairs, MESSAGE transactions being the prototypical examples.
- Call bundles: these are the collections of transactions and dialogs that together make up a logical call, initiated by a single initial request. By call, we don't just mean the transactions and dialogs that surround an INVITE: the term includes publish and subscribe sessions as well.

4.2.6 Free transactions

Free transactions are easy to deal with. In the case of incoming requests, it is often not even necessary to hold any transaction state at all: if the processing of the message body can be done synchronously, then a simple call to `SendResponseL()` on the incoming `CSIPServerTransaction` followed by a deletion might be sufficient. Only one non-error callback needs to be handled: the transaction-only overload of `IncomingRequestL()`. An example framework implementation is shown below, with `ProcessTransactionLC()` taking the load of handling the details of the transaction, and creating the details of the response.

```
void CSymConnection::IncomingRequestL(CSIPServerTransaction* apTrans)
    {
    if(DialogInitiatingRequest(apTrans))
        {
        // Do call bundle stuff here
        }
    else
        {
        // free transaction
        CleanupStack::PushL(apTrans);
        apTrans->SendResponseL(ProcessTransactionLC(apTrans));
        CleanupStack::Pop(); //ProcessTransactionLC's return
        CleanupStack::PopAndDestroy(apTrans);
        }
    }
```

For outgoing requests, the transactions have to be stored while we wait for responses. A generic way of doing this is to use an `RArray`, mapping `CSIPClientTransactions` to `TCallbacks` or a self-defined observer, like so:

```
struct MSymFreeTransObs
    {
    // returns true if the handling is complete
    virtual TBool IncomingResponseL(CSIPClientTransaction& aTrans) = 0;
    };

struct TPair
    {
    CSIPClientTransaction* ipTrans;
    MSymFreeTransObs* ipObs;
    };

TBool PairEquals(const TPair& a, const TPair& b)
    {
    return *a.ipTrans == *b.ipTrans;
    }

RArray<TPair> iFreeTrans;

void CSymConnection::SendMessageL(TMessageDetails aDetails,
                                  MSymFreeTransObs* apObs)
    {
    iFreeTrans.ReserveL(iFreeTrans.Count() + 1);
    CSIPClientTrans* pTrans = SendRequestL(CreateMessageL(aDetails));
    TPair pair = {pTrans,new(ELeave)MSymFreeTransObs()};
    iFreeTrans.Append(pair);
    }

void CSymConnection::IncomingResponseL(CSIPClientTransaction& aTran)
    {
    TPair findPair = {&aTran,NULL};
    TIdentityRelation<TPair> pairEquals(PairEquals);
    TInt idx = iFreeTrans.FindL(findPair,pairEquals);

    if(iFreeTrans[idx].ipObs->IncomingResponseL(*tx))
        {
        delete iFreeTrans[idx].ipTrans;
        iFreeTrans.Remove(idx);
        }
    }
```

Note that in this schema the connection class owns the transaction, and so must be careful to delete all outstanding transactions left in the array in its destructor. Also, the `CSIPClientTransaction` is exposed directly to the callback: in real situations it might be preferable to refine the data before passing it on. Finally, if some application-specific extra state needs to be managed between the request and the response it can easily be held in the implementation of `MSymFreeTransObs`.

4.2.7 The case for call bundles

Call bundles are a much more complicated proposition than free transactions. They potentially contain several transactions, and even multiple dialogs. This complexity might suggest a need for the further splitting of responsibilities, but this is difficult to achieve.

Consider a single INVITE request. Once this is even provisionally accepted, it will result in several responses, and likely at least two further transactions. So any state describing the call initiated by that request has to be shared between the transactions.

Further, the request can fork, as described in the call-making example above. It would theoretically be possible to deal with each fork separately, but in most situations what you do with one fork is dependent on what you do with the others. For example, if you are initiating a VoIP call which forks, once the person being called picks up the phone on one fork, you are almost certainly going to want to drop all the others. It's not just handling state changes: if a transaction is shared between several objects with differing lifetimes, deciding when the transaction should be deleted becomes a tricky business.

4.2.8 Call bundles in practice

Having made a case for handling all the constituent parts of a call in one object, we need to work out what these objects might look like, and how they might interact with their parent connections.

Each call bundle object will, in part, serve as a wrapper round one or more dialog associations of the same type, with each dialog association supporting a different set of behaviors. This suggests that we probably need a separate call bundle class for each association: `CSymInvite`, `CSymSubscribe`, `CSymRefer`, and `CSymNotify`. The first of these should support multiple associations, mirroring the SIP stack's support for INVITE forking; the other three should not.

It might also make sense to separate outgoing and incoming invite classes, but using the same call objects for both is more sensible than it might initially appear. Once a call is up and running, the actions are effectively symmetrical: each side can re-invite, or terminate the call, so the state management required is essentially the same.

To add support for message routing we need some way of determining from the outside which transactions and dialog associations each bundle owns. A simple way of doing this is to have a base interface class for all the bundles, defining three Boolean methods for determining ownership and usage, as well as two methods that accept the routed requests and responses.

```
class MSymCallBundle
    {
public:
    virtual TBool UsesDialog(const CSIPDialog&) = 0;
    virtual TBool OwnsAssociation(const CSIPDialogAssocBase&) = 0;
    virtual TBool OwnsTransaction(const CSIPClientTransaction&) = 0;
    virtual void IncomingRequestL(CSIPServerTransaction*) = 0;
    virtual void IncomingResponseL(CSIPClientTransaction&) = 0;
    };
```

There is no need to test for ownership of a `CSIPServerTransaction`. These are created by the SIP stack immediately before being passed into `IncomingRequest()` functions. This means that they are never owned by any of our bundles at the initial routing stage, and there is never any need to route them once ownership has been passed to the bundles.

Because a single dialog can have several bundles of the same type simultaneously active on it (one for each event), we need to route incoming requests for the three event bundles on a combination of dialog and event. This in turn means that we need a way of distinguishing between bundles based on event.

In addition, it's possible that a given dialog has subscribe/notify sessions for a single event running in both directions. For example, user-agent A might be subscribed to user agent-B's online status while user-agent B is subscribed to user-agent A's online status. So, we can't even reliably route on the combination of event and dialog, we also need to check the type of the incoming request,

and make sure that we send a NOTIFY to a `CSymSubscribe` or a `CSymRefer` and a SUBSCRIBE or REFER to a `CSymNotify`.

As a way of encapsulating these checks, we can introduce an intermediate interface class for the three event bundles, `MSymEventCallBundle`.

```
class MSymEventCallBundle
    {
public:
    virtual TBool UsesEvent(const CSIPEventHeader&) = 0;
    virtual TBool DestinationForRequest (const CSIPServerTransaction&) = 0;
    };
```

`UsesEvent()` is self-explanatory; `DestinationForRequest()` will return `ETrue` if the incoming `CSIPServerTransaction` is of the correct type for that bundle, and `EFalse` otherwise. For example, a `CSymSubscribe` would return `ETrue` to a NOTIFY and `EFalse` to all other requests.

This gives us a bundle class hierarchy like the one shown in figure 2.

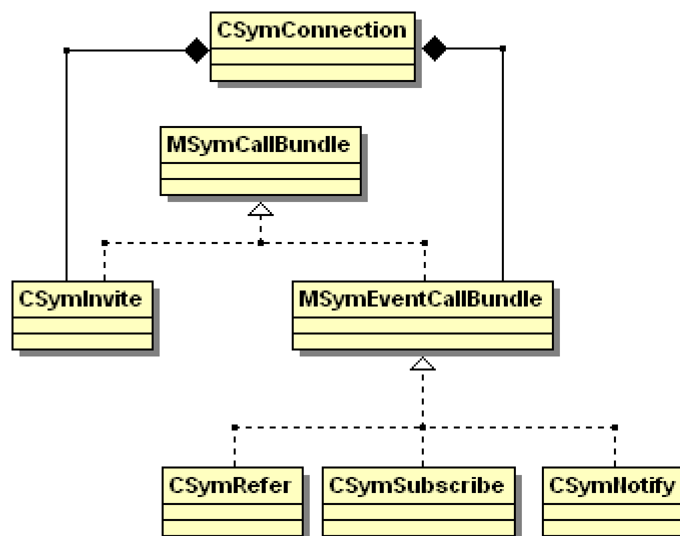

Figure 2: Call bundle class diagram

The generic internal state of each bundle is not that interesting. For invite calls, it will usually be possible to hold the state in lists of active client and server transactions, and a map of dialog associations to their current state, like so:

```
enum TInviteState
    {
    EIdle,
    ECalling,
    EDestinationRinging,
    EDestinationRepliedAcking,
    EcallUp,
    ESelfRinging,
    EReceivedAckWaitingForCall,
    ETerminated
    };
```

```
struct TInviteDialogAssocState
      {
      CSIPInviteDialogAssoc* ipDialogAssoc;
      TInviteState iState;
      };

RArray<TInviteDialogAssocState> iAssocs;
RPointerArray<CSIPClientTransaction> iClientTrans;
RPointerArray<CSIPServerTransaction> iServerTrans;
```

The multiple association states appear potentially complex, but in practice are fairly straightforward to combine. Under normal circumstances, once a call is picked up, all other dialogs are abandoned, so it's only during the negotiation phase that you even have to worry about the multiple dialog states.

For event bundles, it will usually be possible to manage the state using four variables: a client transaction, a server transaction, a dialog association, and a single state variable that holds the overall state of the bundle (e.g., attempting subscription, subscribed, etc.).

```
TSubscriptionState iState;
CSIPClientTransaction iClientTrans;
CSIPServerTransaction iServerTrans;
CSIPSubscribeDialogAssoc iAssoc;
```

This simplicity is exactly what we were aiming for when we started splitting the state. However, to support it we have to implement some quite complicated routing of messages in the connection class. Before we get on to this, we need to discuss how the connection might manage the lifetime of the calls.

There are two ways in which call bundles can be created: by the application initiating a call or subscription, or by the SIP stack notifying the application of an incoming call. The latter is tied up in the routing of requests, which will be dealt with in the next section. The former is more freeform, and so offers some scope for doing things differently.

For an actual INVITE call, the easiest way to support call initiation is to have a simple `MakeCallL()` method on the connection class that returns a `CSymInvite`. Whether this is returned as a reference or a pointer depends on the memory management strategy we adopt.

If we go for the connection owning the bundle we should return a reference, and supply a means of deleting the call on the connection class. If we decide that the caller should own the class, we return the pointer, and the caller deletes the object when they are finished with it. On the surface, the caller ownership option seems like the simpler policy, but it is somewhat complicated by the connection needing to retain a reference to the call bundle for routing purposes. So, adopting a connection ownership policy gives us a `MakeCallL()` that looks something like this, with the very act of creating the `CSymInvite` resulting in an INVITE being sent.

```
RPointerArray<CSymInvite> iInvites;

CSymInvite& CSymConnection::MakeCallL(CUri8* aDestination)
    {
    iInvites->AppendL(CSymInvite::NewLC(aDestination, *this, *ipProfile));
    CleanupStack::Pop();
    }
```

To start a subscription, you would do something very similar, only using an `RPointerArray<MSymEventCallBundle> iEventCallBundles` member instead. It would be possible to have just the one array of owned call bundles, but it would necessitate a different bundle class hierarchy and some ugly type-casting when routing.

4.2.9 Routing incoming requests

Incoming requests come in two flavours: out-of- and in-dialog. Out-of-dialog requests are relatively easy to handle: simply check if the transaction is an event transaction (i.e., REFER or SUBSCRIBE: a NOTIFY request should never be sent outside a dialog) and then create the appropriate bundle (assuming that the `CSymNotifyCall::NewLC()` will leave if it gets passed an incorrectly sent NOTIFY request). The bundles themselves should take responsibility for any further handling as part of their construction process.

```
void CSymConnection::IncomingRequestL(CSIPServerTransaction* apTrans)
    {
    if(DialogInitiatingRequest(apTrans))
        {
        if(IsEventRequest(apTrans))
            {
            iEventCallBundles.AppendL(CSymNotify::NewLC(apTrans,*this,*ipProfile));
            CleanupStack::Pop();
            }
        else
            {
            iInvites.AppendL(CSymInvite::NewLC(apTrans,*this,*ipProfile));
            CleanupStack::Pop();
            }
        }
    else
        {
        // free transaction, see above
        }
    }
```

Resolving in-dialog requests requires a bit more thought, especially with regard to the interaction with different parts of your application.

First of all, using the same dialog for multiple purposes is largely deprecated. For example, if a dialog was created by an INVITE, a SUBSCRIBE message should not be sent as part of the same dialog. However, this does not mean that it doesn't happen in practice. There therefore needs to be a degree of allowance for requests of the 'wrong' type appearing mid-dialog: you cannot necessarily blindly map a request to a dialog without first checking its type.

RFC 3265 states that an incoming NOTIFY must contain the same event header information as its corresponding SUBSCRIBE or REFER. This means that for a given NOTIFY request it should always be possible to pair it back to the `CSymSubscribe` or `CSymRefer` that caused it. Similarly, we should be able to pair a refreshing SUBSCRIBE or REFER back to the `CSymNotify` that they initially caused to be created. By using the `DestinationForRequest()` method we declared above, we can resolve these two sets of pairings into one, which means we only have to re-use our `IsEventRequest()` function and not explicitly check for the type of the transaction. So, we can happily run through the list of event bundles, and then assume that if we don't get a match the request is a new SUBSCRIBE or REFER and then we can create a new `CSymNotify`.

For non-event requests, we can do something similar, but minus the event and destination checks: we run through the list to see if we already have an active `CSymInvite`, and if not create a new one.

```
void CSymConnection::IncomingRequestL(CSIPServerTransaction* apTransaction,
                                      CSIPDialog& aDialog)
    {
    if(IsEventRequest(apTransaction))
        {
        CSIPEventHeader* pEventHeader = ExtractEventHeader(apTransaction);
        for(TInt i = 0; i < iEventCallBundles.Count(); i++)
            {
            MSymEventCallBundle* pEventCall = iEventCallBundles[i];
            if(pEventCall->UsesDialog(aDialog) &&
               pEventCall->UsesEvent(pEventHeader) &&
               pEventCall->DestinationForRequest(apTransaction))
                {
                pEventCall->IncomingRequestL(apTransaction);
                return;
                }
            }
        // no dialog and event pairing so must create one
        // this should always be a notify
        ipEventCallBundles.AppendL(CSymNotifyCall::NewLC(apTransaction,
                                                         *this,
                                                         *ipProfile));
        CleanupStack::Pop();
        }
    else // We have an INVITE
        {
        for(TInt i = 0; i < iInvites.Count(); i++)
            {
            if(iInvites[i]->UsesDialog(aDialog))
                {
                iInvites[i]->IncomingRequestL(apTransaction);
                return;
                }
            }

        iInvites.AppendL(CSymInviteCall::NewLC(apTrans,*this,*ipProfile));
        CleanupStack::Pop();
        }
    }
```

4.2.10 Routing incoming responses

Responses are generally much easier to route into call bundles than requests. Because responses are by definition already associated with a dialog, all incoming responses apart from forked INVITE responses are diverted into the `CSIPDialogAssocBase` overload of `IncomingResponse()`. To properly route the response, we simply iterate through the two arrays of bundles, testing each one until we get a positive response, then call the `IncomingResponseL()` on the bundle.

```
void CSymConnection::IncomingResponseL(CSIPClientTransaction& aTransaction,
                                       CSIPDialogAssocBase& aDialogAssoc)
    {
    for(TInt i = 0; i < iInviteCalls.Count(); i++)
        {
        if(iInviteCalls[i]->OwnsAssoc(aDialogAssoc))
            {
            iInviteCalls[i]->IncomingResponseL(aTransaction);
```

```
                return;
                }
            }

        for(TInt j = 0; j < iEventCalls.Count(); j++)
            {
            if(iEventCalls[j]->OwnsAssoc(aDialogAssoc))
                {
                iEventCalls[j]->IncomingResponseL(aTransaction);
                return;
                }
            }
        }
```

To handle forked INVITEs we also need to implement the `CSIPInviteDialogAssoc` overload. Because the association is new, we use the owned transaction instead of the association or the dialog to identify the correct object to add the new association to.

```
    void CSymConnection::IncomingResponseL(CSIPClientTransaction& aTransaction,
                                          CSIPInviteDialogAssoc* apDialogAssoc)
        {
        for(TInt i = 0; i < iInviteCalls.Count(); i++)
            {
            if(iInviteCalls[i]->OwnsTransaction(aTransaction))
                {
                iInviteCalls[i]->AddAssociationL(aTransaction, apDialogAssoc);
                return;
                }
            }
        }
```

4.2.11 Profiles

The major decision to be made with regard to profile management is whether to support a single profile or multiple profiles per connection. Single profiles are easier to deal with, but effectively limit your connection to receiving and transmitting requests on a single Address of Record (AoR) via a single proxy, unless you resort to manual registration and from/to header setting. Multiple profiles are more complicated, but offer you greater freedom with regards to handling different identities and routings on a single IAP.

4.2.12 Single profile

Implementing single profile handling is simple. Just have a `CSIPProfile*` member in your `CSymConnection` class and enable it on creation. To keep track of changes in the profiles, implement the `MSIPProfileRegistryObserver` interface in `CSymConnection`. This is the approach used in all the code snippets above.

The only complication is if you want to piggyback on the plug-in architecture to auto-start the application on incoming calls. In this case, you need to set your chosen profiles to be always registered (beware, this can have a bad effect on you battery life), and then to carefully handle what happens in the resulting call to your `CSymSession::IncomingRequest()` method.

In particular, you need to check that whatever turns up uses the correct Address of Record. If more than one profile is set to always registered (e.g., for another application), it’s possible for requests nominally sent to a different address to end up being sent to your application.

4.2.13 Multiple profiles

If you want to support multiple AoRs directing to your application on a single connection, you will have to support multiple profiles and routing-by-profile. This is not quite as bad as it sounds: a single dialog will still only have a single profile associated with it, and when operating as a UAC the application will still get to choose a specific profile and IAP to use for its requests.

It's only when dealing with incoming out-of-dialog requests that you have to think carefully about how to associate the connections and profiles. Even then, it should largely just be a case of looping through your profiles and then picking the one that matches the 'To' address of the incoming request.

```
void CSymConnection::IncomingRequestL(CSIPServerTransaction* apTransaction)
    {
    CSIPToHeader* pTo = apTransaction->RequestElements().ToHeader();
    if(!pTo)
        {
        User::Leave(KErrCorrupt);
        }

    for(TInt i = 0; i < iProfiles.Count(); i++)
        {
        if(MatchesAoR(iProfiles[i], pTo))
            {
            // take ownership and route as normal, using this profile
            }
        }

    // no match: throw error
    }
```

4.3 Error Handling

4.3.1 Sinking errors in callbacks

None of the SIP stack-defined callback methods leave. This means that everything should run in a trap, unless you never, ever call a leaving function. As mentioned previously, this can quickly get tedious, and can make code difficult to read. There are two strategies to deal with this.

The first strategy is to implement L versions of all the callback functions, wrap them in TRAPs and sink all the errors through one function.

For example:

```
void CSymConnection::HandleError(TInt aError)
      {
      ipMyApp->HandleError(aError);
      }

void CSymConnection::IncomingRequest(CSIPServerTransaction* apTransaction)
      {
      TRAPD(err,IncomingRequestL(apTransaction));
      if(err != KErrNone)
            {
            // include any specific error conversion here
            HandleError(err);
            }
      }
```

This strategy has the benefit of simplicity, a quality that should not be underestimated when dealing with the complexities of SIP. However, it also results in the loss of an awful lot of contextual information, which might well make it impossible for an application to sensibly handle the error.

The second strategy keeps more contextual information at the cost of a greater amount of complexity. It too uses a general error sink, but instead of naively wrapping L versions of the standard callbacks, it only wraps calls made once a route for a request/response has been established, and then routes the error back to a destination specific error code. For example:

```
TInt MSymCallBundle::HandleError(TInt aErr) = 0;

void CSymConnection::IncomingResponseL(CSIPClientTransaction& aTransaction,
                                       CSIPDialogAssocBase& aDialogAssoc)
    {
    for(TInt j = 0; j < iCallBundles.Count(); j++)
        {
        if(iCallBundles[j]->OwnsAssoc(aDialogAssoc))
            {
            MSymCallBundle* pBundle = iCallBundles[j];
            TRAPD(err,pBundle->IncomingResponseL(aTransaction));
            if(err != KErrNone)
                {
                HandleError(pBundle->HandleError());
                }
            }
        }
    }
```

The major drawback of this scheme is that it will not work when the routing algorithm results in the creation of a new destination and the destination's `NewL()` leaves. In that case, we have to fall back to simply sinking the error straight through the generic `HandleError()`.

4.3.2 Implementing the error callbacks

Once we have a generic error-handling sink in place, implementing the error callbacks becomes straightforward. Depending on the strategy chosen, we either sink the error straight to our generic `HandleError()`, which propagates it to the outside world, or we first resolve the errors using the additional transaction and dialog associations supplied to the callback, and then send the errors through the bundles as appropriate.

4.4 Message contents

4.4.1 Generating messages

As suggested above, it is a good idea to write separate functions for each type of required message response, returning a pointer to the resulting message that you have left on the cleanup stack. For example, a function to generate a `CSIPMessageElements` object should follow this pattern:

```
CSIPMessageElements* XyzMessageElementsLC(TSomeArg aArg)
    {
    CSIPMessageElems* pElems = CSIPMessageElems::NewLC();
    FillXMessageElements(*pElems, aArg);
    return pElems;
    }
```

which allows it to be used like this:

```
iTrans = pDialogAssoc->SendInviteL(XyzMessageElementsLC(aArg));
CleanupStack::Pop();
```

This is much cleaner than the equivalent:

```
CSIPMessageElements* pElems = MessageElementsL(aArg);
CleanupStack::PushL(pElems)
iTrans = pDialogAssoc->SendInviteL(pElems);
CleanupStack::Pop(pElems);
```

Further, using `FillXyzMessageElementsL()` to wrap the actual filling of the object allows the sharing of the filling code between in-dialog requests (which use `CSIPMessageElements`) and out-of-dialog requests (which use `CSIPRequestElements`):

```
CSIPRequestElements* pReqElems = CSIPRequestElements::NewLC(pUri);
FillXyzMessageElementsL(pReqElems->MessageElements());
```

We can theoretically go one step further and genericize `XyzMessageElementsLC()` by defining a mixin interface:

```
class MMessageElemFiller
    {
public:
    void FillL(CMessageElements& aElems) = 0;
    };

void MessageElementsLC(MMessageElemFiller& aFiller)
    {
    CSIPMessageElems* pElems = CSIPMessageElems::NewLC();
    aFiller.FillL(*pElems);
    return pElems;
    };
```

The mixin would be used like this:

```
class TXyzMessageFiller : public MMessageFiller
    {
public:
    TXyzMessageFiller(TSomeArg aArg);
    void FillL(CMessageElemensts& aElems);
    TSomeArg iArg;
    };

TXyzMessageFiller filler(aArg);
iTrans = pDialogAssoc->SendInviteL(MessageElementsLC(filler));
CleanupStack::Pop();
```

In practice this step isn't worth the trouble unless you find you are writing an awful lot of `XyzMessageElementsLC()` methods.

4.4.2 Reading headers

In general, the best way to interrogate the headers of a message is to iterate through them one by one. However, it is sometimes necessary to pull out a single header quickly. In this case, the following generic function can be very useful:

```
template <class Header>
Header* ExtractUserHeader(const CSIPMessageElements* apElems, RStringF aName)
    {
    RPointerArray<CSIPHeaderBase>& userHeaders = apElems->UserHeaders();
    for(TInt i = 0; i < userHeaders.Count(); i++)
        {
```

```
            if(userHeaders[i]->Name() == aName)
                return (Header*)userHeaders[i];
            }
        return NULL;
        }
```

4.4.3 SDP bodies

The basic generation of SDP message bodies is straightforward: create a document and then fill it with fields of the appropriate types.

Once we have filled our document, and want to generate a text string to include a message, the `CSdpDocument` class has an encode method on it for producing correctly formatted SDP text. Unfortunately, this takes an `RStreamWrite` reference, which can be awkward to handle when you want an `HBufC8` to pass to `CSIPMessageElements::Body`. The S60 SDK SIP example provides a method for wrapping this awkwardness, but the following code gives a slightly neater way of doing it:

```
    HBufC8* EncodeSdpDocAsBufLC(CSdpDocument* apDocument)
        {
        if(!apDocument->IsValid())
            {
            // Define your own error code if necessary
            User::Leave(KErrArgument);
            }

        CBufFlat* pStreamBuf = CBufFlat::NewL(256);
        CleanupStack::PushL(pStreamBuf);

        RBufWriteStream writeStream(*pStreamBuf);
        CleanupClosePushL(writeStream);
        apDocument->EncodeL(writeStream);
        CleanupStack::PopAndDestroy(); //writeStream

        HBufC8* pEncodedBuf = pStreamBuf->Ptr(0).AllocL();
        CleanupStack::PopAndDestroy(pStreamBuf);

        CleanupStack::PushL(pEncodedBuf);
        return pEncodedBuf;
        }
```

This version has several advantages. First, it cleans up its resources properly. Secondly, all error conditions result in a leave. Finally, it always returns a pointer to a valid buffer, meaning that it can be used directly in another function leaving call that takes ownership, with no need to use a temporary and check for `NULL`. All that needs to be done is to pop the cleanup stack once we're done, in the same way as described above for message generation.[12]

Reading the document bodies is equally straightforward: the `CSdpDocument` class has a full range of largely self-explanatory accessors for all the field types.

---

[12] This function pattern is useful in general: template it on the parameter type, drop the validity check and replace the `EncodeL()` method with `ExternalizeL()`, and you have a generic `MarshalForIpcLC<T>` function

4.5 The client resolver plug-in

The resolver plug-in architecture allows you to register interest in incoming, out-of-dialog and dialog initiating requests. The concept is that you supply the SIP stack with some data describing the type of requests you are interested in. The stack then uses this data to match against incoming messages and, if a given request matches, sends the request to your application, first asking you to start it if necessary.

In practice this means you have to implement the `CSIPResolvedClient` interface as part of an ECOM plug-in. This has three important methods: `Capabilities()`, `ChannelL()`, and `ConnectL()`.

`Capabilities()` is used by the stack to get the basic matching criteria in the form of a descriptor filled with XML (this is discussed below). It's also possible to statically implement this, by including the XML in the RSS file of the plug-in. In this case, `Capabilities()` itself is never called.

`ChannelL()` is used by the stack to get the UID of the `CSIP` to which the request should be sent, passing in the entire request as it does so.

Finally, `ConnectL()` is called when the stack cannot find an active `CSIP` with the supplied UID. It then expects the implementation to create an appropriate `CSIP` object, usually by starting a new instance of the relevant application.

4.5.1 How the matching works

The `Capabilities()` XML document can contain multiple instances of four types of data: three varieties of header information, Accept-Contact, Allow-Events and Accept, and SDP media lines. These are used for matching against the headers and bodies of incoming requests.

Accept-Contact is an extension header specified in RFC 3481, describing the general types of capabilities that a message might require and a user-agent support. These capabilities include things like level of mobility and whether the user-agent supports audio, rather than the more specific protocol specifications found in the SDP media lines. In the case where multiple destinations are registered for a single AoR, the Accept-Contact header is used by intermediate proxies to determine the correct destination for the message, As such, its use in the plug-in is a natural extension of its usage in the wild.

Accept is a standard header specifying just the content-type of the message itself. This takes the form of a MIME type, such as '`application/sdp`'.

Allow-Events is another standard header, specifying the types of events that a user-agent can offer. According to RFC 3265, if a user agent supports events, it should be included in all dialog initiating requests and response, and in OPTION responses.

The SDP media lines are simply the media specific lines you would find in a standard SDP document, but with some fields set to fixed values. For more details, see the XML DTD in the Symbian guide.[13]

Unfortunately, from the documentation, it is not clear which of these takes precedence. Nokia have supplied a flowchart online,[14] but it doesn't mention Allow-Events. However, it does at least show

---

[13] See http://www.symbian.com/developer/techlib/v9.3docs/doc_source/guide/Multimedia-Protocols-subsystem-guide/SIP/SIPClientResolverAPI/ClientResolverAPIoverview.html#mmprotocols%2esip%2eclientresolverapi%2eoverview

[14] See http://wiki.forum.nokia.com/images/8/8a/Client_Resolver_logic.pdf

that the matching prioritizes Accept-Contact over Accept, and that the SDP media lines are (perhaps surprisingly) the least significant of all.

4.5.2 Implementing the interface

The Nokia S60 SDK example provides an example implementation of the interface, but it is perhaps less clear than it might be. This is particularly so in `ConstructL()`, which goes to great pains to enumerate all the interface implementations contained inside the plug-in. For a typical plug-in, only a single implementation will be supported, and only a single `CSIP` UID used, so this is rather overcomplicated.

The implementation of `Capabilities()` should just return a single static descriptor containing the XML document. As discussed above, the precedence algorithm first checks the Accept-Contact headers, then the Accept headers, and only then SDP media lines. This means that if you are aiming to act as a broad handler for third-party calls, you need to make sure that you include an equally broad range of Accept-Contact classes in your XML. It also means that if you wish to be considered for matching via the SDP media lines, you must always include the MIME type ‘`application/sdp`’ in your ACCEPT field, like so:

```
<SIP_HEADERS>
<ACCEPT value="application/sdp"/>
</SIP_HEADERS>
```

`ChannelL()` offers the possibility of diverting requests to different `CSIP` instances, based on the message itself. In practice, it is unusual to do anything but return the UID of a single instance, but in theory there is nothing to stop an application using multiple `CSIP` instances, with different UIDs. You could then selectively direct requests to the application by AoR, for example.

Under normal circumstances, `ConnectL()` should indirectly create the `CSIP` object by starting the application that will eventually create it. If you have chosen to use the UID of the application as the `CSIP` UID, you simply use the passed in UID. It’s important to note that the generic method for application starting, `RApaLsSession::StartApp()`, does not work reliably for user interface applications on some UIQ 3 devices. To be on the safe side, you should use `QikFileUtils::StartApp()` for all UIQ 3 implementations.

5 Conclusion

In this white paper, the second and final part of the IMS series, we covered some of the technical aspects of using SIP and SDP on Symbian. To begin with, we conducted a tour of the main APIs, elaborating and offering commentary on the reference documentation. This was followed by a series of examples demonstrating the use of the APIs, and some of the problems you might face when using it. Finally, the last part of the paper was concerned with suggesting ways of structuring code that interfaces directly with the SIP stacks, offering an alternative architecture to that used in the S60 example code.

After reading this paper in conjunction with the first in the series, you should be in a better position to write SIP-based smartphone applications that take full advantage of the Symbian OS SIP stack. Even if you do not use our suggested architecture, we hope that illustrating a particular way of structuring code has helped you view the APIs from a new angle.

6 Biography

Founded in 2004, Symsource (www.symsource.com) has developed a reputation for reliably delivering complex and innovative solutions in the wireless broadband arena, particularly in innovative

SIP-applications. Working in collaboration with our customers, we ensure their business needs are addressed in the most efficient manner, mindful of the total cost of ownership of the solution.

Symsource has built a library of mobile utilities that can be deployed rapidly into our clients' internet-enabled value-added services, meaning that our connected solutions always have a head start. This library encompasses server-side and client-side components built by a team with the foremost expertise in mobile development. For examples of the type of application enabled by Symsource's technology, see www.motxt.com.