Telemetry Technology White Paper

Contents

Telemetry

Technical background

Driven by the popularity of networking and the emergence of new technologies, networks have grown radically in size and deployment complexity. Users also have increasingly high demands for service quality. The demands require network operations and maintenance should be more refined and intelligent. Operations and maintenance personnel are facing the following challenges:

- **Mass scale networking**—An abundance of monitored devices, and massive amounts of monitored data.
- **Fast problem location**—Second-level or even subsecond-level problem location is required in complex networks.
- **Granular monitoring**—To have a complete and accurate picture of the network, operations and maintenance personnel must monitor more data types with a high degree of granularity. In this way, they can predict the faults that might occur and base network optimization on the collected data. In addition to interface traffic statistics, per-flow traffic loss, CPU usage, and memory usage, operations and maintenance personnel must monitor per-flow delay and jitter, delay of each packet on its transmission path, and buffer usage on each device.

Traditional monitoring methods (for example, SNMP, CLI, and logging) cannot meet the requirements of networks.

- SNMP and CLI use a pull model to request information from a device, which limits the number of monitored devices and the speed of obtaining information.
- Although SNMP traps and logging use a push model to automatically obtain information from a device, the obtained information is limited to events and alarms and cannot accurately reflect network status.

Telemetry is a remote data collection technology that monitors device performance and faults. It uses a push model to obtain rich sets of data and help rapidly locate faults.

Benefits

Telemetry has the following benefits:

- A variety of implementation methods.
- Finer granularity and more types of data to be collected.
- Compared with the traditional network monitoring technology featuring one query, one reporting, telemetry requires only one-time configuration for continuous data reporting, thereby reducing the request processing load of the device.
- Rapid fault location.

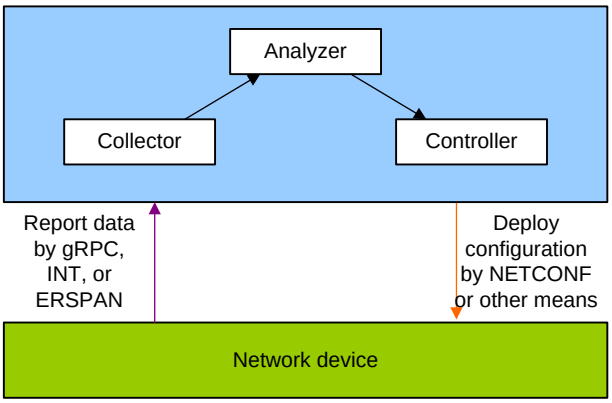
Telemetry network model

As shown in Figure 1, a telemetry network contains the following components:

- **Network device**—Monitored device. It samples monitored data and sends sampled data to the collector at regular intervals through Google Remote Procedure Call (gRPC), Inband Telemetry (INT), or Encapsulated Remote Switch Port Analyzer (ERSPAN).
- **Collector**—Receives and stores the data from a network device.
- **Analyzer**—Analyzes and processes data collected by the collector and presents them to users in a GUI interface.

- **Controller**—Manages a network device by deploying configurations to it through NETCONF or other means. It can deploy configurations or adjust the forwarding behaviors of the device based on the analyzed data. Additionally, it can control which data is sampled and reported by the device.

Figure 1 Telemetry network model



Telemetry implementation

Depending on the data reporting mode, telemetry is divided into the following types:

- gRPC-based telemetry

  gRPC is an open-source, high-performance RPC framework that uses HTTP 2.0 for transport. gRPC supports a variety of programming languages to configure and manage network devices.

  gRPC-based telemetry can collect interface traffic statistics, CPU usage, alarm information and more. It uses protocol buffers to encode collected information and then report it to the collector.

- INT-based telemetry

  Proposed by Barefoot, Arista, Dell, Intel, and VMware, INT is a network monitoring technology designed to collect data from the device. The device proactively sends data to a collector in real time for device performance monitoring and network monitoring.

  INT collects per-packet data plane information such as path and delay information and can implement comprehensive, real-time monitoring.

- ERSPAN-based telemetry

  ERSPAN encapsulates mirrored packets into GRE packets with protocol number 0x88BE and sends them to a remote monitoring device.

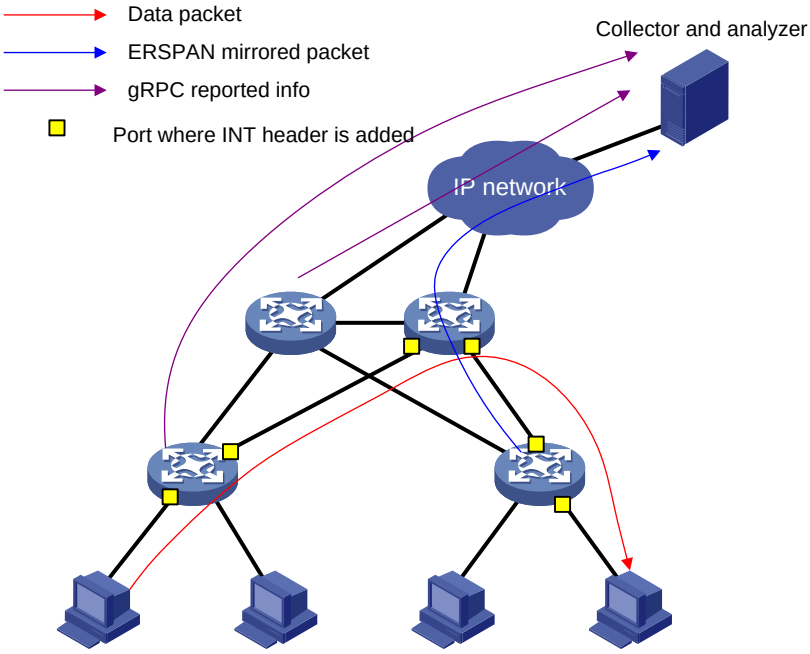
  You can define the packets to be mirrored according to your actual requirements. For example, you can mirror TCP three-way handshake messages to monitor TCP connection setup or mirror Remote Direct Memory Access (RDMA) signaling messages to monitor the RDMA session state.

Application scenarios

As shown in Figure 2, you can deploy multiple telemetry technologies in a network to achieve a comprehensive, multifaceted monitoring strategy. Alternatively, you can deploy a single telemetry technology targeted at a specific monitoring aim.

After a telemetry technology sends collected data to the collector, the administrator can present it through a GUI interface on the collector for rapid fault location. The administrator can also identify potential issues in time and take actions to prevent faults from occurring.

Figure 2 Application scenario



gRPC-based telemetry

gRPC-based telemetry enables the device to read various types of data (for example, CPU, memory, and interface statistics) and push data to collectors as subscribed. Compared with traditional monitoring methods, this design features real-time monitoring and high performance.

About gRPC

gRPC is an open source remote procedure call (RPC) system initially developed at Google. It uses HTTP 2.0 for transport and provides network device configuration and management methods that support multiple programming languages.

gRPC protocol stack layers

As shown in Figure 3, the gRPC protocol stack contains the following layers:

- **TCP transport layer**—Provides connection-oriented reliable data links.
- **TLS transport layer**—(Optional.) Uses TLS to perform channel encryption and bidirectional certificate authentication to implement secure communication.
- **HTTP 2.0 layer**—Carries gRPC. This layer takes advantage of the performance enhancement features provided by HTTP 2.0, for example, HTTP header compression, full request and response multiplexing, and flow control.
- **gRPC layer**—Defines the protocol interaction format for remote procedure calls. The definitions of the public RPC methods are provided in public .proto files, for example, grpc_dialout.proto.
- **Content layer**—Carries coded service data. The following are the supported service data code formats:
  - **Google Protocol Buffers**—A high-performance binary data format that uses proto definition files to describe the data structures used by the code. During data transmission

between the device and a collector, this code format supports carrying more information than other formats, for example, JSON.

Proto definition files for service modules are required to decode service data in Google Protocol Buffers format.

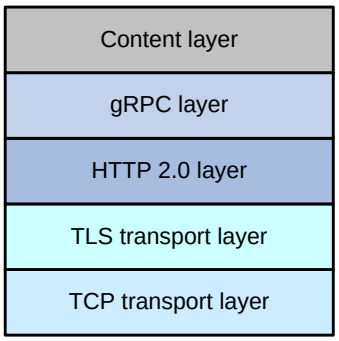
The device does not support service data in Google Protocol Buffers format in the current software version.

- **JavaScript Object Notation (JSON)**—A lightweight data-interchange text format that is language independent. JSON is easy for humans to read and write and easy for devices to parse and generate.

  Service data in JSON format can be decoded by using public proto definition files. No proto definition files for service modules are required.

To decode data correctly, a device and a collector must use the same proto definition files.

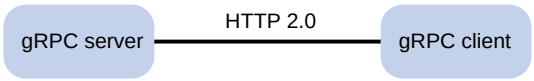
Figure 3 gRPC protocol stack layers



gRPC network architecture

As shown in Figure 4, the gRPC network uses the client/server model. It uses HTTP 2.0 for packet transport.

Figure 4 gRPC network architecture



The gRPC network uses the following mechanism:

1. The gRPC server listens to connection requests from clients at the gRPC service port.
2. A user runs the gRPC client application to log in to the gRPC server, and uses methods provided in the .proto file to send requests.
3. The gRPC server responds to requests from the gRPC client.

The device can act as the gRPC server or client.

Dial-in mode and dial-out mode

The device supports the following modes:

- **Dial-in mode**—The device acts as a gRPC server and the collectors act as gRPC clients. A collector initiates a gRPC connection to the device to subscribe to device data.

  Dial-in mode applies to small networks where collectors need to deploy configurations to devices.

- **Dial-out mode**—The device acts as a gRPC client and the collectors act as gRPC servers. The device initiates gRPC connections to the collectors and pushes device data to the collectors as configured.

  Dial-out mode applies to larger networks where a large number of devices need to be monitored.

Protocol buffer code format

Google Protocol Buffers provide a flexible mechanism for serializing structured data. Different from XML code and JSON code, the protocol buffer code is binary and provides higher performance.

Table 1 compares a protocol buffer code format example and the corresponding JSON code format example. The JSON code format is used for all service data notifications.

Table 1 Protocol buffer and JSON code format examples

Protocol buffer code format example	Corresponding JSON code format example

{
1:“H3C”
2:“H3C”
3:“H3C Simware”
4:“Syslog/LogBuffer”
5:"notification": {
"Syslog": {
"LogBuffer": {
"BufferSize": 512,
"BufferSizeLimit": 1024,
"DroppedLogsCount": 0,
"LogsCount": 100,
"LogsCountPerSeverity": {
"Alert": 0,
"Critical": 1,
"Debug": 0,
"Emergency": 0,
"Error": 3,
"Informational": 80,
"Notice": 15,
"Warning": 1
},
"OverwrittenLogsCount": 0,
"State": "enable"
}
},
"Timestamp": "1527206160022"
}
}
	{
	"producerName": "H3C",
	"deviceName": "H3C",
	"deviceModel": "H3C Simware",
	"sensorPath": "Syslog/LogBuffer",
	"jsonData": {
	"notification": {
	"Syslog": {
	"LogBuffer": {
	"BufferSize": 512,
	"BufferSizeLimit": 1024,
	"DroppedLogsCount": 0,
	"LogsCount": 100,
	"LogsCountPerSeverity": {
	"Alert": 0,
	"Critical": 1,
	"Debug": 0,
	"Emergency": 0,
	"Error": 3,
	"Informational": 80,
	"Notice": 15,
	"Warning": 1
	},
	"OverwrittenLogsCount": 0,
	"State": "enable"
	}
	},
	"Timestamp": "1527206160022"
	}
	}
	}

Proto definition files

You can define data structures in a proto definition file. Then, you can compile the file with utility protoc to generate code in a programing language such as Java and C++. Using the generated code, you can develop an application for a collector to communicate with the device.

H3C provides proto definition files for both dial-in mode and dial-out mode.

Proto definition files in dial-in mode

Public proto definition files

The **grpc_service.proto** file defines the public RPC methods for dial-in mode, for example, login method and logout method.

The following are the contents of the **grpc_service.proto** file:

```
syntax = "proto2";
package grpc_service;
message GetJsonReply { // Reply to the Get method
    required string result = 1;
}
message SubscribeReply { // Subscription result
    required string result = 1;
}
message ConfigReply { // Configuration result
    required string result = 1;
}
message ReportEvent { // Subscribed event
    required string token_id = 1; // Login token_id
    required string stream_name = 2; // Event stream name
    required string event_name = 3; // Event name
    required string json_text = 4; // Subscription result, a JSON string
}
message GetReportRequest{ // Obtains the event subscription result
    required string token_id = 1; // Returns the token_id upon a successful login
}
message LoginRequest {  // Login request parameters
    required string user_name = 1; // Username
    required string password = 2; // Password
}
message LoginReply {  // Reply to a login request
    required string token_id = 1; // Returns the token_id upon a successful login
}
message LogoutRequest { // Logout parameter
    required string token_id = 1; // token_id
}
message LogoutReply { // Reply to a logout request
    required string result = 1; // Logout result
}
message SubscribeRequest { // Event stream name
    required string stream_name = 1;
```

```
}
service GrpcService { // gRPC methods
    rpc Login (LoginRequest) returns (LoginReply) {}  // Login method
    rpc Logout (LogoutRequest) returns (LogoutReply) {} // Logout method
    rpc SubscribeByStreamName (SubscribeRequest) returns (SubscribeReply) {} // Event
subscription method
    rpc GetEventReport (GetReportRequest) returns (stream ReportEvent) {} // Method for
obtaining the subscribed event
}
```

Proto definition files for service modules

The dial-in mode supports proto definition files for the following service modules: Device, Ifmgr, IPFW, LLDP, and Syslog.

The following are the contents of the **Device.proto** file, which defines the RPC methods for the Device module:

```
syntax = "proto2";
import "grpc_service.proto";
package  device;
message DeviceBase { // Structure for obtaining basic device information
    optional string HostName = 1; // Device name
    optional string HostOid = 2; // sysoid
    optional uint32 MaxChassisNum = 3; // Maximum number of chassis
    optional uint32 MaxSlotNum = 4; // Maximum number of slots
    optional string HostDescription = 5; // Device description
}
message DevicePhysicalEntities { // Structure for obtaining physical entity information
of the device
    message Entity {
        optional uint32 PhysicalIndex = 1; // Entity index
        optional string VendorType = 2; // Vendor type
        optional uint32 EntityClass = 3;// Entity class
        optional string SoftwareRev = 4; // Software version
        optional string SerialNumber = 5; // Serial number
        optional string Model = 6; // Model
    }
    repeated Entity entity = 1;
}
service DeviceService { // RPC methods
    rpc GetJsonDeviceBase(DeviceBase) returns (grpc_service.GetJsonReply) {} // Method
for obtaining basic device information
    rpc GetJsonDevicePhysicalEntities(DevicePhysicalEntities) returns
(grpc_service.GetJsonReply) {}
} // Method for obtaining physical entity information of the device
```

Proto definition file in dial-out mode

The **grpc_dialout.proto** file defines the public RPC methods in dial-out mode. The following are the contents of the file:

```
syntax = "proto2";
package grpc_dialout;
```

```
message DeviceInfo{ // Pushed device information
    required string producerName = 1; // Vendor name
    required string deviceName = 2; // Device name
    required string deviceModel = 3; // Device model
}
message DialoutMsg{   // Format of the pushed data
    required DeviceInfo deviceMsg = 1; // Device information described by DeviceInfo
    required string sensorPath = 2; // Sensor path, which corresponds to xpath in NETCONF
    required string jsonData = 3; // Sampled data, a JSON string
}
message DialoutResponse{  // Response from the collector. Reserved. The value is not
processed.
    required string response = 1;
}
service gRPCDialout {  // Data push method
    rpc Dialout(stream DialoutMsg) returns (DialoutResponse);
}
```

Obtaining proto definition files

Contact H3C Support.

Supported service data types

In dial-in mode, the device can provide data of the following service modules: Device, Ifmgr, IPFW, LLDP, and Syslog. For more information, see the proto definition files provided by H3C.

In dial-out mode, the device can provide data sampled by using one of the following sampling methods:

- **Event-triggered sampling**—Samples data when certain events occur. For more information about the events, see *NETCONF XML API Event Reference* for the module.
- **Periodic sampling**—Samples data at intervals. For more information about supported data types, see the NETCONF XML API references for the module except for *NETCONF XML API Event Reference*.

To view the types of service data that can be sampled in dial-out mode, execute the **`sensor path`** command and read the online help.

INT-based telemetry

Overview

Technical background

As data center technologies advance rapidly and data center networks scale in size, a lack of automated operations and maintenance platform poses big challenges to operations and maintenance personnel.

In traditional networks, the radar detection technology is typically used to detect the packet forwarding path. Requiring the intervention of controller software, the radar detection technology

involves a series of complicated designs and implementations. Additionally, it cannot completely simulate real packet forwarding.

Ping and tracert utilities can monitor the network delay and path, but they cannot accurately determine on which interface a packet is longest delayed in a delay-sensitive network.

As an important part of the network visibility solution, INT is the first and the most important step in the journey towards automated operations and maintenance. With INT, we can know information about each device a packet traverses, egress/ingress port and queue information, timestamp information, and more.

Benefits

INT-based telemetry has the following benefits:

- Full hardware support.
- One-time configuration, continuous data reporting.
- Mirroring and sampling support.
- QoS policy support for flexibly matching packets to be monitored.
- Support for encapsulating and sending packets to the collector on the last node.
- Finer granularity of data collected: You can collect information about each device a packet traverses, egress/ingress port and queue information, timestamp information, and more.

Protocols and standards

INT is defined in the IETF's Internet Draft *Inband Flow Analyzer draft-kumar-ifa-00*. This draft describes the formats of the INT header and metadata in detail. Theoretically, network devices that support this draft can implement INT packet analysis and processing functions.

Network model

As shown in Figure 5, an INT-based telemetry network contains the following components:

- Entry node

  The entry node samples packets and mirrors sampled packets to the INT processor, adds an INT header and metadata to the packets, and sends them to the transit node.

- Transit node

  The transit node identifies INT packets, adds metadata to the INT packets, and sends them to the exit node.

- Exit node

  As the last hop in an INT network, the exit node identifies INT packets, adds metadata to the INT packets, encapsulates the UDP header and IP header for the INT packets, and sends them to the collector.

- Ingress port

  For the entry node, this port is the input interface of original packets. For the transit node and exit node, this port is the input interface of INT packets.

- Egress port

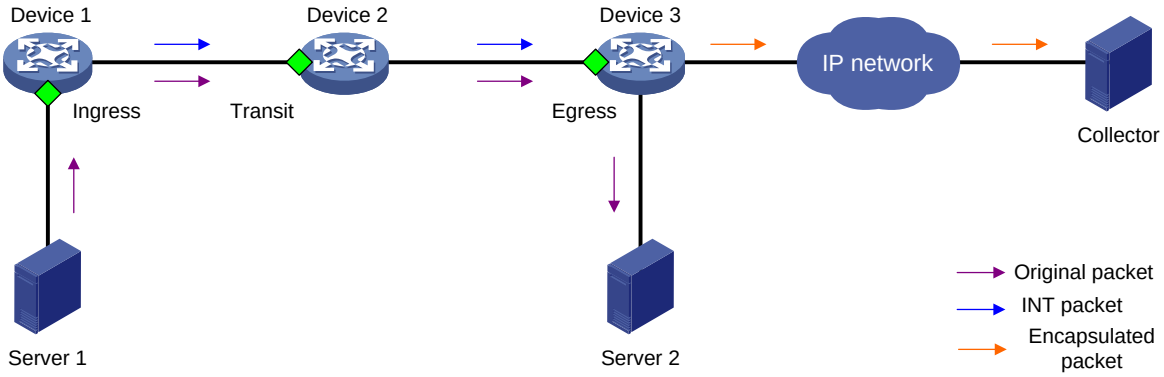
  For the entry node and transit node, this port is the output interface of INT packets. For the exit node, this port is the output interface of encapsulated packets.

- INT processor

  The INT processor is a dedicated processor in the CPU used for processing INT packets. For the mirrored packets on the entry node, the INT processor adds an INT header to generate INT

packets. On the exit node, the INT processor performs consistency checks on the encapsulation format of metadata and encapsulates the outer UDP header for the INT packets.

Figure 5 INT network diagram



Packet format

INT packet header

As shown in Figure 6 and Figure 7, an INT packet header contains two parts: **INT Probe HDR** (header inherent to INT) and **MD #1-N** (inserted metadata). INT has two packet types: INT over TCP and INT over UDP.

An original TCP packet is called an INT-over-TCP packet after it is mirrored and inserted with an INT header.

An original UDP packet is called an INT-over-UDP packet after it is mirrored and inserted with an INT header.

Figure 6 INT over TCP

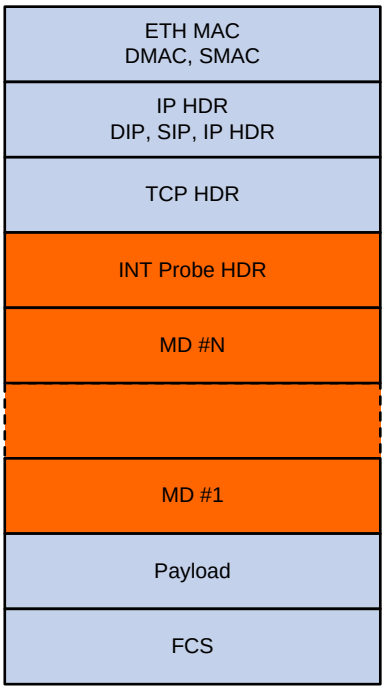
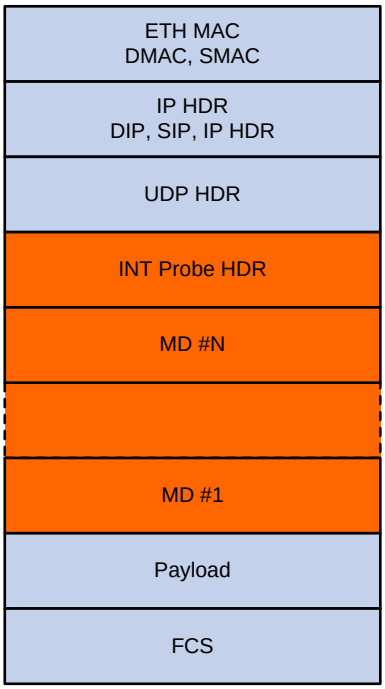


Figure 7 INT over UDP



INT-inherent header format

Figure 8 shows the format of the inherent header.

Figure 8 Inherent header format

Probe Maker (1) (32 bits)		
Probe Maker (2) (32 bits)		
Version (8 bits)	Message Type (8 bits)	Flags (16 bits)
Telemetry Request Vector (32 bits)		
Hop Limit (8 bits)	Hop Count (8 bits)	Must Be Zero (16bits)
Maximum Length (16 bits)		Current Length (16 bits)
Sender`s Handle (16 bites)		Sequence Number (16 bites)

The following explains the meanings of the fields:

- **Probe Marker**—Used by the device to identify INT packets. Its value is fixed at 0xaaaaaaaabbbbbbbb.
- **Version**—Currently fixed at 0x01.
- **Message Type**—Currently fixed at 0x01.
- **Flags**—A reserved field, currently fixed at 0x0000.
- **Telemetry Request Vector**—Currently fixed at 0xffffffff.
- **Hop Limit**—Maximum number of hops allowed.
- **Hop Count**—Number of nodes the packet has traversed.

- **Maximum Length**—Maximum length of metadata that can be collected, in bytes.
- **Current Length**—Length of metadata that has been collected, in bytes.
- **Sender's Handle**—Set by the entry node for the collector to uniquely identify an INT flow.
- **Sequence Number**—Sequence number of a packet in an INT flow.

Metadata format

Figure 9 shows the format of the metadata.

Figure 9 Metadata format

<table>
<tr><td colspan="5">Device-ID (32 bits)</td></tr>
<tr><td>Template-ID (3bits)</td><td>Congestion (5 bits)</td><td>Egress Port Drop Pkt Byte Cnt Upper (8 bits)</td><td>IP_TTL (8 bits)</td><td>Queue_Id (8 bits)</td></tr>
<tr><td colspan="5">Rx Timestamp Seconds Upper (32 bits)</td></tr>
<tr><td colspan="3">Rx Timestamp Seconds (16 bits)</td><td colspan="2">Rx Timestamp Nano-Seconds Upper (16 bits)</td></tr>
<tr><td colspan="3">Rx Timestamp Nano-Seconds Upper (16 bits)</td><td colspan="2">Tx Timestamp Nano-Seconds Upper (16 bits)</td></tr>
<tr><td colspan="3">Tx Timestamp Nano-Seconds Upper (16 bits)</td><td colspan="2">Egress Port Utilization [%] (16 bits)</td></tr>
<tr><td colspan="3">Ingress Port [module, port] (16 bits)</td><td colspan="2">Egress Port [module, port] (16 bits)</td></tr>
<tr><td colspan="5">Egress Port Drop Pkt Byte Cnt (32 bits)</td></tr>
</table>

The following explains the meanings of the fields:

- **Device-ID**—Device ID.
- **Template-Id**—A reserved field, currently fixed at 000.
- **Congestion**—Indicates the congestion state. The three most significant bits are fixed at 000, and the two least significant bits are the ECN field.
- **Egress Port Drop Pkt Byte Cnt Upper**—Drop count in bytes for the egress port, currently fixed at 0x00.
- **IP_TTL**—TTL value.
- **Queue-Id**—Egress queue ID, currently fixed at 0x00.
- **Rx Timestamp Seconds Upper/Rx Timestamp Seconds—**Ingress timestamp in seconds.
- **Rx Timestamp Nano-Seconds Upper**—Ingress timestamp in nanoseconds.
- **Tx Timestamp Nano-Seconds Upper**—Egress timestamp in nanoseconds.
- **Egress Port Utilization [%]**—Egress port utilization in percentage, currently fixed at 0x0000.
- **Ingress Port [module, port]**—Ingress port.
- **Egress Port [module, port]**—Egress port.
- **Egress Port Drop Pkt Byte Cnt**—Drop count in bytes for the egress port, currently fixed at 0x00000000.

Mechanism

INT-capable devices form an INT zone. Each INT device performs different functions in the INT zone.

- **Entry node**—By using a QoS policy on the ingress port, the entry node classifies incoming traffic and mirrors classified traffic to the INT processor. The INT processor adds the inherent

INT header to matching packets, adds metadata, and forwards the packets by looking up the routing table.

- **Transit node**—The transit node automatically identifies INT packets, adds metadata to them, and forwards them by looking up the routing table.
- **Exit node**—The transit node automatically identifies INT packets, sends the INT packets to the INT processor, and performs the following actions on the packets:
  a. Adds metadata.
  b. Encapsulates the outer UDP header for the INT packets.
  c. Encapsulates the outer IP header for the INT packets by using the configured encapsulation parameters.
  d. Routes the packet to the collector according to the destination IP address in the outer IP header.

Metadata that INT can collect

INT can collect and monitor the following metadata:

- **Device ID**.
- **Ingress port ID**—Logical input interface of packets.
- **Ingress timestamp**—The local time on the device when a packet enters the ingress port. For the entry node, it is the time when an INT packet enters the loopback interface.
- **Egress port ID**—Logical output interface of packets
- **Egress timestamp**—The local time on the device when a packet leaves the egress port.
- **Cache information**:
  - **Queue ID**—ID of the queue that caches original packets.
  - **ECN information**.

ERSPAN-based telemetry

Overview

As a Layer 3 remote monitoring technology, ERSPAN allows you to copy traffic on ports, VLANs, or CPUs and sends the copied traffic to a remote monitoring device through a GRE tunnel. You can analyze the traffic (mirrored packets) for monitoring and troubleshooting purposes.

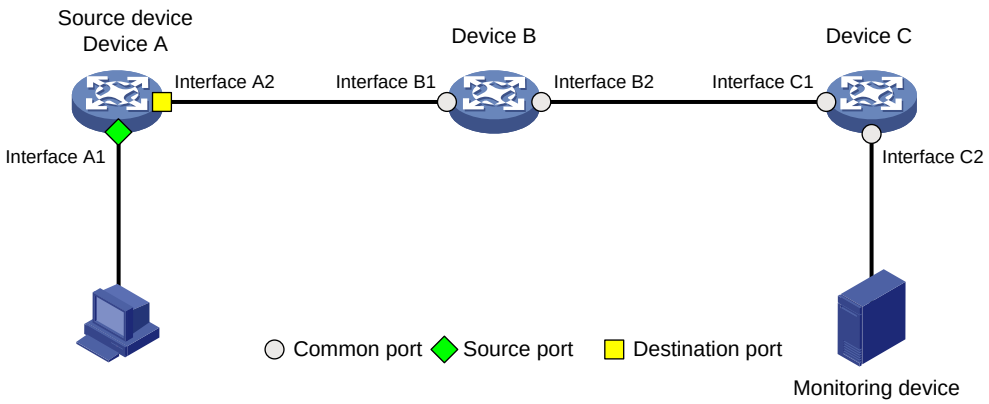
Network model

As shown in Figure 10, an ERSPAN-based telemetry network contains the following components:

- **Mirroring source**—Can be one or more monitored ports (called source ports), VLANs (called source VLANs), or CPUs (called source CPUs).
- **Source device**— The device where the mirroring sources reside is called a source device.
- **Mirroring destination**—The mirroring destination connects to a data monitoring device and is the destination port (also known as the monitor port) of mirrored packets. Mirrored packets are sent out of the monitor port to the data monitoring device.
- **Mirroring direction**—The mirroring direction specifies the direction of the traffic that is copied on a mirroring source.
  - **Inbound**—Copies packets received.

- **Outbound**—Copies packets sent.
- **Bidirectional**—Copies packets received and sent.

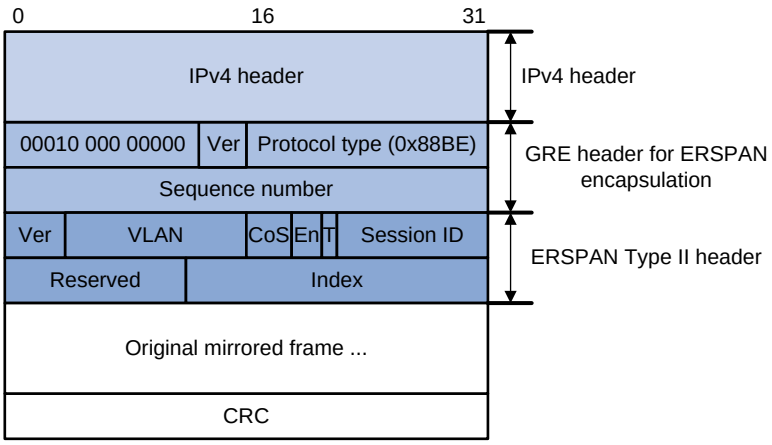
Figure 10 ERSPAN network model



Packet format

ERSPAN has three packet formats. The device supports only Type II format, which encapsulates mirrored packets into GRE packets with protocol number 0x88BE, as shown in Figure 11.

Figure 11 Packet format



ERSPAN adds an ERSPAN header, recalculates the CRC, and adds a GRE header and an IPv4 header for mirrored packets.

The meanings of fields in the GRE header and ERSPAN header are described as follows:

- GRE header
  - **Flags**— The S bit is 1, which indicates that a packet can be determined as an in-order or out-of-order packet through the sequence number. All other bits are 0.
  - **Ver**—Version number, which is fixed at 0.
  - **Protocol type (0x88BE)** —The passenger protocol of GRE is ERSPAN type II.
  - **Sequence number**—The sequence number increments by 1 when new packet is received.
- ERSPAN header
  - **Ver**—ERSPAN version number. The version number for ERSPAN type II is 1.
  - **CoS**—The original class of service in the mirrored packet.

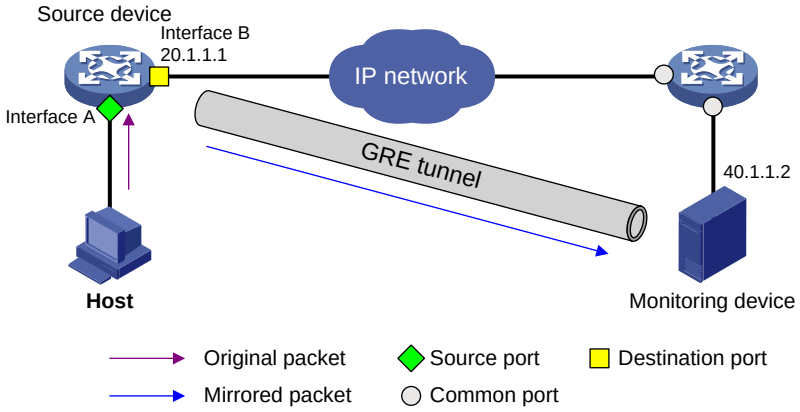
- **Sequence number**—The sequence number increments by 1 when new packet is received.
- **En**—Data encapsulation type of the source port. The value 00 indicates encapsulation without VLAN tags. The value 01 indicates ISL encapsulation. The value 10 indicates 802.1Q encapsulation. The value 11 indicates encapsulation with VLAN tags.
- **T**— The value 1 indicates that the mirrored packet was fragmented when being encapsulated in ERSPAN because it exceeded the interface MTU.
- **Session ID**—ERSPAN session ID. This ID must be unique for the same source device and destination device.
- **Index**—Index for the source port and mirroring direction.

Mechanism

As shown in Figure 12, ERSPAN mirrors packets in the following procedure:

1. The source device replicates a packet received on a source port, source VLAN, or source CPU.
2. The source device adds ERSPAN encapsulation for the replicated packet, with 20.1.1.1 as the source IP address and 40.1.1.2 (IP address of the monitoring device) as the destination IP address.
3. The source device sends the replicated packet to the monitoring device through a GRE tunnel.
4. The monitoring device decapsulates the packet and analyzes the contents of the packet.

Figure 12 ERSPAN mechanism



Application scenarios

ERSPAN allows you to mirror packets of interest to a remote monitoring device for analysis and monitoring purposes. For example, you can mirror TCP three-way handshake messages to monitor TCP connection setup or mirror RDMA signaling messages to monitor the RDMA session state.