Cisco Unity Connection .NET Notification SDK

Contents

Overview

The .NET SDK library for the Unity Connection notification interfaces is a set of library code intended to make development of applications using .NET framework easier, faster and less error prone. This SDK wraps the CUNI (Cisco Unity Notification Interface). This includes setting up HTTP listeners and raising simple events for debug, error and message events that are easy to consume in any .NET application including forms applications and web services if you choose.

This library is often used in conjunction with the REST SDK for Unity Connection which gives access to the administration interfaces, phone and messaging interfaces to Connection. For instance creating an application that responds to message events and then does something with that message (i.e. such as archiving it or sending it to a remote location or the like) is trivial using these two SDKs together.

The CUNI interface is intended for bulk notifications or, as the kids say, “server to server” notifications. In other words it designed to monitor lots of mailboxes and raise events for updates to those mailboxes to one or more servers processing those events in bulk. This differs from the CUMI COMET notifications that are designed to use a “long polling” technique

for giving individual clients updates about a single mailbox they care about (i.e. theirs). The REST for Users SDK offers a simple wrapper around the CUMI notification interface if that's what you application calls for.

Special note: The HTTP requests that are issued from Unity Connection to the callback URIs setup for CUMI are HTTP, not HTTPS. A secure pipeline is not established for each request in other words. The data contained in the notification is all pretty generic and does not present a serious security risk: the user that owns the message is identified only by a GUID as is the message ID. Information about what the event is (new or deleted message for instance), the sender's ANI (if provided) and when that event happened is included in the body which can be "sniffed" on the network. Just be aware that this is "in the open".

Requirements/Special Notes

The .NET Notification SDK is written and tested against all versions of Unity Connection 8.6 and later. The CUNI interface is supported in 8.5 builds but is missing some functionality. The SDK does not do version checking for you, if you call a method that's not supported the server error information that comes back will indicate this.

Use of the SDK is not supported by TAC – in other words if you need assistance with an application you are developing using the SDK, TAC is not going to provide that assistance for you. Support is through development services and the public support forums. Check the www.CiscoUnityTools.com site on the "links" page for current links to get to those forums. "Not supported by TAC" does not mean the SDK is somehow dangerous or risky to use. To be clear it simply wraps the existing supported CUNI APIs provided with the platform – it does not go outside the bounds of those protocols and applications developed using the SDK are just as safe and supported as those written directly to the API.

Any .NET framework can use these libraries. This means you can, of course, develop desktop and web applications on Windows using C#, VB.NET, Iron Python etc… but you can also use Mono for development of applications on Mac or Linux platforms as well. It's even possible to build a web server using .NET MVC4 that will run on your Mac if that's what you want to do (although I'm not sure that's a great deployment model). Training videos will be made available off www.CiscoUnityTools.com that will walk through the process of using the SDKs which will include some details about doing cross platform development. .NET isn't just for Windows any more!

The library is build and tested using Visual Studio 2010 and Visual Studio 2012 in Windows, however I do validate it works with Mono Develop and Xamarin Studio on Mac.

Installing and Using the Library in Your Project

The Easy Way

The SDK is available via NuGet package manager and by far the easiest way to install it is to manage NuGet packages for your project, select online and search on "CUNI" or "Cisco Notification SDK" or any number of other combinations of Cisco Unity Connection Notification SDK" and the project should come right up. When you add it the dependent libraries are automatically installed and all you need to do is add the "**Cisco.UnityConnection.NotificationFunctions**" to your using list and you're off to the races.

If the search is not working properly or you're in an environment such as on a Mac using MonoDevelop and you only have access to the NuGet package manager console you can install the package by issuing the following command:

Install-Package Cisco.UnityConnection.NotificationSdk

NuGet is nice since it always grabs the latest and notifies you when an updated version of the library is available and gives you the option of installing it. Couldn't be easier.

The Hard Way

You can still download the source code for the project and include it manually. This can be handy if you prefer to debug into the SDK library code directly and see what's going on or the like or if you just don't trust NuGet or doing things the easy way makes you generally suspicious. Whatever works for you.

Using any SubVersion client you like you'll need to download the SDK project code off the site's download page: http://www.ciscounitytools.com/CodeSamples/Connection/Notifications/NotificationSdk.html

To add the project right click on your solution node in the solution explorer in Visual Studio. Select "Add" and then "Existing Project" and a file explorer window will appear. Navigate to where you downloaded the library code and select the "UnityConnectionNotificationSdk.csproj" file. This will pull the library into your solution and have it build when you rebuild your project. This will result in the "UnityConnectionNotificationSdk.dll" ending up in the target BIN output (debug or release) for your project.

Once you've included the project you then need to add a reference to it in your project – in your project right click the "references" node in the solution explorer and select "add reference" – in the resulting dialog select "projects" and select the UnityConnectionNotificationSdk project and add it. Then you only need to add a "using `Cisco.UnityConnection.NotificationFunctions`;" directive in your project and you're off to the

races. . The full project includes a couple different project examples such as a simple CLI application and a basic WinForms project demonstrating some of the basic capabilities of the library.

NOTE: The Notification SDK requires the .NET 4.0 framework but will work with either the full or "client" versions of the framework. The full REST SDK requires the full 4.0 framework but the notification SDK can get along with the smaller subset of the .NET library if this is how you want to deploy your application.

Watching the traffic with Fiddler2

I highly recommend you download and install Fiddler2 on your development machine so you can watch the traffic going to and coming from Connection while you're using the library. In the case of the notification SDK you will see anything originating from your application and going to Connection. These will be SOAP requests for setting up HTTP notification callbacks, adding mailboxes to the monitored list and such. You will not see the HTTP requests coming in from Unity Connection in Fiddler. However you can hook the debug event monitor in the SDK to see these raw HTTP requests if you need to – see the Logging and Debugging section for more on that.

For customers that don't want to use the .NET wrapper library for their projects but want to get a jump start on seeing how commands and requests should be formatted and what they return this can be a very fast way of doing that.

Using the .NET Notification SDK

This document uses a "task based" approach to demonstrating the use of the library – As a developer I know I learn faster with a simple "show me" approach so that's what I endeavor to do here.

Getting Started

Registering for Callbacks from Unity Connection

Unlike the REST SDK which conceptually involves a "pipeline" to and from Connection for adding/removing/finding and updating various objects in the directory, the Notification SDK is all about letting Connection know what you want to hear about with regards to mailbox changes and then handling those notifications when they come in. Pretty simple, really. The amount of communication going out from your application to the server will be relatively tiny compared to a typical REST based administration application.

So, the exercise here is to announce to Connection which mailboxes you'd like to monitor and how you'd like those notifications to get to you. The SDK is nice enough to handle most of the tedium of setting up HTTP listeners and processing those inbound notifications for you and provides simple event triggers for easy handling in your application. You also have the option to instead dictate callback URLs yourself and the like, however first let's do it the easy way.

```
//first, create the monitor class instance for a Unity Connection server.
try
{
    _cxnMonitor = new UnityConnectionEventMonitor("192.168.0.186", "CCMAdmin",
            "ecsbulab", 8080, "", 10, DateTime.Now.AddDays(5), true,"operator");
}
catch (Exception ex)
{
    Console.WriteLine("Failed to create event monitor:"+ex);
    return;
}
```

OK, so what's that doing? It's connecting to a Unity Connection server at "192.168.0.186", authenticating as an administrator (CCMAdmin account) and tell it you wish to get callbacks for notification events on this instance on port 8080 for 5 days and to include the operator mailbox in the list of mailboxes to be monitored. In this case you're telling the class to setup HTTP listeners for you and construct the callback URL on your behalf. All you need to do is provide a unique port to use (8080 here).

The "10" there is an indication of how often you'd like Connection to send a heartbeat in minutes. You can pass this as 0 and no heartbeat will ever be sent, however this is not a great idea. If your application goes off line or network connectivity is lost you don't want Unity Connection continuing to broadcast HTTP notifications for mailboxes on that subscription until it runs out (in 5 days in this case). If Connection sends a heartbeat and you don't respond it unregisters the subscription and stops sending out updates. The SDK automatically responds to heartbeats for you in this case (assuming your application is running and the server you're running on is visible on the network of course.

Also notice that "operator" is sent there – you can send as many user aliases as you like in a list (array of strings) or chunking in the params at the end. However you must provide at least one alias to include or the SOAP call to

Connection will fail. Normally you already have a list of mailboxes to monitor in hand or you wouldn't be setting up a monitor so this isn't a huge deal but if you find yourself needing to create a monitor without knowing any aliases to pass, using "operator" is usually pretty safe.

So, what's up with the port and why do you care? The short version is you want to limit the number of mailboxes being monitored on a single subscription to about 500 mailboxes. So to monitor all mailboxes on a server several instances will be necessary, which is fine – you just need to provide a separate port for each one. For instance start at 8080 and increment by one for each new monitor you create as you add more users to the list being monitored.

In the case above, the URI that will be used for callbacks on this monitor will look like this:

```
http://192.168.0.186:8080/MessageEvent/AllEvents
```

A lot has been done already, you just need to wire up a couple of event handlers and sit back and respond to them. Unity Connection has now added the above URI to its notification queue and any mailboxes you've identified for notification will send HTTP requests to that URI for any event on that mailbox (new message, saved message, deleted message). We'll explore the properties and methods available off the event monitor class later, but for now let's just wire up the message event handler so our application can know when a new message event comes in and if there's any errors we need to respond to. It's as easy as adding a method handle to the ConnectionEventstoBeProcessed event like this:

```
//setup an event handler to fire when a message event from a mailbox you've told the
//monitor to watch triggers.
_cxnMonitor.ConnectionEventsToBeProcessed +=
            CxnMonitorOnConnectionEventsToBeProcessed;
…


/// <summary>
/// Simple message event handler
/// </summary>
private static void CxnMonitorOnConnectionEventsToBeProcessed(object sender,
            MessageEventArgs e)
{
    Console.WriteLine("[MESSAGE EVENT] "+e.MessageEvent);
}
```

Obviously not a very interesting event handler there – just dumping the details to the console. But with less than a page of code (which you can see in the CliEvenMonitor project in the SDK solution) you've created an application that will spit out message events to the console as you call in and leave/retrieve messages. The console output in this case will look like this:

```
[MESSAGE EVENT] NEW_MESSAGE, at:[6/9/2013 10:27:04 AM] for:operator,
message:Normal-Priority Voice arrived at [6/9/2013 10:27:04 AM],
subscriptionId=4e5e1cff-9be2-453c-858e-420fbf7c398b
```

We'll break down the details that are available off the MessageEvent class passed back in the error handler below, however you get the general idea. It include what type of event (new message in this case) who it was for (operator), it's priority, when it arrived and the subscription ID (generated by Connection when you register for notifications) it's for. Details of the message itself (notably its ID so you can go fish it's details from the message store if you have administrative rights to do so). More on that later.

Keep Alive Events

Keep alive messages are special and are hence on their own event handler. In some cases you're not interested in seeing keep alive messages at all and you can configure your notification subscription to not send them. This, however, is a bad idea in general. It's a good idea to have keep alive messages sent every 5 or 10 minutes for all subscriptions such that you don't have "dead" subscriptions hanging out sending HTTP requests out that are not being handled by any client. Further, the keep alive messages contain the name of the server that actually sent the message in addition to the subscription ID that's included with "normal" event messages. This is important if you're working with clusters and you want to know when control passes from one server to another in the cluster – the keep alive messages will tell you this whereas other message events like new/delete will only include the subscription ID which may mean it comes from either server in the pair.

Setting up monitoring of the keep alive messages from the SDK is simple – just wire in a method to handle the ConnectionKeepAliveEvents off the monitor instance like this:

```
//setup an event handler to fire when a keep alive event from a server you are
//monitoring comes in
_cxnMonitor.ConnectionKeepAliveEvents += CxnKeepAliveEvent;


/// <summary>
/// Simple keep alive event handler
/// </summary>
private static void CxnKeepAliveEvent (object sender, KeepAliveEventArgs e)
{
    Console.WriteLine("[KEEP ALIVE] "+e.KeepAlive);
}
```

Note that the server name and IP address that sent the keep alive can be fetched from the KeepAliveEventArgs instance shown above in the KeepAlive instance in the properties named “serverHostname” and “serverIP” respectively.

Logging and Debugging with the SDK

Since I’ve been asked a few times, let me just state up front here that the SDK is not *supposed* to log to a file on the hard drive for you. Since the SDK can be (and is) used in a variety of application types such as desktop applications and web servers it cannot assume access to the local file system for logging purposes. It does provide a few event handles you can wire up to provide more “dialog like” logging in your application if you prefer and/or can provide more diagnostic output you can handle as you like at your application level as disused in this next section.

The UnityConnectionEventMonitor object exposes a couple of events you can use if you wish to be notified of any error and, optionally, debug event data that you can “hook” in your application to provide a more “dialog” logging output for instance. To wire up the error event would look like this:

```
//setup an event handler to deal with any errors raised by the class
_cxnMonitor.ErrorEvents+= CxnMonitor_ErrorEvents
…


/// <summary>
/// Simple error handler: just dump the error string to the console
/// </summary>
private static void CxnMonitor_ErrorEvents(object sender, LogEventArgs e)
{
    Console.WriteLine("[ERROR] "+e.Line);
}
```

The error information is simply provided as a string that comes up from the class – the SDK does not break down error IDs and failure types, this is provided for logging purposes only so a simple string is all that’s provided.

Similarly you can wire up a debug handler, however this will be quite a bit more “chatty” than the error handling. The raw HTTP requests coming in from Unity Connection as well as all heartbeat events (not normally presented to you since they’re handled automatically) are included in the events. Make sure you need this level of detail before wiring it up.

```
//deal with debug events raised
_cxnMonitor.DebugEvents += CxnMonitorOnDebugEvents;

…


/// <summary>
/// Simple debug event handler
/// </summary>
private static void CxnMonitorOnDebugEvents(object sender, LogEventArgs e)
{
    Console.WriteLine("[DEBUG] "+e.Line);
}
```

Not too tricky. I highly encourage folks to wire up and log/alert on error events but leave the debug events out of the picture unless you have a driving need for them in a particular scenario.

Event Monitor Methods

Editing Monitored Mailbox List

You can add additional users to a subscription or remove them using the user's aliases via two methods off the class provided for this:

```
_cxnMonitor.AddAliasesToSubscription("jlindborg","jsmith");

_cxnMonitor.RemoveAliasesFromSubscription("ajackson");
```

You can stack in as many aliases as you like in either method or pass an array of strings instead.

Note that it's recommended that you don't monitor more than 500 mailboxes per subscription – which correlates to each instance of the UnityConnectionEventMonitorClass. If you have a single instance approaching 500 simply create a new instance on a new (unique) port and start adding new users to that.

Converting Milliseconds into Local/UDT Time

Times of events and message arrival times are presented as a long integer which is milliseconds from 1/1/1970. The event monitor class provides a simple static method for converting this into local time or UTC.

```
DateTime oDate;

//converts to LocalTime
oDate=UnityConnectionEventMonitor.ConvertFromSecondsToTimeDate(eventTime);

//converts to UTC
oDate=UnityConnectionEventMonitor.ConvertFromSecondsToTimeDate(eventTime,false);
```

Checking Monitored Users

There are two methods you can use to get a list of users currently being monitored by a subscription to Unity Connection. One returns a list of user aliases and one returns a list of user ObjectIds (GUIDs).

```
//returns an array of ObjectIds
string[] strObjectIds;
_cxnMonitor.GetSubscriptionUserIdList(out strObjectIds);

//returns a list of aliases
List<string> strAliases;
strAliases= _cxnMonitor.MonitoredAliasList;
```

The array of ObjectIds is authoritative in this case as it is fetched via a SOAP call from Unity Connection directly. The alias list is merely a local property off the class that's updated when you add/remove aliases from a subscription. If users are added/removed via another class instance (possible but not a good idea) or a user has been removed in Connection's database or the like, this list will not be accurate.

It's best to use the ObjectId list if reporting on the actual users the subscription is currently monitoring is important to your application.

Checking Status

You can check the status of a current subscription using the GetSubscriptionStatus method:

```
SubscriptionInfo subInfo;
_cxnMonitor.GetSubscriptionStatus(out subInfo);

Console.WriteLine("SubscriptionID={0}, set to expire:{1}",
    subInfo.subscriptionId,subInfo.expiration);
```

Unsubscribing and Re-Subscribing to Events

When you dispose of the class it automatically sends a SOAP request to Connection tearing down the subscription. However you may want to "recycle" the subscription without disposing of it entirely. The list of aliases you've added and removed from the class instance are preserved using this method and those aliases are automatically included in the new subscription:

```
//tears down subscription on Connection and turns off HTTP listeners
_cxnMonitor.UnsubscribeToEvents();

//Establishes a new subscription and HTTP listeners using the existing list of
//aliases stored in the class property
_cxnMonitor.SubscribeToEvents(8081, DateTime.Now.AddDays(5), 10);
```

Monitoring Multiple Connection Servers

Monitoring multiple Unity Connection servers is simple – just create separate instances of the monitor class against each of them. Be sure, however, to use separate ports for every instance of the class regardless of if multiple instances are receiving notification from the same or separate Unity Connection servers. All inbound HTTP traffic for each subscription for notification from all Connection server must come in on their own ports.

Note that the platform you're running on may impose limitations for how many HTTP listeners can be spun up. For instance if you are running on Windows XP you may be limited to as few as 5 total. You will get an error when you go to create a new instance of the class (assuming you are having it establish new HTTP listeners for you). More modern versions and server versions of Windows will have many more sessions supported, of course.

Dictating Custom Callback URIs

If you are making a web service or you're just a "take charge" kinda person, you may wish to provide your own HTTP callback handling code in which case you will not want the class establishing its own listeners and doing its own event handling logic even though you do want to use the class wrapper around the SOAP driven CUNI API. You can do this by passing your own URI into the class constructor and providing logic to handle that on your own. The constructor for this would look like:

```
_cxnMonitor=new UnityConnectionEventMonitor("server22.myco.com","Admin","password",
    8080,"MyMessageEvent/AllEvents/Server22",10,DateTime.Now.AddDays(5),false);
```

In the above example the full URI hit would be:

http://server22.myco.com:8080/MyMessageEvent/AllEvents/Server22

If you're making a web service you'll then want to have a controller in place to handle inbound messages to that URI that contain a SOAP body that contains the message notification details. This can be found in the CUNI documentation but I'd also point out that the raw text of all notification events can also be displayed using the DebugEvent discussed in the Logging and Debugging with the SDK above if you'd like to go that route instead.

The Message Event Details

When the ConnectionEventsToBeProccessed event is raised, the method registered for it in your application will receive an instance of the MessageEventArgs class which contains a link to a MessageEvent class instance. The MessageEvent contains 6 properties and an instance of the MessageInfo class with more details of the message.

MessageEvent:

- **SubscriptionId**. GUID of the subscription ID from Unity Connection. Needed for asking Connection for status and monitored resources list for that subscription.
- **EventType**. New Message, Saved Message, Deleted Message, Deleted Message Created, Deleted Message Deleted, Unread Message or Keep Alives. The class handles responding for Keep Alive messages for you so you'll only ever see the first 6 event types. New messages are newly arrived (unheard) messages. Unread Message

event is used when the user marks an existing read message unread again. This distinguishes it from a newly arrived message. Note that the deleted message created and deleted message deleted are events added in 10.5 and are specific to messages in the deleted items folder if that option is enabled.

- **EventTime**. Milliseconds from 1/1/1970. You can use conversion method provided by class to convert it to local time or UTC as noted above.
- **MailboxId**. The alias of the user that owns the message that triggered the notification.
- **DisplayName**. The display name of the user that owns the message that triggered the notification. DisplayName can be blank for users so this is not guaranteed to be filled out.
- **USN**. Update Sequence Number generated by Unity Connection for keeping notification events straight.
- **MessageInfo**. Instance of the MessageInfo class mentioned next.

MessageInfo:

- **MessageId**. GUID identifying the message in the mailstore.
- **RecieveTime**. Long integer which is the number of milliseconds from 1/1/1970. You can use the static method provided off the class for converting into local or UTC time.
- **MsgType**. Can be Voice, Fax, Text, NDR (non delivery receipt) , DR (delivery receipt) or RR (read receipt).
- **Priority**. Urgent or normal.
- **Sender**. SMTP address of sending party.
- **CallerANI**. ANI details of sender if provided.

Batch Events

TBD

Revision History

Version 1.0.11 – 7/9/2014

- Updated for deleted items folder events in the 10.5 release

Version 1.0.8 – 9/21/2013

- Updated package manager to include XML in output to properly show comments on parameters and method signatures in the development environment.

Version 1.0.7 – 8/18/2013

- Broke the keep alive events in their own event handler with their own details to make monitoring server changes in cluster configuration easier.

Version 1.0.5 – 8/13/2013

- Packaged for NuGet download

Version 1.0.4 - 6/9/2013

- First public version of SDK and documentation.