IPDR Deployment Guide

High Level Overview

CPAN Perl Module IPDR

http://search.cpan.org/search?query=ipdr&mode=all

Andrew S. Kennedy

Version 0.1 (DRAFT)

Document Control

Change History

Date	Author	Version	Change Reference
18th November 2009	Andrew S. Kennedy	0.1	Initial draft.

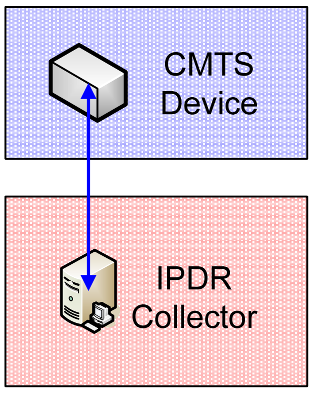
Distribution

Name	Role	Approval Req

Contents

1. Overview

IPDR is a protocol to allow the collection of usage data from a variety of devices to provide additional information or revenue generating opportunities. The IPDR specification allows multiple types of implementations from client to server and data encoding types either XDR or XML.



The IPDR collector can either retrieve the data from the CMTS device or it can be sent from the CMTS device depending on the implementation by the CMTS vendor. All IPDR connections are made using TCP and some vendors also support SSL.

1.1. Data Encoding

The IPDR protocol has two methods for encoding data either XML or XDR however both provide the same type of data but this does determine the session and transport mechanisms being employed.

1.1.1. XML Encoding

An example of XML encoding can be seen below and shows the header, body and footer of the XML created.

```
<?xml version="1.0" encoding="UTF-8"?>
<IPDRDoc
xmlns="http://www.ipdr.org/namespaces/ipdr"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="DOCSIS-3.1-B.0.xsd"
docId="CEAE9DC7-0000-0000-0000-000000000000"
creationTime="2009-11-18T16:22:31Z"
IPDRRecorderInfo="test"
version="3.1">
<IPDR xsi:type="DOCSIS-Type">
<IPDRcreationTime>2009-11-18T16:22:31Z</IPDRcreationTime>
<CMTShostName>test</CMTShostName>
<CMTSipAddress>192.168.1.1</CMTSipAddress>
<CMTSsysUpTime>61878200 </CMTSsysUpTime>
<CMTScatvIfName>Cable5/0</CMTScatvIfName>
<CMTScatvIfIndex>9</CMTScatvIfIndex>
<CMTSupIfName>Ca5/0-upstream3</CMTSupIfName>
<CMTSupIfType>129</CMTSupIfType>
<CMTSdownIfName>Ca5/0-downstream</CMTSdownIfName>
<CMmacAddress>00-02-8A-68-DD-54</CMmacAddress>
<CMipAddress>172.26.65.8</CMipAddress>
<CMdocsisMode>10</CMdocsisMode>
<CMCPEipAddress></CMCPEipAddress>
<RecType>1</RecType>
<serviceIdentifier>77</serviceIdentifier>
<serviceClassName></serviceClassName>
<serviceDirection>2</serviceDirection>
<serviceOctetsPassed>759477</serviceOctetsPassed>
<servicePktsPassed>9077</servicePktsPassed>
<serviceSlaDropPkts>0</serviceSlaDropPkts>
<serviceSlaDelayPkts>874</serviceSlaDelayPkts>
<serviceTimeCreated>1850800</serviceTimeCreated>
<serviceTimeActive>600274</serviceTimeActive>
</IPDR>
<IPDRDoc.End count="1" endTime="2009-11-18T16:22:32Z"/>
</IPDRDoc>
```

The basic structure of the XML document is broken down into three major components header, body and footer.

Header

The header of the XML document describes the specification used, date, the name of the IPDR device sending the data and the version being used.

Body

The body of the document should contain two (2) entries per device being reported. The two entries reflect both the upstream and downstream datasets made available.

Footer

The footer of the document should only contain the number of records within the body and a final timestamp as to when the data was made available.

Known vendors to use the XML encoding mechanism are Cisco Systems for the UBR7200 and UBR10k CMTS products.

1.1.1. XDR Encoding

XDR encoding cannot easily be described within this document as it a compact structure and has no textual elements but purely hexadecimal transport (data) mechanism.

When using XDR encoding the collecting agent retrieves a template describing how each field is encoded thus providing the mechanism to convert into a more humanly readable form if required.

Details of XDR encoding can be found within the IPDR specification at

http://www.ipdr.org/public/DocumentMap/XDR3.5.1.pdf

2. Client/Server Implementations

The IDPR CPAN module should provide functionality for two (2) types of implementations however one is vendor specific but can also be used as a TCP client/server implementation if required.

2.1. Cisco Specific Implementation - SAMIS

The Cisco implementation comes in two flavors (may be merged in the future). These are non-secure and secure and can be reference as Cisco.pm or CiscoSSL.pm when being included with any other code.

Cisco has implemented a client within their CMTS product range that will connect to a remote server and send data via a TCP connection. The transport of the data is done via XML encoding and an example can be seen in the encoding section of this document. It should be noted that certain versions of Cisco IOS are known to send incomplete or corrupt XML segments and this Cisco.pm and CiscoSSL.pm attempt to work around this problem by only throwing away the specific record which is corrupt, not the entire XML. This may break certain XML conventions however at the moment there are no ways around this.

The Cisco modules are TCP servers and can be configured to listen on any TCP port (with correct privileges) and also use SSL certificates and Host Keys if desired. They can also be used to accept relay data from either a collecting server or Client collecting data.

This implementation supports the SAMIS features of the Cisco CMTS product set.

2.2. Generic IPDR XDR Collection Client

The generic implementation provides an IPDR client with XDR support which can connect to any IPDR server, maintain a connection and process data sent. The client implementation allows keep alive time and capabilities to be set along with the ability to relay the data to a secondary host via clear or a secure (SSL) connection.

By default the IPDR protocol has no defined security and this module allows the data to be collected locally and then relayed either clear or with SSL to a remote processing server. Using this module for just relaying takes very little processing CPU.

Using later versions of IOS such as 12.4 means the client version should be used when connecting to Cisco CMTS routers in preference to the Cisco module with SAMIS support.

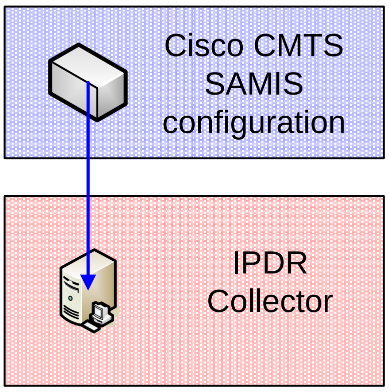
3. Deployment Scenarios and Code Examples

3.1. Cisco UBR Series

The UBR7200 only supports 32bit counters and UBR10k only supports 64bit counters. This must be taken into account when deploying any IPDR accounting.

3.1.1. Cisco Deployment – 7200 and 10k – SAMIS

Conceptual Flow Diagram



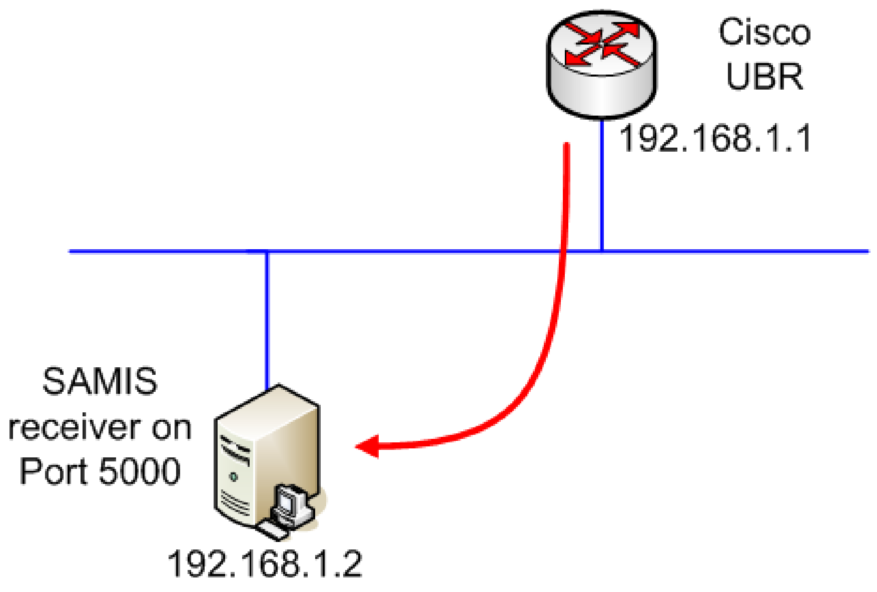
The Cisco SAMIS feature allows the Cisco CMTS to send data to a remote server either in over a standard TCP connection or using SSL.
The configuration for SAMIS is very simple however to allow SSL connections an IOS version with crypto support is required.

Configuration Elements

To configure a Cisco UBR to send SAMIS data to a destination you can use a simple configuration command as follows

```
cable metering destination <destination IP> <destination port> 1 <time> <secure | non-secure>
```

If we now apply some example IP addresses to our diagram we can see how this can be applied to a real network and how the data is collected.



Non Secure Configuration

The command required to configure the Cisco UBR would be

```
cable metering destination 192.168.1.2 5000 1 15 non-secure
```

This would configure the UBR to send data to the remote location which is listening on port 5000.

To collect the data into simple XML files, you could then use the code example in the appendix section of this document marked UBR-XML-Collect

Secure Configuration

The command required to configure the Cisco UBR would be

```
cable metering destination 192.168.1.2 5000 1 15 secure
```

You also need to generate a SSL certificate which is required to be configured on the UBR. To generate a self signed certificate use the following command line (assuming you have openssl installed)

```
openssl req -x509 -days 365 -newkey rsa:1024 -keyout hostkey.pem -nodes -out hostcert.pem
```

When executing the command you will see the following output. You will be prompted to enter information, which you can of course just leave blank.

```
Generating a 1024 bit RSA private key
..........................++++++
................++++++
writing new private key to 'hostkey1.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:Somewhere
Locality Name (eg, city) []:City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Test
Organizational Unit Name (eg, section) []:Test
Common Name (eg, YOUR name) []:Test
Email Address []:Test
```

You should now have two new files hostkey.pem and hostcert.pem. The hostcert.pem file contents are required for the UBR configuration.

To add the hostcert.pem file to the UBR configuration you need to apply the configuration as follows.

```
Configure terminal
crypto ca trustpoint IPDR
enrollment terminal
crl optional
exit
crypto ca authenticate IPDR
```

```
Enter the base 64 encoded CA certificate.
End with a blank line or the word "quit" on a line by itself
```

You should cut and paste the hostcert.pem file into the prompt and then type quit (as described above). If the import has been successful you will be asked if you wish to accept this key, indicate yes as needed and the key will be added into the configuration.

To collect the data into simple XML files, you could then use the code example in the appendix section of this document marked UBR-XML-CollectSSL

3.1.2. Cisco Deployment – 7200/10K – IPDR – Non Secure – 12.4

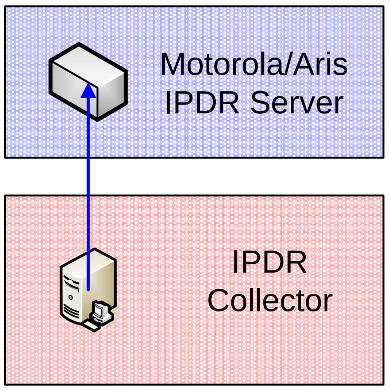
```
No Covered within this document
```

3.2. Motorola

The Motorola/Aris platforms support the IPDR protocol as specified in the IPDR 3.5 specification and use XDR encoding to transport data. These vendors have deployed server code on their CMTS' and so consequently a client is required to connect and collect data as, and when, it becomes available.

3.2.1. Motorola Deployment – BSR64000 – SAMIS

Conceptual Flow Diagram



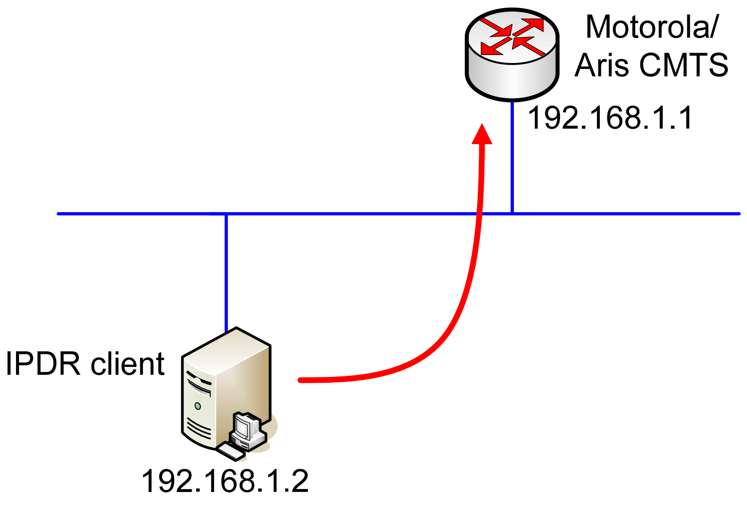
The Motorola BSR series provide the ability for a IPDR client to connect to the TCP port specified and receive IPDR data as and when it is available. The IPDR connection must be maintained, per the specification, for any data to be collected which will be sent in the XDR format. The IPDR specification does not provide a facility for SSL.

Configuration Elements

To configure a Motorola to allow an IPDR client to connect the following configuration is as follows

```
ipdr enable
ipdr collection-interval 15
ipdr collector 192.168.1.2 5000 3
```

If we now apply some example IP addresses to our diagram we can see how this can be applied to a real network and how the data is collected.



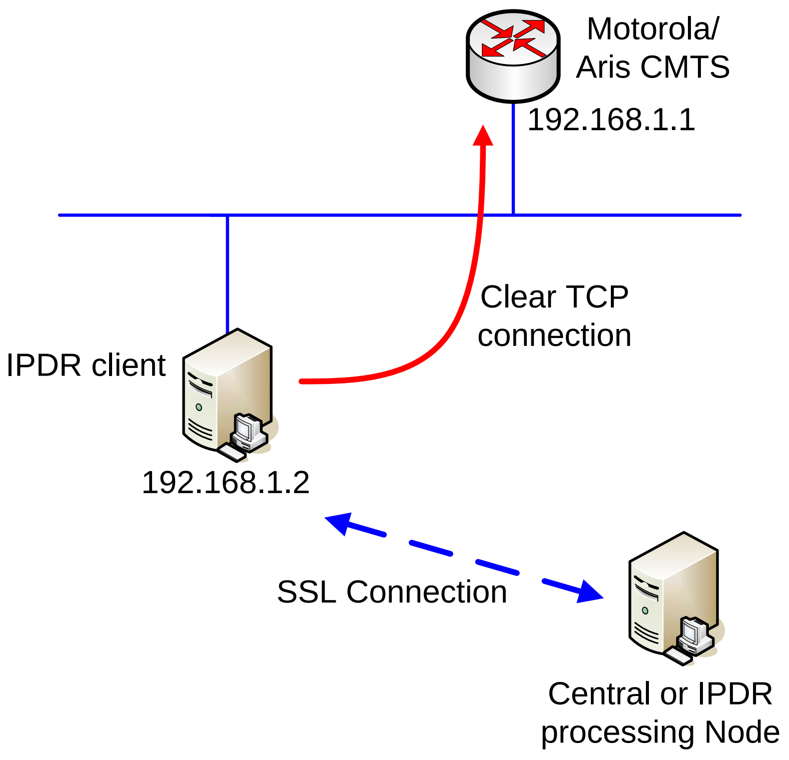
The configuration allows a client of the IP 192.168.1.2 to connect to the Motorola BSR on port 5000 and has a priority of 3. A separate client could be configured with a lower priority, say 1, however while the priority 3 client was connected the lower priority client would receive no data.

3.2.2. Motorola - Non Secure

To collect the data, you could then use the code example in the appendix section of this document marked Motorola-IPDR-Collect. It connects to the Motorola and when data is made available it is store in the XMLDirectory location. The connection to the Motorola is a NON SSL TCP connection.

3.2.3. Motorola - Non Secure with Secure Relay

In order to provide further security to a standard IPDR deployment a local collection mechanism could be utilised which also has the ability to relay the data over SSL to a central or processing server.



In order to deploy this scenario please see Motorola-IPDR-SSLRelay-Collect section. It is also possible to provide multiple destinations however the example does not cover this. The SSL keys can be generated in the same way as describe previously and need to be put on both the IPDR Client and Central/IPDR processing Node.

The IPDR processing Node can run the UBR-XML-CollectSSL example as the IPDR client will connect the Remote IP/Port specified.

One thing to note the XML generated by the IPDR client connecting the Motorola is marked as IPDR version 99.99.

4. Observations

The Cisco 7200 series routers only provide 32Bit counters. When doing any wrapping calculations for usage this should be taken into account.

If doing TCP relaying only using the Cisco module the Force32BitMode attribute can be used so allowing a 32bit system to be used. If *any* processing is to be done then this is not advised.

Only DOCSIS 1.1 and above provide Service Flow speeds, via Class attributes, if using this for data collection for DOCSIS 1.0 then the following OIDs will need to be gathered via SNMP

System Level QoS profile provisioned

'docsIfQosProfPriority'	**=>'1.3.6.1.2.1.10.127.1.1.3.1.2',**
'docsIfQosProfMaxUpBandwidth'	**=>'1.3.6.1.2.1.10.127.1.1.3.1.3',**
'docsIfQosProfMaxDownBandwidth'	**=>'1.3.6.1.2.1.10.127.1.1.3.1.5',**

Per SID QoS profile

'cdxCmtsCmCurrQoSPro'	*=>'1.3.6.1.4.1.9.9.116.1.3.6.1.3',*

Additional OIDs may be required to identify users further.

The IPDR module set version 0.22 should be used in any deployment and is currently under peer review and should be available from the 3rd of January 2010.

5. Appendix – Code Examples

5.1. UBR-XML-Collect

```
#!/usr/local/bin/perl

use strict;
use IPDR::Collection::Cisco;


my $ipdr_client = new IPDR::Collection::Cisco (
                        [
                        VendorID => 'IPDR Client',
                        ServerIP => '192.168.1.2',
                        ServerPort => '5000',
                        Timeout => 10,
                        XMLDirectory => '/collect/XMLDir',
                        DataHandler => \&display_data,
                        ]
                        );


# Check for data from the IPDR server.
my $status = $ipdr_client->connect();

if ( !$status )
        {
        print "Status was '".$ipdr_client->return_status()."'\n";
        print "Error was '".$ipdr_client->return_error()."'\n";
        exit(0);
        }

$ipdr_client->check_data_available();

exit(0);

sub display_data
{
my ( $remote_ip ) = shift;
my ( $remote_port ) = shift;
my ( $data ) = shift;
my ( $self ) = shift;

# We are not processing the data, just saving the XML.

return 1;
}
```

5.2. UBR-XML-CollectSSL

```
#!/usr/local/bin/perl

use strict;
use IPDR::Collection::CiscoSSL;

my $ipdr_client = new IPDR::Collection::CiscoSSL (
                        [
                        SSLKeyFile => 'hostkey.pem',
                        SSLCertFile => 'hostcert.pem',
                        VendorID => 'IPDR Client',
                        ServerIP => '192.168.1.2',
                        ServerPort => '5000',
                        Timeout => 2,
                        XMLDirectory => '/collect/XMLDir',
                        DataHandler => \&display_data,
                        ]
                        );


# Check for data from the IPDR server.
my $status = $ipdr_client->connect();

if ( !$status )
        {
        print "Status was '".$ipdr_client->return_status()."'\n";
        print "Error was '".$ipdr_client->return_error()."'\n";
        exit(0);
        }

$ipdr_client->check_data_available();

exit(0);

sub display_data
{
my ( $remote_ip ) = shift;
my ( $remote_port ) = shift;
my ( $data ) = shift;
my ( $self ) = shift;

# We are not processing the data, just saving the XML.

return 1;
}
```

5.3. Motorola-IPDR-Collect

```
#!/usr/local/bin/perl

use strict;
use IPDR::Collection::Client;

my $ipdr_client = new IPDR::Collection::Client (
                        [
                        VendorID => 'IPDR Client',
                        ServerIP => '192.168.1.1',
                        ServerPort => '5000',
                        KeepAlive => 60,
                        Capabilities => 0x01,
                        XMLDirectory => '/collect/XMLDir',
                        DataHandler => \&display_data,
                        Timeout => 2,
                        ]
                        );

# We send a connect message to the IPDR server
$ipdr_client->connect();

# If we do not connect stop.
if ( !$ipdr_client->connected )
        {
        print "Can not connect to destination.\n";
        exit(0);
        }

# We now send a connect message
$ipdr_client->check_data_available();

print "Error was '".$ipdr_client->get_error()."'\n";

exit(0);

sub display_data
{
my ( $remote_ip ) = shift;
my ( $remote_port ) = shift;
my ( $data ) = shift;
my ( $self ) = shift;

# We are not processing the data, just saving the XML.

}
```

5.4. Motorola-IPDR-SSLRelay-Collect

```
#!/usr/local/bin/perl

use strict;
use IPDR::Collection::Client;

my $ipdr_client = new IPDR::Collection::Client (
                        [
                        SSLKeyFile => 'pubhostkey.pem',
                        SSLCertFile => 'pubhostcert.pem',
                        VendorID => 'IPDR Client',
                        ServerIP => '192.168.1.2',
                        ServerPort => '5000',
                        RemoteIP => '10.1.1.1',
                        RemotePort => 5000,
                        RemoteSecure => 1,
                        KeepAlive => 60,
                        Capabilities => 0x01,
                        DataHandler => \&display_data,
                        DEBUG => 5,
                        Timeout => 2,
                        ]
                        );

# We send a connect message to the IPDR server
$ipdr_client->connect();

# If we do not connect stop.
if ( !$ipdr_client->connected )
        {
        print "Can not connect to destination.\n";
        exit(0);
        }

# We now send a connect message
$ipdr_client->check_data_available();

print "Error was '".$ipdr_client->get_error()."'\n";

exit(0);

sub display_data
{
my ( $remote_ip ) = shift;
my ( $remote_port ) = shift;
my ( $data ) = shift;
my ( $self ) = shift;

# We are not processing the data, just relaying the XML.

}
```

6. Data Collection and Processing Examples

The examples in this section show how to configure the IPDR module to process the IPDR which could then be used to update/create device usage information. The IPDR module suite should provide a consistent interface for both a Cisco and Motorola deployment.

6.1. Cisco-Processing

```
#!/usr/local/bin/perl

use strict;
use IPDR::Collection::Cisco;

my $ipdr_client = new IPDR::Collection::Cisco (
                        [
                        VendorID => 'IPDR Client',
                        ServerIP => ‘192.168.1.2',
                        ServerPort => '5000',
                        Timeout => 2,
                        Type => 'docsis',
                        DataHandler => \&display_data,
                        ]
                        );

# Check for data from the IPDR server.
my $status = $ipdr_client->connect();

if ( !$status )
        {
        print "Status was '".$ipdr_client->return_status()."'\n";
        print "Error was '".$ipdr_client->return_error()."'\n";
        exit(0);
        }

$ipdr_client->check_data_available();

exit(0);

sub display_data
{
my ( $remote_ip ) = shift;
my ( $remote_port ) = shift;
my ( $data ) = shift;
my ( $self ) = shift;

foreach my $host ( sort { $a<=> $b } keys %{$data} )
        {
        print "Host  is '$host' \n";
        foreach my $document_attribute ( keys %{${$data}{$host}{'document'}} )
                {
                print "Document id '$document_attribute' value is
'${$data}{$host}{'document'}{$document_attribute}'\n";
                }

        foreach my $sequence ( keys %{${$data}{$host}} )
                {
                next if $sequence=~/^document$/i;
                foreach my $attribute ( keys %{${$data}{$host}{$sequence}} )
                        {
                        print "Sequence is '$sequence' Attribute is '$attribute' value is
'${$data}{$host}{$sequence}{$attribute}'\n";
                        }
                }
        }
return 1;
}
```

This example processes the XML provided and breaks it down into flows and each attribute within the flow. Each flow is returned as a key within a hash and each attribute a key within the flow hash. The XML document descriptors are also returned so additional checking can be done that the correct number of flows has been processed.

As each flow is now provided is would be quite simple to query/update any data store to provide usage figures against MAC address and service flow name.

6.2. Motorola-Processing

```
#!/usr/local/bin/perl

use strict;
use IPDR::Collection::Client;

my $ipdr_client = new IPDR::Collection::Client (
                        [
                        VendorID => 'IPDR Client',
                        ServerIP => '192.168.1.1',
                        ServerPort => '5000',
                        KeepAlive => 60,
                        Capabilities => 0x01,
                        DataHandler => \&display_data,
                       Timeout => 2,
                        ]
                        );

# We send a connect message to the IPDR server
$ipdr_client->connect();

# If we do not connect stop.
if ( !$ipdr_client->connected )
        {
        print "Can not connect to destination.\n";
        exit(0);
        }

# We now send a connect message
$ipdr_client->check_data_available();

print "Error was '".$ipdr_client->get_error()."'\n";

exit(0);

sub display_data
{
my ( $remote_ip ) = shift;
my ( $remote_port ) = shift;
my ( $data ) = shift;
my ( $self ) = shift;

foreach my $sequence ( sort { $a<=>$b } keys %{$data} )
        {
        print "Sequence  is '$sequence'\n";
        foreach my $attribute ( keys %{${$data}{$sequence}} )
                {
                print "Sequence '$sequence' attribute '$attribute' value
'${$data}{$sequence}{$attribute}'\n";
                }
        }

}
```

This example processes the IPDR data converts the XDR data into flows and each attribute within the flow. Each flow is returned as a key within a hash and each attribute a key within the flow hash. There are not document descriptors provided for each data set so no additional checking can be done that the correct number of flows have been processed.

As each flow is now provided is would be quite simple to query/update any data store to provide usage figures against MAC address and service flow name.