INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

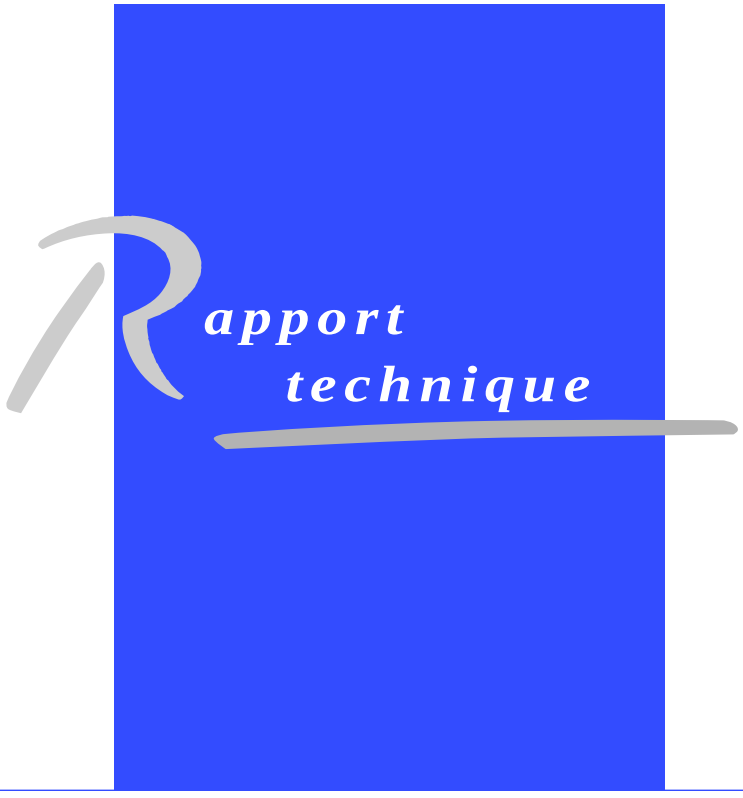
Programming the Flowoid NetFlow v9 Exporter on Android

Julien Vaubourg

N° 7003

April 2013

THÈME ?



ISSN 0249-0803

Programming the Flowoid NetFlow v9 Exporter on Android

Julien Vaubourg*

Thème ? — (hors thème INRIA)
Projet MADYNES

Rapport technique n° 7003 — April 2013 — 12 pages

Abstract: The RFC 3954 documents the NetFlow services export protocol Version 9. This project - associated to the Flowoid Projet - allows an Android program to listen the network traffic and send it to a remote NetFlow Collector using this standard.

Thanks to an highly scalable design, any sort of template can be implemented and any kind of network can be exported as NetFlow. The data generator side uses the flexibility of Java (with the Android API) whereas the sniffer side uses the power of C (native code).

This document explains how to configure the Exporter in order to export consistent information to the remote collector.

Key-words: netflow, flowoid, network traffic, monitoring, security, android

* TELECOM Nancy (last year internship in MADYNES)

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Programmation de l'exporteur NefFlow v9 du projet Flowoid sur Android

Résumé : Ce document explique comment programmer l'exporteur de NetFlow v9 sur Android : gestion des pilotes, définition des templates, génération des données et filtrage des paquets.

Mots-clés : netflow, flowoid, trafic réseau, supervision, sécurité, android

1 Design

NetFlows are traveling between two sides:

- **Exporter:** Catches all network packets through Drivers (directly on the Android device), resumes network flows with the NetFlow format, generates some additional data to add into NetFlows and exports them to the remote Collector.
- **Driver:** Listens the traffic network on the Exporter side and transmits packets headers in live to the Exporter.
- **Collector:** On the other side (remote server), receives and treats NetFlows sent by the Exporter.

In order to transmit NetFlows, an Exporter send one or more Export Packets to a remote Collector:

- **Export Packet:** Contains one or more Data FlowSets, and sometimes a Template FlowSet.
- **Data FlowSet:** Contains one or more Data Records. All of these are defined with the same Template Record and the Data FlowSet ID corresponds to the ID of this Template Record.
- **Data Record:** Contains one or more Data Fields. This kind of data corresponds to what is generally called a NetFlow.
- **Data Field:** Contains only one value (e.g. IP address, TCP port, timestamp).

The Collector knows how parsing Export Packets because this one sends regularly a Template Flowset among the Data FlowSets:

- **Template FlowSet:** Contains one or more Template Records. Its FlowSet ID is inevitably zero.
- **Template Record:** Contains one or more Template Fields ordered.
- **Template Field:** Is defined with a Field Type Definition (i.e. an ID and a length in bytes).
- **Field Type Definition:** Described in RFC 3954 section 8. A Data Field associated with a Template Field should be interpreted by the remote Collector in the same manner as described in RFC. Some types have a fixed length.

The Collector side needs to know all of the Field Type Definitions used by the Exporter especially if this one is out of standards.

2 Environment and prerequisites

Setting up your project:

1. The minimum SDK Version of your Android application must be at least 9.
2. **Your device must be rooted.**
3. Add *madynes-flowoid-exporter.jar*[1] to your Java Build Path. If you want to use the IP Driver, you could also add *netutils-parse-v1.jar*[2] for parsing headers.
4. Copy your Driver binary files[3] in your *res/raw/* directory.
5. Your *AndroidManifest* file must contain at least these permissions:

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

6. Create a new class extending `ExporterService` (e.g. *Exporter.java*):

```
import madynes.flowoid.exporter.*;
import madynes.flowoid.exporter.TemplateRecord.*;
import madynes.flowoid.exporter.TemplateField.InadequateLengthException;
import madynes.flowoid.exporter.TemplateFlowSet.TemplateNotFoundException;

public class Exporter extends ExporterService {

        @Override
        public void registerTemplates() throws AlreadyUsedIDException,
            InadequateIDException, InadequateLengthException {

        }

        @Override
        public void processPacketFiltering(byte[] headers) throws
            TemplateNotFoundException {

        }

        @Override
        public void onStart() {

        }

        @Override
        public void onStop() {
```

[1] TODO: Add web address
[2] TODO: Add web address
[3] TODO: Add web address

```
25
26          }
27 }
```

7. In your *AndroidManifest* think to declare your Exporter Service (here *your/package/-Exporter.java*):

```
1 <service android:name="your.package.Exporter" ></service>
```

The next sections explain you how to implement your own Exporter.

3 Creating templates

All templates are defined inside of the `registerTemplates()` function.

1. Create all of the Template Fields you need, by creating `TemplateField` objects. The constructor needs an ID among those available in `TemplateField.TypeDefinition.*`. If the associated length is zero, you need to use the alternative constructor to specify a length in bytes.

   Example:

   ```
   TemplateField tcpPortSrc = new TemplateField(TemplateField.TypeDefinition.L4_SRC_PORT).
   ```

2. Create a new Template Record with your own ID. This one is unique and must be between 256 and 65535.

   Example:

   ```
   TemplateRecord ipv6TcpTemplate = new TemplateRecord(257);
   ```

3. Add Template Fields to your Template Record with the `addTemplateField(TemplateField)` function. A same Template Field instance can be reused for many different Template Records.

   Example:

   ```
   ipv6TcpTemplate.addTemplateField(tcpPortSrc);
   ```

4. Register your new Template Record with the `registerTemplateRecord(RecordTemplate)`.

   Example:

   ```
   registerTemplateRecord(ipv6TcpTemplate);
   ```

4 Defining how to fill your NetFlows

Template Fields define how the data is represented and - for the standardized ones - what is it. Data Fields define how the value can be retrieved from the System. So each Template Field should have a corresponding Data Field.

To create a Data Field, you have to add a new class extending `DataField`. Then you have to implement four functions:

1. `onInit(byte[])`: Called when a new NetFlow is created. The argument corresponds to the headers (depending of the Driver used) of the first packet received for this NetFlow.
2. `onUpdate(byte[])`: Called when a packet is catching for an existent NetFlow. The argument corresponds to the new packet headers.
3. `onExport(byte[])`: Called just before the Export Packet is sent with this NetFlow (Data Record) inside. The argument corresponds to the last packet received. But if the Data Record was flagged ended (`isEnded()` condition met) it's the end packet. E.g. in the case of a TCP connection, if the record is ended it's the packet containing a FIN or RST flag. And not the least packet (final ACK). However in the case of an UDP connection, if the record is ended it's the least packet before the final time out.
4. `getValue()`: Called when the Exporter creates the Export Packet. Please use the `getByteBuffer()` function to create a `ByteBuffer`. For instance, if this Data Field is associated to a Template Field with a length of 4 (bytes), you have to use the `putInt(int)` function on your `ByteBuffer` before returning it.

Example of implementation (*FieldTCPPortSrc.java*):

```
1  import java.nio.ByteBuffer;
2  import edu.huji.cs.netutils.parse.TCPPacket;
3
4  public class FieldTCPPortSrc extends DataField {
5      private short value;
6
7      @Override
8      public void onInit(byte[] headers) {
9          TCPPacket tcpPacket = new TCPPacket(headers);
10         value = (short) tcpPacket.getSourcePort();
11     }
12
13     @Override
14     public void onUpdate(byte[] headers) {}
15
16     @Override
17     public void onExport(byte[] headers) {}
18
19     @Override
```

```
20      public ByteBuffer getValue() {
21          ByteBuffer bytes = getByteBuffer();
22
23          bytes.putShort(value);
24
25          return bytes;
26      }
27  }
```

5 Condition to cut NetFlows

A new NetFlow is created when the combination of these discriminant fields is not yet known. This point will be explained later in the section 6 page 7.

Conditions to stop NetFlows are evaluated for every new packet. If conditions are met, the NetFlow is moved from the current NetFlows list to the Export Queue.

So, for each Template Record you register, you have to create a Data Record extending the `DataRecord` class and override the `isEnded(byte[])` function. The argument corresponds to the new packet headers added to the NetFlow.

For instance (*RecordTCP.java*):

```
1  import madynes.flowoid.exporter.TemplateRecord;
2  import edu.huji.cs.netutils.parse.TCPPacket;
3
4  public class RecordTCP extends DataRecord {
5
6          public RecordTCP(TemplateRecord template) {
7                  super(template);
8          }
9
10         @Override
11         public boolean isEnded(byte[] headers) {
12                 TCPPacket tcpPacket = new TCPPacket(headers);
13
14                 return tcpPacket.isFin() || tcpPacket.isRst();
15         }
16
17 }
```

6 Associating data with templates

Your programming space for that is the `processPacketFiltering(byte[])` function.

1. First choose a previously registred Template Record with a Template Record ID.

Example:

```
TemplateRecord template = getRegistredTemplateRecord(257);
```

2. Then create a Data Record with one of your previoulsy created class.

   Examples:

   ```
   record = new RecordTCP(template);
   ```

3. Associate Template Fields of your Data Record with Data Fields one by one. You must associate all of them and in the same order you added Template Fields in your Template Record. The boolean argument specify whether this Data Field is a part of the NetFlow Key (see below and the example section 7 page 8) or not.

   Example:

   ```
   record.associateTemplateWithData(new FieldIPv6AddrSrc(), true);
   ```

4. Finally, request the packet processing with your record.

   Example:

   ```
   processPacket(record, headers);
   ```

Your record - in fact only fields declared as NetFlow Key part - will be used to identify the NetFlow/record associated to the current packet. If the associated NetFlow/record already exists, the packet will be added to it. Otherwise your new record will be used as new NetFlow.

7 Filtering packets

Every sniffed packet is sent to the `processPacketFiltering(byte[])` packet. So before creating records, you have to parse and check headers.

For instance:

```
1  @Override
2  public boolean processPacketFiltering(byte[] headers) throws
       TemplateNotFoundException {
3          LocationFinder locationFinder = LocationFinder.getInstance(this);
4          DataRecord record = null;
5
6          try {
7                  // TCP / IPv6
8                  if(EthernetFrame.statIsIpv6Packet(headers)
9                          && IPFactory.isTCPPacket(headers)) {
10
```

```
11                          TemplateRecord template = getRegistredTemplateRecord("
                                ipv6-tcp");
12                          record = new RecordTCP(template);
13
14                          // NetFlow Key Parts
15                          record.associateTemplateWithData(new FieldIPv6AddrSrc
                                (), true);
16                          record.associateTemplateWithData(new FieldIPv6AddrDst
                                (), true);
17                          record.associateTemplateWithData(new FieldTCPPortSrc()
                                , true);
18                          record.associateTemplateWithData(new FieldTCPPortDst()
                                , true);
19
20                          // Others
21                          record.associateTemplateWithData(new FieldIPv6Length()
                                , false);
22                  }
23
24                  // Other kind of packet for other record type
25                  else if(...) {
26                          ...
27                  }
28
29                  // Go! (or not)
30                  if(record != null) {
31                          processPacket(record, headers);
32                  }
33
34          // Parser error
35          } catch(...) {
36                  return false;
37          }
38
39          return true;
40  }
```

8 New Driver

Drivers are written in native code (C) in order to access to low functionnalities as sniffing network traffic.

Your only constraint is sending every packet headers to the Java Exporter through a TCP socket (see section 9 page 10 to know the port) and creating templates on the Exporter side. You will need a Java parser for your headers.

Put your sources in the *jni/drivers/* directory and use the Makefile to build your Driver. This one will copy the binary in the *res/raw/* directory. Then just after the Exporter Service starting, call `Driver.useDriver` with your Driver name resource and the network interface

to listen on.

For instance: `Driver.useDriver(this, R.raw.driver_ip, "eth0");`.

Finaly just reinstall your Android application.

9 Preferences

All of the below variables are available from your *ExporterService* implementation through classical getters and setters and you have to set them before using the Exporter Service:

- `prefCheckToExportInterval:` The Export Thread checks every N seconds if it can create and send an Export Packet.
- `prefCollectorAddr:` Collector address (IP address or hostname).
- `prefCollectorPort:` UDP port to connect to for sending NetFlow to the remote Collector.
- `prefDriversPort:` TCP port to listen on for receiving packets headers from drivers.
- `prefCheckDriversInterval:` The Drivers Thread checks every N seconds if all Drivers are still working and restarts them if necessary.
- `prefMaxHeadersSize:` Max headers size in bytes authorized for a sniffed packet transmitted by a Driver.
- `prefMaxNetflowsToTrack:` Max NetFlows/records to track simultaneous.
- `prefMinRecordsByFlowset:` Don't send a FlowSet if there are less than N records into it.
- `prefMaxExportPacketSize:` In order to avoid fragmentation, cut Export Packets nearly N bytes.
- `prefNetflowTimeout:` After N seconds of life, a NetFlow/record is considered ended and is added to the Export Queue. This is useful for non-connected protocols (e.g. UDP) and long term connections (e.g. TCP streaming).
- `prefPacketTimeout:` Even though `prefMinRecordsByFlowset` is not reached, after N seconds, the Export Thread empties the Export Queue and sends an Export Packet.
- `prefSendTemplatesEveryNPackets:` Send the Template FlowSet every N Export Packet.
- `prefTimeBeforeNetFlowEnd:` Returns the life time remaining when a NetFlow is detected as ended. This is useful to catch the last ACK with a TCP three-way handshake avoiding to create a second NetFlow.

10 Statistics

All of the below getters are available from your *ExporterService* implementation:

- `getStatActiveNetflows()`: Number of currently active NetFlows/records.
- `getStatConnectedDrivers()`: Number of currently connected Drivers.
- `getStatExportedNetFlows()`: Number of NetFlows/records sent from the Exporter to the Collector.
- `getStatExportedPackets()`: Number of Export Packets sent from the Exporter to the Collector.
- `getStatFailedSniffedPackets()`: Number of sniffed packets with which a parser error has occurred at the filtering step.
- `getStatNetflowsWaitingToBeExported()`: Number of NetFlows/records currently waiting to be exported (Export Queue size).
- `getStatRecognizedSniffedPackets()`: Number of sniffed packets recognized by at least one parser at the filtering step.
- `getStatRejectedNetFlows()`: Number of NetFlows/record not created due to the `prefMaxNetflowsToTrack` preference.
- `getStatSniffedPackets()`: Number of sniffed packets received by the Exporter from all Drivers.
- `getStatUnrecognizedSniffedPackets()`: Number of sniffed packets unrecognized by any parser (`statSniffedPackets` - `statRecognizedSniffedPackets` - `statFailedSniffedPackets`).

11 Activity

The minimal code for calling your Exporter is (with an IP driver, a network interface *eth0* and your implementation *Exporter.java*):

```
1  import madynes.flowoid.exporter.ExporterService;
2
3  public class MainActivity extends Activity {
4
5          @Override
6          protected void onCreate(Bundle savedInstanceState) {
7                  super.onCreate(savedInstanceState);
8
9                  ExporterService.start(this, Exporter.class);
10                 Driver.use(this, R.raw.driver_ip, "eth0", "tcp or udp");
11         }
```

```
12 }
13
14 The last empty argument corresponds to optional libcap filters.
```

Contents

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803