                     User Devices Explicit Monitoring
             draft-cnbf-ippm-user-devices-explicit-monitoring-00

Abstract

   This document describes a methodology to monitor network performance
   exploiting user devices. This can be achieved using the Explicit
   Flow Measurement Techniques, protocol independent methods that employ
   few marking bits, inside the header of each packet, for loss and
   delay measurement. User devices and servers, marking the traffic,
   signal these metrics to intermediate network observers allowing them
   to measure connection performance, and to locate the network segment
   where impairments happen. In addition or in alternative to network
   observers, a probe can be installed on the user device with
   remarkable benefits in terms of hardware deployment and measurement
   scalability.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 30, 2021.

Copyright Notice

Table of Contents

1. Introduction

   Explicit Performance Monitoring enables a passive observer (a probe)
   to measure delay and loss just watching the marking (a few header
   bits) of live traffic packets. It works on client-server protocols:
   e.g. QUIC [QUIC-TRANSPORT], TCP [TCP]. The different methods are
   described in [EXPLICIT-FLOW-MEASUREMENTS].

   This document explains how to employ the methods described in
   [EXPLICIT-FLOW-MEASUREMENTS] by proposing the user device as a
   convenient place for the Explicit Performance Observer.

2. Notational Conventions

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in [RFC2119].

3. Explicit Performance Open Issues

   There are some open issues to consider for the deployment of
   [EXPLICIT-FLOW-MEASUREMENTS]:

   - Who decides whether to mark traffic? Explicit measures only work
      if both the server and the client mark the production traffic.

   - What about scalability? Could network probes monitor all the
      connections? If they cannot, which ones to choose?

   - Which connections to monitor within the network? Network probes
      need an effective way to identify which connections really need to
      be monitored.

   - How to monitor both traffic directions? Not always possible for
      network probes (asymmetric connections).

4. Explicit Performance Probes on User Devices

   This document proposes the user device (e.g. mobile phones, PCs) as a
   convenient place where to put the Explicit Performance Observer.

   The placement of the observer on the user device helps to mitigate
   the issues reported in the previous section, in particular:

   - The device should decide whether to mark the traffic or not.

   - Regarding the scalability issue, on the user device there are few
      connections to monitor so it becomes less relevant.

   - Connections eligible for monitoring should be the impaired ones.
      User devices and network probes can cooperate to achieve this
      goal. It is possible to set alarm thresholds on the user device
      and to signal to the network probes only the sessions with
      impairments. This allows to segment the performance measurements
      and to locate the faults. In this way network probes, that could
      also be embedded into network nodes, have to monitor a limited
      number of connections.

   - Monitoring both directions is always possible on the user device.

5. Device Owner Activates Explicit Performance Measurement

   The decision whether to activate the marking (e.g. [SPIN-BIT],
   [ANRW19-PM-QUIC], [EXPLICIT-FLOW-MEASUREMENTS]) or not should be made
   by the device owner by properly configuring the applications (e.g.

browsers) based on connection-oriented protocols that support
explicit measurements (e.g. QUIC).

All applications should provide the activation or deactivation of
packet marking, for example by providing a user interface or exposing
API.

So, during the client-server handshake, the client will decide
whether the marking is active or not within a session and notify its
decision to the server.

6. Who Will Handle the Performance Data?

Performance data are stored only on the user device or also sent to
"external bodies" according to the will of the device owner.

The main recipient would be the Internet Service Provider. Indeed,
as explained in the previous section, this enables user device and
network probes coordination that permits an improved performance
measurement approach.

Moreover these data could also be of interest for the national
regulatory authorities or others authorized subjects.

7. The Explicit Performance App

This methodology could be implemented with an "Explicit Performance
App" installed on the user device.

The App should perform the following tasks:

- collect user preferences;

- activate/deactivate marking on device Apps (e.g. browsers);

- implement the observer;

- show performances to the user;

- send data to the "Explicit Performance Management Center";

- set performance thresholds.

8. Security Considerations

TBD

9. Privacy Considerations

   TBD

10. IANA Considerations

   This document makes no request of IANA.

11. Change Log

   TBD

12. Contributors

   TBD

13. Acknowledgements

   TBD

14. References

14.1. Normative References

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119,
               DOI 10.17487/RFC2119, March 1997,
               <https://www.rfc-editor.org/info/rfc2119>.

   [TCP]   Postel, J., "Transmission Control Protocol", STD 7,
               RFC 793, DOI 10.17487/RFC0793, September 1981,
               <https://www.rfc-editor.org/info/rfc793>.

14.2. Informative References

   [ANRW19-PM-QUIC]
               Bulgarella, F., Cociglio, M., Fioccola, G., Marchetto, G.,
               and R. Sisto, "Performance measurements of QUIC
               communications", Proceedings of the Applied Networking
               Research Workshop, DOI 10.1145/3340301.3341127, July 2019.

   [EXPLICIT-FLOW-MEASUREMENTS]
               Cociglio, M., Fioccola, G., Nilo, M., Bulgarella, F., and
               R. Sisto, "Client-Server Explicit Performance
               Measurements", draft-cfb-ippm-spinbit-measurements-02
               (work in progress), July 2020.

   [QUIC-TRANSPORT]
              Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed
              and Secure Transport", draft-ietf-quic-transport-32 (work
              in progress), October 2020.

   [SPIN-BIT]
              Trammell, B., Vaere, P., Even, R., Fioccola, G., Fossati,
              T., Ihlar, M., Morton, A., and S. Emile, "Adding Explicit
              Passive Measurability of Two-Way Latency to the QUIC
              Transport Protocol", draft-trammell-quic-spin-03 (work in
              progress), May 2018.

Authors' Addresses

   Mauro Cociglio
   Telecom Italia (TIM)
   Via Reiss Romoli, 274
   Torino 10148
   Italy

   EMail: mauro.cociglio@telecomitalia.it

   Massimo Nilo
   Telecom Italia (TIM)
   Via Reiss Romoli, 274
   Torino 10148
   Italy

   EMail: massimo.nilo@telecomitalia.it

   Fabio Bulgarella
   Telecom Italia (TIM)
   Via Reiss Romoli, 274
   Torino 10148
   Italy

   EMail: fabio.bulgarella@guest.telecomitalia.it

   Giuseppe Fioccola
   Huawei Technologies
   Riesstrasse, 25
   Munich 80992
   Germany

   EMail: giuseppe.fioccola@huawei.com

                       Edge Data Discovery for COIN
                draft-mcbride-edge-data-discovery-overview-05

Abstract

   This document describes the problem of distributed data discovery in
   edge computing, and in particular for computing-in-the-network
   (COIN), which may require both the marshalling of data at the outset
   of a computation and the persistence of the resultant data after the
   computation. Although the data might originate at the network edge,
   as more and more distributed data is created, processed, and stored,
   it becomes increasingly dispersed throughout the network. There
   needs to be a standard way to find it. New and existing protocols
   will need to be developed to support distributed data discovery at
   the network edge and beyond.

Copyright Notice

Table of Contents

1. Introduction

   Edge computing is an architectural shift that migrates Cloud
   functionality (compute, storage, networking, control, data
   management, etc.) out of the back-end data center to be more
   proximate to the IoT data being generated and analyzed at the edges

of the network. Edge computing provides local compute, storage and
connectivity services, often required for latency- and bandwidth-
sensitive applications. Thus, Edge Computing plays a key role in
verticals such as Energy, Manufacturing, Automotive, Video
Surveillance, Retail, Gaming, Healthcare, Mining, Buildings and Smart
Cities.

1.1. Edge Data

Edge computing is motivated at least in part by the sheer volume of
data that is being created by endpoint devices (sensors, cameras,
lights, vehicles, drones, wearables, etc.) at the very network edge
and that flows upstream, in a direction for which the network was not
originally designed. In fact, in dense IoT deployments (e.g., many
video cameras are streaming high definition video), where multiple
data flows collect or converge at edge nodes, data is likely to need
transformation (to be transcoded, subsampled, compressed, analyzed,
annotated, combined, aggregated, etc.) to fit over the next hop link,
or even to fit in memory or storage. Note also that the act of
performing computation on the data creates yet another new data
stream! Preservation of the original data streams is needed
sometimes but not always.

In addition, data may be cached, copied and/or stored at multiple
locations in the network on route to its final destination. With an
increasing percentage of devices connecting to the Internet being
mobile, support for in-the-network caching and replication is
critical for continuous data availability, not to mention efficient
network and battery usage for endpoint devices.

Additionally, as mobile devices' memory/storage fill up, in an edge
context they may have the ability to offload their data to other
proximate devices or resources, leaving a bread crumb trail of data
in their wakes. Therefore, although data might originate at edge
devices, as more and more data is continuously created, processed and
stored, it becomes increasingly dispersed throughout the physical
world (outside of or scattered across managed local data centers),
increasingly isolated in separate local edge clouds or data silos.
Thus, there needs to be a standard way to find it. New and existing
protocols will need to be identified/developed/enhanced for these
purposes. Being able to discover distributed data at the edge or in
the middle of the network will be an important component of Edge
computing.

1.2. Background

   Several IETF T2T RG Edge Computing discussions have been held over
   the last couple years. A comparative study on the definition of Edge
   computing was presented in multiple sessions in T2T RG in 2018 and an
   Edge Computing I-D was submitted early 2019. An IETF BEC (beyond
   edge computing) effort has been evaluating potential gaps in existing
   edge computing architectures. Edge Data Discovery is one potential
   gap that was identified and that needs evaluation and a solution.
   The newly proposed COIN RG highlights the need for computations in
   the network to be able to marshal potentially distributed input data
   and to handle resultant output data, i.e., its placement, storage
   and/or possible migration strategy.

1.3. Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [RFC2119].

1.4. Terminology

   o Edge: The edge encompasses all entities not in the back-end cloud.
      The device edge represents the very leaves of the network and
      encompasses the entities found in the last mile network. Sensors,
      gateways, compute nodes are included. Because the things that
      populate the IoT can be both physical and/or cyber, in some
      solutions, particularly in software-defined or digital-twin
      contexts, the device edge can include logical (vs physical)
      entities. The infrastructure edge includes equipment on the
      network operator side of the last mile network including cell
      towers, edge data centers, cable headends, POPs, etc. See
      Figure 1 for other possible tiers of edge clouds between the
      device edge and the back-end cloud data center.

   o Edge Computing: Distributed computation that is performed near the
      network edge, where nearness is determined by the system
      requirements. This includes high performance compute, storage and
      network equipment on either the device or infrastructure edge.

   o Edge Data Discovery: The process of finding required data from
      edge entities, i.e., from databases, file systems, and device
      memory that might be physically distributed in the network, and
      providing access to it logically as if it were a single unified
      source, perhaps through its namespace, that can be evaluated or
      searched.

o ICN: Information Centric Networking. An ICN-enabled network
   routes data by name (vs address), caches content natively in the
   network, and employs data-centric security. Data discovery may
   require that data be associated with a name or names, a series of
   descriptive attributes, and/or a unique identifier.

2. Edge Data Discovery Problem Scope

   Our focus is on how to define and scope the edge data discovery
   problem. This requires some discussion of the evolving definition of
   the edge as part of a cloud-to-edge continuum and in turn what is
   meant by edge data, as well as the meta-data about the edge data.

2.1. A Cloud-Edge Continuum

   Although Edge Computing data typically originates at edge devices,
   there is nothing that precludes edge data from being created anywhere
   in the cloud-to-edge computing continuum (Figure 1). New edge data
   may result as a byproduct of computation being performed on the data
   stream anywhere along its path in the network. For example,
   infrastructure edges may create new edge data when multiple data
   streams converge upon this aggregation point and require
   transformation (e.g., to fit within the available resources, to
   smooth raw measurements to eliminate high-frequency noise, or to
   obfuscate data for privacy).

   Initially our focus is on discovery of edge data that resides at the
   Device Edge and the Infrastructure Edge.

```
                  +------------------------------+
                  | Back-end Cloud Data Center   |
                  +------------------------------+
                         ***   Cloud
                        *   * Interconnect
                         ***
                  +------------------------------+
                  | Core Data Center             |
                  +------------------------------+
                         ***   Backbone
                        *   * Network
                         ***
                  +------------------------------+
                  | Regional Data Center         |
                  +------------------------------+
                         ***   Metropolitan
                        *   * Network
                         ***
                  +------------------------------+
                  | Infrastructure Edge          |
                  +------------------------------+
                         ***   Access
                        *   * Network
                         ***
                  +------------------------------+
                  | Device Edge                  |
                  +------------------------------+
```
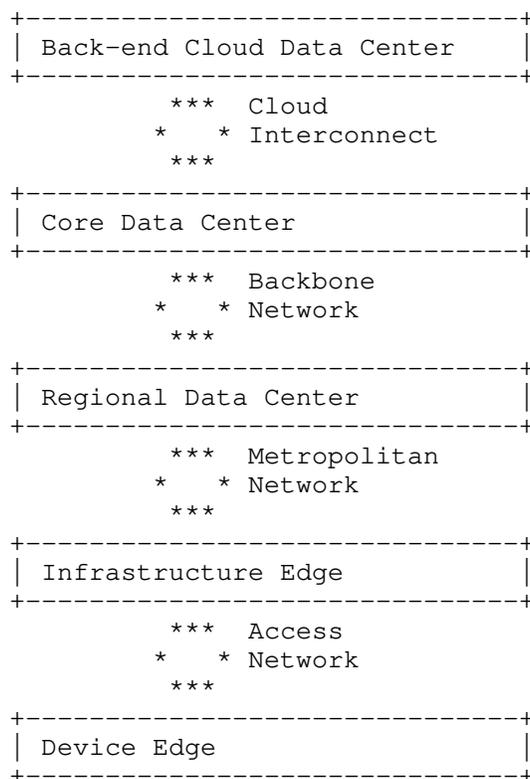
Figure 1: Cloud-to-edge computing continuum

2.2. Types of Edge Data

   Besides classically constrained IoT device sensor and measurement
   data accumulating throughout the edge computing infrastructure, edge
   data may also take the form of higher frequency and higher volume
   streaming data (from a continuous sensor or from a camera), meta data
   (about the data), control data (regarding an event that was
   triggered), and/or an executable that embodies a function, service,
   or any other piece of code or algorithm. Edge data also could be
   created after multiple streams converge at an edge node and are
   processed, transformed, or aggregated together in some manner.

   Regardless of edge data type, a key problem in the Cloud-Edge
   continuum is that data is often kept in silos. Meaning, data is
   often sequestered within the Edge where it was created. A goal of
   this discussion is to consider the prospect that different types of
   edge data will be made accessible across disparate edges, for example
   to enable richer multi-modal analytics. But this will happen only if

data can be described, searched and discovered across heterogeneous
edges in a standard way. Having a mechanism to enable granular edge
data discovery is the problem that needs solving either with existing
or new protocols. The mechanisms shouldn't care to which flavor
cloud or edge the request for data discovery is made.

3. Edge Scenarios Requiring Data Discovery

    1. A set of data resources appears (e.g., a mobile node hosting data
        joins a network) and they want to be discoverable by an existing
        but possibly virtualized and/or ephemeral data directory
        infrastructure.

    2. A device wants to discover data resources available at or near
        its current location. As some of these resources may be mobile,
        the available set of edge data may vary over time.

    3. A device wants to discover to where best in the edge
        infrastructure to opportunistically upload its data, for example
        if a mobile device wants to offload its data to the
        infrastructure (for greater data availability, battery savings,
        etc.).

4. Edge Data Discovery

    How can we discover data on the edge and make use of it? There are
    proprietary implementations that collect data from various databases
    and consolidate it for evaluation. We need a standard protocol set
    for doing this data discovery, on the device or infrastructure edge,
    in order to meet the requirements of many use cases. We will have
    terabytes of data on the edge and need a way to identify its
    existence and find the desired data. A user requires the need to
    search for specific data in a data set and evaluate it using their
    own tools. The tools are outside the scope of this document, but the
    discovery of that data is in scope.

4.1. Types of Discovery

    There are many aspects of discovery and many different protocols that
    address each aspect.

    Discovery of new devices added to an environment. Discovery of their
    capabilities/services in client/server environments. Discovery of
    these new devices automatically. Discovering a device and then
    synchronizing the device inventory and configuration for edge
    services. There are many existing protocols to help in this
    discovery: UPnP, mDNS, DNS-SD, SSDP, NFC, XMPP, W3C network service
    discovery, etc.

Edge devices discover each other in a standard way. We can use DHCP,
SNMP, SMS, COAP, LLDP, and routing protocols such as OSPF for devices
to discover one another.

Discovery of link state and traffic engineering data/services by
external devices. BGP-LS is one such solution.

The question is if one or more of these protocols might be a suitable
contender to extend to support edge data discovery?

4.2. Early Stage of Discovery

The different types of discovery may involve mobile devices, which
can be the source, or target, of discovery operations. Mobile
devices may have an influence on discovery in COIN, and early stage
discovery may be necessary in some scenarios.

In many cases (e.g. crowds, drones or vehicular scenarios), multiple
networks, or attachment points, are available to a mobile device.
This type of device needs to efficiently select among multiple
interfaces, or multiple attachment points, which one(s) to use for
discovery. An early discovery stage should provide enough
information to perform such a selection and therefore reduce power
consumption, service latency, and impact on network usage.

To select among (already attached) multiple interfaces, we can
leverage provisioning domains, router advertisements, DHCP, etc. to
convey information about service or data. To select among multiple
attachment points, pre-attachment discovery (e.g. 802.11aq, or
obtaining provisioning domains through a control plane) or a
discovery protocol over a control plane (e.g. as described in 3GPP
edge computing) can be used.

What are suitable protocols to extend to support this early stage of
discovery? There is also a tradeoff between the amount of exposed
information and the limited resources available at this early stage.
Trust and privacy are also important early stage discovery factors.

4.3. Naming the Data

Information-Centric Networking (ICN) RFC 7927 [RFC7927] is a class of
architectures and protocols that provide "access to named data" as a
first-order network service. Instead of host-to-host communication
as in IP networks, ICNs often use location-independent names to
identify data objects, and the network provides the services of
processing (answering) requests for named data with the objective to
finally deliver the requested data objects to a requesting consumer.

Such an approach has profound effects on various aspects of a
networking system, including security (by enabling object-based
security on a message/packet level), forwarding behavior (name-based
forwarding, caching), but also on more operational aspects such as
bootstrapping, discovery etc.

The CCNx and NDN (https://named-data.net) variants of ICN are based
on a request/response abstraction where consumers (hosts, application
requesting named data) send INTEREST messages into the network that
are forwarded by network elements to a destination that can provide
the requested named data object. Corresponding responses are sent as
so-called DATA messages that follow the reverse INTEREST path.

Each unique data object is named unambiguously in a hierarchical
naming scheme and is authenticated through Public-Key cryptography
(data objects can also optionally be encrypted in different ways).
The naming concept and the object-based security approach lay the
foundation for location-independent operation. The network can
generally operate without any notion of location, and nodes
(consumers, forwarders) can forward requests for named data objects
directly, i.e., without any additional address resolution. Location
independence also enables additional features, for example the
possibility to replicate and cache named data objects. On-path
caching is a standard feature in many ICN systems -- typically for
enhancing reliability and performance.

In CCNx and NDN, forwarders are stateful, i.e., they keep track of
forwarded INTEREST to later match the received DATA messages.
Stateful forwarding (in conjunction with the general named-based and
location-independent operation) also empowers forwarders to execute
individual forwarding strategies and perform optimizations such as
in-network retransmissions, multicasting requests (in cases there are
several opportunities for accessing a particular named data object)
etc.

Naming data and application-specific naming conventions are naturally
important aspects in ICN. It is common that applications define
their own naming convention (i.e., semantics of elements in the name
hierarchy). Such names can often be directly derived from
application requirements, for example a name like /my-home/living-
room/light/switch/main could be relevant in a smart home setting, and
corresponding devices and applications could use a corresponding
convention to facilitate controllers finding sensors and actors in
such a system with minimal user configuration.

The aforementioned features make ICN amenable to data discovery.
Because there is no name/address chasm as in IP-based systems, data
can be discovered by sending an INTEREST to named data objects

directly (assuming a naming convention as described above).
Moreover, ICN can authenticate received data objects directly, for
example using local trust anchors in the network (for example in a
home network).

Advanced ICN features for data discovery include the concept of
manifests in CCNx, i.e., ICN objects that describe data collections,
and data set synchronization protocols in NDN (https://named-
data.net/publications/li2018sync-intro/) that can inform consumers
about the availability of new data in a tree-based data structure
(with automatic retrieval and authentication). Also, ICN is not
limited to accessing static data. Frameworks such as Named Function
Networking (http://www.named-function.net) and RICE can provide the
general ICN feature for discovery not only for data but also for name
functions (for in-network computing) and for their results.

5. Use Cases of Edge Data Discovery

5.1. Autonomous Vehicles

   Autonomous vehicles rely on the processing of huge amounts of complex
   data in real-time for fast and accurate decisions. These vehicles
   will rely on high performance compute, storage and network resources
   to process the volumes of data they produce in a low latency way.
   Various systems will need a standard way to discover the pertinent
   data for decision making.

5.2. Video Surveillance

   The majority of the video surveillance footage will remain at the
   edge infrastructure (not sent to the cloud data center). This
   footage is coming from vehicles, factories, hotels, universities,
   farms, etc. Much of the video footage will not be interesting to
   those evaluating the data. A mechanism, perhaps a set of protocols,
   is needed to identify the interesting data at the edge. What
   constitutes interesting will be context specific, e.g., a video frame
   might be considered interesting if and only if it includes a car, or
   person, or bicyclist, or a backyard nocturnal creature, or etc.
   Interesting video data may be stored longer in storage systems at the
   very edge of the network and/or in networking equipment further away
   from the device edge that has access to data in flight as it transits
   the network.

5.3. Elevator Networks

   Elevators are one of many industrial applications of edge computing.
   Edge equipment receives data from hundreds of elevator sensors. The
   data coming into the edge equipment is vibration, temperature, speed,

level, video, etc. We need the ability to identify where the data we
need to evalute is located.

5.4. Service Function Chaining

Service function chaining (SFC) allows the instantiation of an
ordered set of service functions (SFs) and the subsequent "steering"
of traffic through them. Service functions are expected to be
deployed at the edge of the network, as a feasible deployment of
"Compute In the Network", with multiple types of potential use cases
(e.g., fog robotics, Industry 4.0 automation, etc). Service
functions provide a specific treatment of received packets, therefore
they need to be discoverable so they can be used in a given service
composition via SFC. In addition, these functions can be producers
and/or consumers of data. So far, how the functions are discovered
and composed has been out of the scope of discussions in the IETF.
While there are some mechanisms that can be used and/or extended to
provide this functionality, more work needs to be done. An example
of this can be found in [I-D.bernardos-sfc-discovery].

In an SFC environment deployed at the edge, the discovery protocol
may also need the following kind of meta-data information per
(service) function:

o Service Function Type: identifying the category of function
   provided.

o SFC-aware: Yes/No. Indicates if the function is SFC-aware.

o Route Distinguisher (RD): IP address indicating the location of
   the function.

o Pricing/costs details.

o Migration capabilities of the function: whether a given function
   can be moved to another provider (potentially including
   information about compatible providers topologically close).

o Mobility of the device hosting the function, with e.g. the
   following sub-options:

      Level: no, low, high; or a corresponding scale (e.g., 1 to 10).

      Current geographical area (e.g., GPS coordinates, post code).

      Target moving area (e.g., GPS coordinates, post code).

o Power source of the device hosting the function, with e.g. the
   following sub-options:

       Battery: Yes/No. If Yes, the following sub-options could be
       defined:

          Capacity of the battery (e.g., mmWh).

          Charge status (e.g., %).

          Lifetime (e.g., minutes).

Discovery of resources in an NFV environment: virtualized resources
do not need to be limited to those available in traditional data
centers, where the infrastructure is stable, static, typically
homogeneous and managed by a single admin entity. Computational
capabilities are becoming more and more ubiquitous, with terminal
devices getting extremely powerful, as well as other types of devices
that are close to the end users at the edge (e.g., vehicular onboard
devices for infotainment, micro data centers deployed at the edge,
etc.). It is envisioned that these devices would be able to offer
storage, computing and networking resources to nearby network
infrastructure, devices and things (the fog paradigm). These
resources can be used to host functions, for example to offload/
complement other resources available at traditional data centers, but
also to reduce the end-to-end latency or to provide access to
specialized information (e.g., context available at the edge) or
hardware. Similar to the discovery of functions, while there are
mechanisms that can be reused/extended, there is no complete solution
yet defined. An example of work in this area is
[I-D.bernardos-intarea-vim-discovery]. The availability of this
meta-data about the capabilities of nearby physical as well as
virtualized resources can be made discoverable through edge data
discovery mechanisms.

5.5. Ubiquitous Witness

Ubiquitous Witness (UW) is the name of a use case that has been
presented in past COINRG and ICNRG meetings at the IETF. It
describes what might occur in dense IoT deployments when an anomaly
occurs. There are many "witnesses" to report on what happened within
a limited region of interest and around an approximate point in time.
The use case highlights the need for upstream data discovery and
management. It is agnostic to where the dense IoT deployment
resides, whether in a factory, home, commercial building, city,
entertainment venue, et cetera. For example, as cameras and other
sensors have become ubiquitous in Smart Cities, it would be helpful
to be able to discover and examine data from all devices and sensors

that witnessed an accident in a city intersection; this could be data
from cameras mounted at the intersection itself, on nearby buildings,
in cars, and cell phones of individuals on location.

If an anomaly were to automatically trigger independent upstream
flows of video data from all of the witnesses (within a proximal
vicinity and time window), the data flows would naturally converge at
shared collection or aggregation points in the network. These edge
nodes might opt to vault any data deemed part of a safety-related
anomaly, which would enable interested parties (the car owner, the
car manufacturer, an insurance company, a city traffic planner) to
investigate the root cause of the anomaly after the fact. The
implication however is that enough meta data has been generated
alongside the data itself (e.g., a data name, an identifier, or a geo
location and timestamp), to allow the retrieval of this distributed
data, provided those asking have proper authorization to access it.

The UW streams are contextually-related and as such it can be
advantageous also to be able to process them simultaneously, at the
time they are first generated. For example if collection nodes could
derive that groups of data streams are contextually-related, they
could stitch streams together to create a 360-degree view of the
anomalous event (e.g., to walk around in the data), or to winnow the
set of vaulted data to only the "best" video (e.g., highest
resolution, unoccluded views) or to perform compute-in-the-network to
enable them to fit within the available resources (e.g., at the
receiving node due to the convergence or implosion of upstream data,
or over the next hoplink). Ubiquitous Witness data doesn't have to
be video data, but video illustrates why one might want to jointly
process upstream flows in real-time.

6. IANA Considerations

   N/A

7. Security Considerations

   Security considerations will be a critical component of edge data
   discovery particularly as intelligence is moved to the extreme edge
   where data is to be extracted.

   An assumption is that all data will have associated policies
   (default, inherited or configured) that describe access control
   permissions. Consequently, the discoverability of data will be a
   function of who or what has requested access. In other words, the
   discoverable view into the available data will be limited to those
   who are authorized. Discovering edge data that is exclusively
   private is out of scope of this document, the assumption being that

there will be some edge clouds that do not expose or publish the availability of their data. Although edge data may be sent to the back-end cloud as needed, there is nothing that precludes it from being discoverable if the cloud offers it as public.

A trust relationship may be needed between the source and target of a discovery operation to avoid denial of service attacks from a malicious source or target of the operation. And discovery information, which is exposed by a node or network, may need to be protected for privacy purposes, e.g. not leak information in the presence of a certain type of data in a network.

8. Acknowledgement

The authors thank Dave Oran, Greg Skinner and Lixia Zhang for contributing to this document.

9. Normative References

[I-D.bernardos-intarea-vim-discovery]
          Bernardos, C. and A. Mourad, "IPv6-based discovery and
          association of Virtualization Infrastructure Manager (VIM)
          and Network Function Virtualization Orchestrator (NFVO)",
          draft-bernardos-intarea-vim-discovery-04 (work in
          progress), September 2020.

[I-D.bernardos-sfc-discovery]
          Bernardos, C. and A. Mourad, "Service Function discovery
          in fog environments", draft-bernardos-sfc-discovery-05
          (work in progress), September 2020.

[I-D.irtf-icnrg-ccnxmessages]
          Mosko, M., Solis, I., and C. Wood, "CCNx Messages in TLV
          Format", draft-irtf-icnrg-ccnxmessages-09 (work in
          progress), January 2019.

[I-D.irtf-icnrg-ccnxsemantics]
          Mosko, M., Solis, I., and C. Wood, "CCNx Semantics",
          draft-irtf-icnrg-ccnxsemantics-10 (work in progress),
          January 2019.

[I-D.kutscher-icnrg-rice]
          Krol, M., Habak, K., Oran, D., Kutscher, D., and I.
          Psaras, "Remote Method Invocation in ICN", draft-kutscher-
          icnrg-rice-00 (work in progress), October 2018.

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119,
               DOI 10.17487/RFC2119, March 1997,
               <https://www.rfc-editor.org/info/rfc2119>.

   [RFC7927]   Kutscher, D., Ed., Eum, S., Pentikousis, K., Psaras, I.,
               Corujo, D., Saucez, D., Schmidt, T., and M. Waehlisch,
               "Information-Centric Networking (ICN) Research
               Challenges", RFC 7927, DOI 10.17487/RFC7927, July 2016,
               <https://www.rfc-editor.org/info/rfc7927>.

Authors' Addresses

   Mike McBride
   Futurewei

   Email: michael.mcbride@futurewei.com

   Dirk Kutscher
   Emden University

   Email: ietf@dkutscher.net

   Eve Schooler
   Intel

   Email: eve.m.schooler@intel.com
   URI: http://www.eveschooler.com

   Carlos J. Bernardos
   Universidad Carlos III de Madrid
   Av. Universidad, 30
   Leganes, Madrid 28911
   Spain

   Phone: +34 91624 6236
   Email: cjbc@it.uc3m.es
   URI: http://www.it.uc3m.es/cjbc/

   Diego R. Lopez
   Telefonica

   Email: diego.r.lopez@telefonica.com
   URI: https://www.linkedin.com/in/dr2lopez/

      Xavier de Foy
      InterDigital Communications, LLC
      1000 Sherbrooke West
      Montreal
      Canada

      Email: Xavier.Defoy@InterDigital.com

                   Explicit Flow Measurements Techniques
                 draft-mdt-ippm-explicit-flow-measurements-00

Abstract

   This document describes protocol independent methods called Explicit
   Flow Measurement Techniques that employ few marking bits, inside the
   header of each packet, for loss and delay measurement. The
   endpoints, marking the traffic, signal these metrics to intermediate
   observers allowing them to measure connection performance, and to
   locate the network segment where impairments happen. Different
   alternatives are considered within this document. These signaling
   methods apply to all protocols but they are especially valuable when
   applied to protocols that encrypt transport header and do not allow
   traditional methods for delay and loss detection.

Discussion Venues

   This note is to be removed before publishing as an RFC.

   Discussion of this document takes place on the IPPM Working Group
   mailing list (ippm@ietf.org), which is archived at
   https://mailarchive.ietf.org/arch/browse/ippm/.

   Source for this draft and an issue tracker can be found at
   https://github.com/igorlord/draft-xxx-ippm-flow-measurements.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on 6 May 2021.

Copyright Notice

Table of Contents

1. Introduction

   Packet loss and delay are hard and pervasive problems of day-to-day
   network operation. Proactively detecting, measuring, and locating
   them is crucial to maintaining high QoS and timely resolution of
   crippling end-to-end throughput issues. To this effect, in a TCP-
   dominated world, network operators have been heavily relying on
   information present in the clear in TCP headers: sequence and
   acknowledgment numbers and SACKs when enabled (see [RFC8517]). These
   allow for quantitative estimation of packet loss and delay by passive
   on-path observation. Additionally, the problem can be quickly
   identified in the network path by moving the passive observer around.

   With encrypted protocols, the equivalent transport headers are
   encrypted and passive packet loss and delay observations are not
   possible, as described in [TRANSPORT-ENCRYPT].

   Measuring TCP loss and delay between similar endpoints cannot be
   relied upon to evaluate encrypted protocol loss and delay. Different
   protocols could be routed by the network differently, and the
   fraction of Internet traffic delivered using protocols other than TCP
   is increasing every year. It is imperative to measure packet loss
   and delay experienced by encrypted protocol users directly.

   This document defines Explicit Flow Measurement Techniques. These
   hybrid measurement path signals (see [IPM-Methods]) are to be
   embedded into a transport layer protocol and are explicitly intended
   for exposing RTT and loss rate information to on-path measurement
   devices. These measurement mechanisms are applicable to any
   transport-layer protocol, and, as an example, the document describes
   QUIC and TCP bindings.

   The Explicit Flow Measurement Techniques described in this document
   can be used alone or in combination with other Explicit Flow
   Measurement Techniques. Each technique uses a small number of bits
   and exposes a specific measurement.

   Following the recommendation in [RFC8558] of making path signals
   explicit, this document proposes adding a small number of dedicated
   measurement bits to the clear portion of the protocol headers. These
   bits can be added to an encrypted portion of a header belonging to
   any protocol layer, e.g. IP (see [IP]) and IPv6 (see [IPv6]) headers
   or extensions, such as [IPv6AltMark], UDP surplus space (see
   [UDP-OPTIONS] and [UDP-SURPLUS]), reserved bits in a QUIC v1 header
   (see [QUIC-TRANSPORT]).

The measurements are not designed for use in automated control of the network in environments where signal bits are set by untrusted hosts. Instead, the signal is to be used for troubleshooting individual flows as well as for monitoring the network by aggregating information from multiple flows and raising operator alarms if aggregate statistics indicate a potential problem.

The spin bit, delay bit and loss bits explained in this document are inspired by [AltMark], [SPIN-BIT], [I-D.trammell-tsvwg-spin] and [I-D.trammell-ippm-spin].

Additional details about the Performance Measurements for QUIC are described in the paper [ANRW19-PM-QUIC].

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Latency Bits

This section introduces bits that can be used for round trip latecy measurements. Whenever this section of the specification refers to packets, it is referring only to packets with protocol headers that include the latency bits.

[QUIC-TRANSPORT] introduces an explicit per-flow transport-layer signal for hybrid measurement of RTT. This signal consists of a spin bit that toggles once per RTT. [SPIN-BIT] discusses an additional two-bit Valid Edge Counter (VEC) to compensate for loss and reordering of the spin bit and increase fidelity of the signal in less than ideal network conditions.

This document introduces an additional single-bit delay signal that can be used together with the spin bit by passive observers to measure the RTT of a network flow, avoiding the spin bit ambiguities that arise as soon as network conditions deteriorate.

3.1. Spin Bit

This section is a small recap of the spin bit working mechanism. For a comprehensive explanation of the algorithm, please see [SPIN-BIT].

The spin bit is an alternate marking [AltMark] generated signal, where the size of the alternation changes with the flight size each RTT.

The latency spin bit is a single bit signal that toggles once per
RTT, enabling latency monitoring of a connection-oriented
communication from intermediate observation points.

A "spin period" is a set of packets with the same spin bit value sent
during one RTT time interval. A "spin period value" is the value of
the spin bit shared by all packets in a spin period.

The client and server maintain an internal per-connection spin value
(i.e. 0 or 1) used to set the spin bit on outgoing packets. Both
endpoints initialize the spin value to 0 when a new connection
starts. Then:

* when the client receives a packet with the packet number larger
   than any number seen so far, it sets the connection spin value to
   the opposite value contained in the received packet;

* when the server receives a packet with the packet number larger
   than any number seen so far, it sets the connection spin value to
   the same value contained in the received packet.

The computed spin value is used by the endpoints for setting the spin
bit on outgoing packets. This mechanism allows the endpoints to
generate a square wave such that, by measuring the distance in time
between pairs of consecutive edges observed in the same direction, a
passive on-path observer can compute the round trip delay of that
network flow.

Spin bit enables round trip latency measurement by observing a single
direction of the traffic flow.

Note that packet reordering can cause spurious edges that require
heuristics to correct. The spin bit performance deteriorates as soon
as network impairments arise as explained in Section 3.2.

3.2. Delay Bit

The delay bit, different from a two-bit VEC, has been designed to
overcome accuracy limitations experienced by the spin bit under
difficult network conditions:

* packet reordering leads to generation of spurious edges and errors
   in delay estimation;

* loss of edges causes wrong estimation of spin periods and
   therefore wrong RTT measurements;

   * application-limited senders cause the spin bit to measure the
      application delays instead of network delays.

   If enabled, delay bit has to be used in addition to the spin bit.
   Unlike the spin bit, which is set in every packet transmitted on the
   network, the delay bit is set only once per round trip.

   When the delay bit is used, a single packet with a second marked bit
   (the delay bit) bounces between a client and a server during the
   entire connection lifetime. This single packet is called "delay
   sample".

   An observer placed at an intermediate point, observing a single
   direction of traffic, tracking the delay sample and the relative
   timestamp in every spin period, can measure the round trip delay of
   the connection.

   The delay sample lifetime is comprised of two phases: initialization
   and reflection. The initialization is the generation of the delay
   sample, while the reflection realizes the bounce behavior of this
   single packet between the two endpoints.

   The next figure describes the Delay bit mechanism: the first bit is
   the spin bit and the second one is the delay bit.
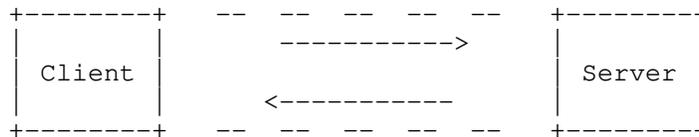
```
         +--------+   --  --  --  --  --   +--------+
         |        |        ----------->    |        |
         | Client |                        | Server |
         |        |        <-----------     |        |
         +--------+   --  --  --  --  --   +--------+

         (a) No traffic at beginning.

         +--------+   00  00  01  --  --   +--------+
         |        |        ----------->    |        |
         | Client |                        | Server |
         |        |        <-----------     |        |
         +--------+   --  --  --  --  --   +--------+

          (b) The Client starts sending data and
           sets the first packet as Delay Sample.

         +--------+   00  00  00  00  00   +--------+
         |        |        ----------->    |        |
         | Client |                        | Server |
         |        |        <-----------     |        |
         +--------+   --  --  01  00  00   +--------+
```
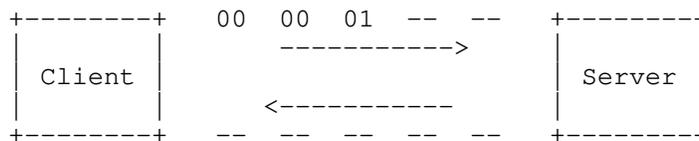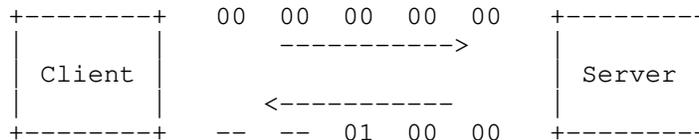
         (c) The Server starts sending data
          and reflects the Delay Sample.

```
+--------+    10  10  11  00  00   +--------+
|        |         ----------->    |        |
| Client |                         | Server |
|        |        <-----------     |        |
+--------+    00  00  00  00  00   +--------+
```
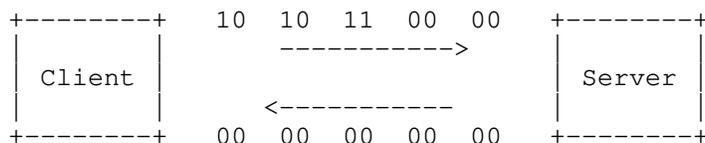
         (d) The Client inverts the spin bit and
          reflects the Delay Sample.

```
+--------+    10  10  10  10  10   +--------+
|        |         ----------->    |        |
| Client |                         | Server |
|        |        <-----------     |        |
+--------+    00  00  11  10  10   +--------+
```
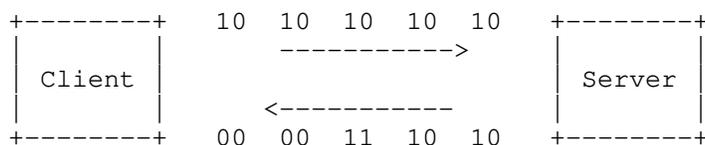
         (e) The Server reflects the Delay Sample.

```
+--------+    00  00  01  10  10   +--------+
|        |         ----------->    |        |
| Client |                         | Server |
|        |        <-----------     |        |
+--------+    10  10  10  10  10   +--------+
```
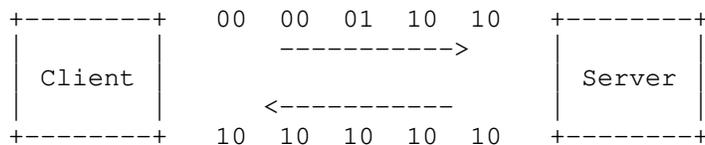
         (f) The client reverts the spin
          bit and reflects the Delay Sample.

                  Figure 1: Spin bit and Delay bit

3.2.1. Generation Phase

   Only client is actively involved in the generation phase.

   When connection starts and spin bit is set to 0, the client
   initializes the delay bit of the first packet to 1, so it becomes the
   delay sample for that marking period. Only this packet is marked
   with the delay bit set to 1 for this round trip period; the other
   ones will carry the spin bit, while the delay bit will be set to 0.

   The server initializes the delay bit to 0 at the beginning of the
   connection, and its only task during the connection is described in
   Section 3.2.2.

   In absence of network impairments, the delay sample should bounce
   between client and server continuously, for the entire duration of
   the connection. That is highly unlikely for two reasons:

1. the packet carrying the delay bit might be lost;

2. an endpoint could stop or delay sending packets because the
    application is limiting the amount of traffic transmitted;

To deal with these problems, the algorithm provides a procedure named
"recovery process" to regenerate the delay sample and to inform a
possible observer of the problem so the measurement can be restarted.

3.2.1.1. The Recovery Process

Absent packet loss or reordering, every spin period ends with a delay
sample inside. If that does not happen and a spin period terminates
without a delay sample inside, the client waits a further spin
period; then, it creates a new delay sample by setting the delay bit
to 1 on the first outgoing packet of the subsequent period.

The spin period with all delay bits set to 0 informs observers that
there was a problem and delay measurements for this flow should be
reset till the next delay sample is received.

3.2.2. Reflection Phase

Reflection is the process that enables the bouncing of the delay
sample between a client and a server. The behavior of the two
endpoints is slightly different.

* Server side reflection: when a delay sample arrives, the server
   marks the first packet in the opposite direction as the delay
   sample, if the outgoing packet has the same spin bit value as the
   delay sample. Otherwise, the delay sample is ignored.

* Client side reflection: when a delay sample arrives, the client
   marks the first packet in the opposite direction as the delay
   sample, if the outgoing packet has the opposite spin bit value
   then the delay sample. Otherwise, the delay sample is ignored.

In both cases, if the outgoing delay sample is being transmitted with
a delay greater than a predetermined threshold after the reception of
the incoming delay sample (1ms by default), the delay sample is not
reflected, and the outgoing delay bit is kept at 0.

Note that reflection takes place for the delay sample regardless of
its position within the spin period, as long as it stays within its
original spin period.

A time threshold for the retransmission of the delay sample is used
to eliminate measurements that would overestimate the delay due to
lack of traffic on the endpoints. Hence, the maximum estimation
error would amount to twice the threshold (e.g. 2ms) per measurement.

3.2.3. Two Bits Delay Measurement: Spin Bit + Delay Bit

When the Delay Bit is used, a passive observer can use delay samples
directly and avoid inherent ambiguities in the calculation of the RTT
in spin bit analysis, such as heuristic validation of the goodness of
an edge signal.

3.2.3.1. RTT Measurement

The delay sample generation process ensures that only one packet
marked with the delay bit set to 1 runs back and forth between two
endpoints per round trip time. To determine the RTT measurement of a
flow, an on-path passive observer computes the time difference
between two delay samples observed in a single direction.

To ensure a valid measurement, the observer must identify spin
periods in the packet flow and assign delay samples to spin periods.
If a spin period is missing a delay sample, the measurement needs to
be restarted from the subsequent delay sample. Hence, measurements
must take into account delay samples belonging to adjacent spin
periods.
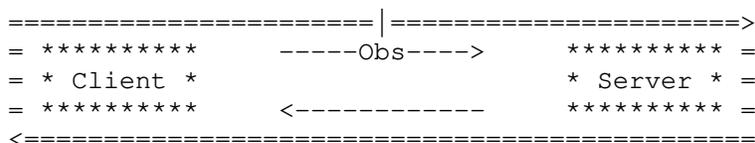
```
              =====================|=====================>
              = *********      ----Obs---->     ********* =
              = * Client *                      * Server * =
              = *********      <------------     ********* =
              <=============================================

                       (a) client-server RTT


              =============================================>
              = *********      ------------>     ********* =
              = * Client *                      * Server * =
              = *********      <----Obs-----     ********* =
              <=====================|=====================

                       (b) server-client RTT
```
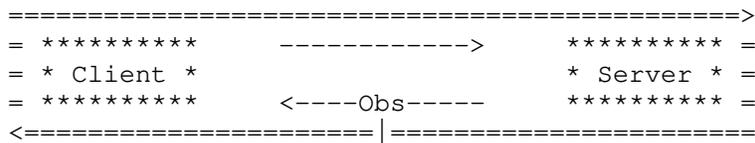
                Figure 2: Round-trip time (both direction)

3.2.3.2. Half-RTT Measurement

   An observer that is able to observe both forward and return traffic
   directions can use the delay samples to measure "upstream" and
   "downstream" RTT components, also known as the half-RTT measurements.
   It does this by measuring the time between a delay sample observed in
   one direction and the reflected delay sample observed in the opposite
   direction.

   As with RTT measurement, the observer must identify spin periods in
   the packet flow and assign delay samples to spin periods. If a spin
   period is missing a delay sample, the measurement needs to be
   restarted from the subsequent delay sample. So a measurement, to be
   valid, must take into account delay samples belonging to adjacent
   periods, for the upstream component, or to the same period for the
   downstream component.

   Note that upstream and downstream sections of paths between the
   endpoints and the observer, i.e. observer-to-client vs client-to-
   observer and observer-to-server vs server-to-observer, may have
   different delay characteristics due to the difference in network
   congestion and other factors.

```
                    =======================>
       = **********       ------|----->      **********
       = * Client *          Obs            * Server *
       = **********       <-----|------      **********
       <======================
```
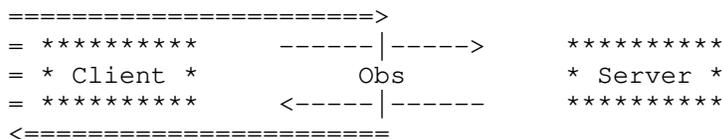
                (a) client-observer half-RTT

```
                              =======================>
         **********       ------|----->      ********** =
         * Client *          Obs            * Server * =
         **********       <-----|------      ********** =
                              <======================
```
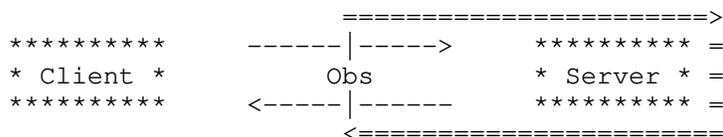
                (b) observer-server half-RTT

              Figure 3: Half Round-trip time (both direction)

3.2.3.3. Intra-Domain RTT Measurement

   Intra-domain RTT is the portion of the entire RTT used by a flow to
   traverse the network of a provider. To measure intra-domain RTT, two
   observers capable of observing traffic in both directions must be
   employed simultaneously at ingress and egress of the network to be
   measured. Intra-domain RTT is difference between the two computed
   upstream (or downstream) RTT components.

```
             ========================================>
             = ====================>
             = = **********      ---|-->        ---|-->      **********
             = = * Client *         Obs            Obs       * Server *
             = = **********      <--|---        <--|---      **********
             = <====================
             <========================================
```
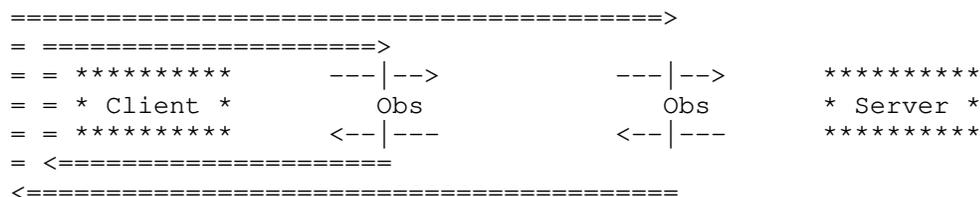
              (a) client-observer RTT components (half-RTTs)

```
                           ===================>
             **********      ---|-->        ---|-->      **********
             * Client *         Obs            Obs       * Server *
             **********      <--|---        <--|---      **********
                           <===================
```
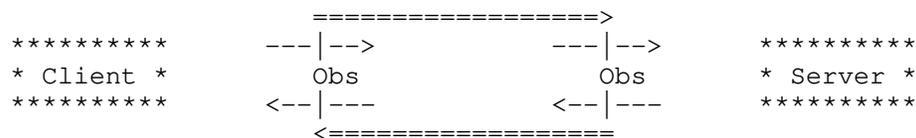
              (b) the intra-domain RTT resulting from the
                  subtraction of the above RTT components

      Figure 4: Intra-domain Round-trip time (client-observer: upstream)

3.2.4. Observer's Algorithm and Edge Rejection Interval

   To provide a formal description of the observer behavior, we define a
   "matching delay sample" to be a delay sample with the spin bit value
   that matched the spin bit value of then-current spin period.

   Upon detecting a matching delay sample, if a matching delay sample
   was also detected in the previous period, then the two delay samples
   can be used to calculate RTT measurement.

   If the observer can observe both forward and return traffic flows,
   and it is able to determine which direction contains the client and
   the server (e.g. by observing the spin bit or connection handshake):

   * If matching delay samples have been detected in both directions in
      the current spin period, they can be used to measure the observer-
      server half-RTT.

   * If a matching delay sample has been detected in client-to-observer
      direction, AND a matching delay sample had been detected in
      observer-to-client direction in the previous spin period, they can
      be used to measure the observer-client half-RTT.

   The described observer behavior depends on the ability to accurately
   identify current spin periods and to reject spurious spin edges,
   caused by packet reordering. Failure to do so will lead to many
   missed measurement opportunities and will decrease the amount of
   usable delay samples available to the observer.

   To implement spurious edge rejection, every time a spin bit edge is
   detected, the observer starts a new spin period and begins a time
   interval during which it rejects spin edges (e.g. 5ms). This
   guarantees protection against spurious edges due to packets that have
   been reordered by less than the time interval. The mechanism only
   works for intervals smaller than the RTT of the observed connection;
   if RTT is smaller than the edge rejection interval, the observer
   cannot measure the RTT.

4. Loss Bits

   This section introduces bits that can be used for loss measurements.
   Whenever this section of the specification refers to packets, it is
   referring only to packets with protocol headers that include the loss
   bits - the only packets whose loss can be measured.

   * T: the "round Trip loss" bit is used in combination with the Spin
      bit to measure round-trip loss. See Section 4.1.

   * Q: the "sQuare signal" bit is used to measure upstream loss. See
      Section 4.2.

   * L: the "Loss event" bit is used to measure end-to-end loss. See
      Section 4.3.

   * R: the "Reflection square signal" bit is used in combination with
      Q bit to measure end-to-end loss. See Section 4.1.

   Loss measurements enabled by T, Q, and L bits can be implemented by
   those loss bits alone (T bit requires a working Spin Bit). Two-bit
   combinations Q+L and Q+R enable additional measurement opportunities
   discussed below.

   Each endpoint maintains appropriate counters independently and
   separately for each separately identifiable flow (each sub-flow for
   multipath connections).

Since loss is reported independently for each flow, all bits (except for L bit) require a certain minimum number of packets to be exchanged per flow before any signal can be measured. Therefore, loss measurements work best for flows that transfer more than a minimal amount of data.

4.1. T Bit - Round Trip Loss Bit

The round Trip loss bit is used to mark a variable number of packets exchanged twice between the endpoints realizing a two round-trip reflection. A passive on-path observer, observing either direction, can count and compare the number of marked packets seen during the two reflections, estimating the loss rate experienced by the connection. The overall exchange comprises:

* The client selects, generates and consequently transmits a first train of packets, by setting the T bit to 1;

* The server, upon receiving each packet included in the first train, reflects to the client a respective second train of packets of the same size as the first train received, by setting the T bit to 1;

* The client, upon receiving each packet included in the second train, reflects to the server a respective third train of packets of the same size as the second train received, by setting the T bit to 1;

* The server, upon receiving each packet included in the third train, finally reflects to the client a respective fourth train of packets of the same size as the third train received, by setting the T bit to 1.

Packets belonging to the first round trip (first and second train) represent the Generation Phase, while those belonging to the second round trip (third and fourth train) represent the Reflection Phase.

A passive on-path observer can count and compare the number of marked packets seen during the two round trips (i.e. the first and third or the second and the fourth trains of packets, depending on which direction is observed) and estimate the loss rate experienced by the connection. This process is repeated continuously to obtain more measurements as long as the endpoints exchange traffic. These measurements can be called Round Trip losses.

Since packet rates in two directions may be different, the number of marked packets in the train is determined by the direction with the lowest packet rate. See Section 4.1.2 for details on packet

generation and for a mechanism to allow an observer to distinguish
between trains belonging to different phases (Generation and
Reflection).

4.1.1. Round Trip Packet Loss Measurement

Since the measurements are performed on a portion of the traffic
exchanged between the client and the server, the observer calculates
the end-to-end Round Trip Packet Loss (RTPL) that, statistically,
will correspond to the loss rate experienced by the connection along
the entire network path.

```
                  ======================|=====================>
                  = **********     -----Obs---->     ********** =
                  = * Client *                       * Server * =
                  = **********     <------------      ********** =
                  <=================================================
```
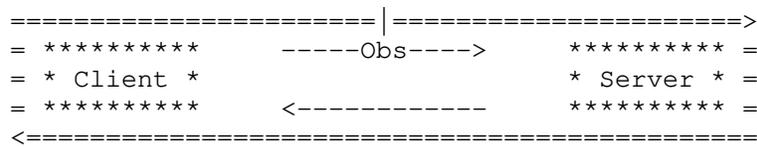
                         (a) client-server RTPL


```
                  ================================================>
                  = **********     ------------>     ********** =
                  = * Client *                       * Server * =
                  = **********     <----Obs-----     ********** =
                  <=====================|=====================
```
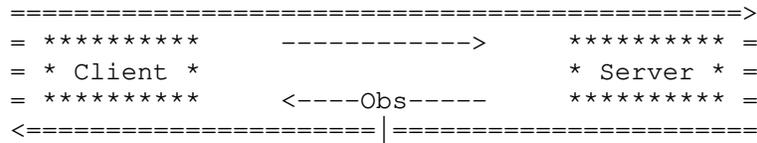
                         (b) server-client RTPL

                  Figure 5: Round-trip packet loss (both direction)

This methodology also allows the Half-RTPL measurement and the Intra-
domain RTPL measurement in a way similar to RTT measurement.

```
                  ========================>
                  = **********     ------|----->     **********
                  = * Client *          Obs          * Server *
                  = **********     <-----|------      **********
                  <========================
```
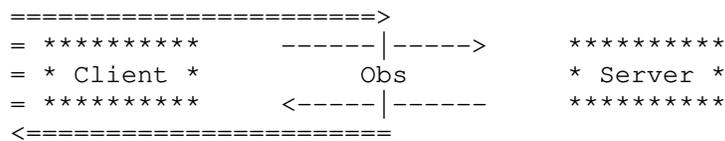
                         (a) client-observer half-RTPL


```
                                    ========================>
                  **********     ------|----->     ********** =
                  * Client *          Obs          * Server * =
                  **********     <-----|------      ********** =
                                    <========================
```
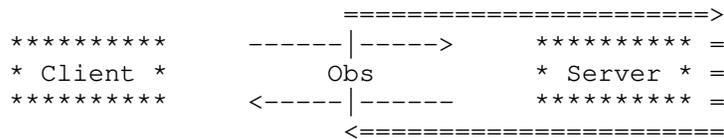
                         (b) observer-server half-RTPL

                    Figure 6: Half Round-trip packet loss (both direction)

```
                        ==========================================>
                        ====================> =
    **********        ---|-->           ---|-->      ********** = =
    * Client *           Obs               Obs       * Server * = =
    **********        <--|---           <--|---       ********** = =
                                        <==================== =
                        <==========================================
```
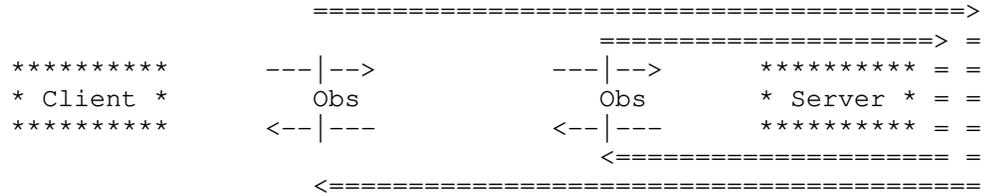
                    (a) observer-server RTPL components (half-RTPLs)

```
                        ===================>
    **********        ---|-->           ---|-->      **********
    * Client *           Obs               Obs       * Server *
    **********        <--|---           <--|---       **********
                        <==================
```
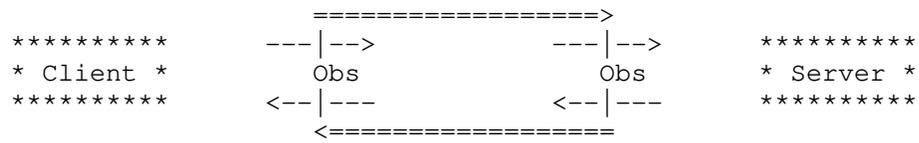
                        (b) the intra-domain RTPL resulting from the
                        subtraction of the above RTPL components

            Figure 7: Intra-domain Round-trip packet loss (observer-server)

4.1.2. Setting the Round Trip Loss Bit on Outgoing Packets

   The round Trip loss signal requires a working Spin-bit signal to
   separate trains of marked packets (packets with T bit set to 1). A
   "pause" of at least one empty spin-bit period between each phase of
   the algorithm serves as such separator for the on-path observer.

   The client is in charge of launching trains of marked packets and
   does so according to the algorithm:

   1. Generation Phase. The client starts generating marked packets
       for two consecutive spin-bit periods; it maintains a "generation
       token" count that is reset to zero at the beginning of the
       algorithm phase and is incremented every time a packet arrives.
       When the client transmits a packet and a "generation token" is
       available, the client marks the packet and retires a "generation
       token". If no token is available, the outgoing packet is
       transmitted unmarked. At the end of the first spin-bit period
       spent in generation, the reflection counter is unlocked to start
       counting incoming marked packets that will be reflected later;

   2. Pause Phase. When the generation is completed, the client pauses
       till it has observed one entire spin bit period with no marked
       packets. That spin bit period is used by the observer as a
       separator between generated and reflected packets. During this

marking pause, all the outgoing packets are transmitted with T
bit set to 0. The reflection counter is still incremented every
time a marked packet arrives;

3. Reflection Phase. The client starts transmitting marked packets,
    decrementing the reflection counter for each transmitted marked
    packet until the reflection counter reached zero. The
    "generation token" method from the generation phase is used
    during this phase as well. At the end of the first spin-period
    spent in reflection, the reflection counter is locked to avoid
    incoming reflected packets incrementing it;

4. Pause Phase 2. The pause phase is repeated after the reflection
    phase and serves as a separator between the reflected packet
    train and a new packet train.

The generation token counter should be capped to limit the effects of
a subsequent sudden reduction in the other endpoint's packet rate
that could prevent that endpoint from reflecting collected packets.
The most conservative cap value is "1".

A server maintains a "marking counter" that starts at zero and is
incremented every time a marked packet arrives. When the server
transmits a packet and the "marking counter" is positive, the server
marks the packet and decrements the "marking counter". If the
"marking counter" is zero, the outgoing packet is transmitted
unmarked.

4.1.3. Observer's Logic for Round Trip Loss Signal

The on-path observer counts marked packets and separates different
trains by detecting spin-bit periods (at least one) with no marked
packets. The Round Trip Packet Loss (RTPL) is the difference between
the size of the Generation train and the Reflection train.

In the following example, packets are represented by two bits (first
one is the spin bit, second one is the loss bit):

```
        Generation              Pause              Reflection            Pause
 _____  _____  _____  _____
|                       ||                ||                       ||         |
 01 01 00 01 11 10 11 00 00 10 10 10 01 00 01 01 10 11 10 00 00 10
```
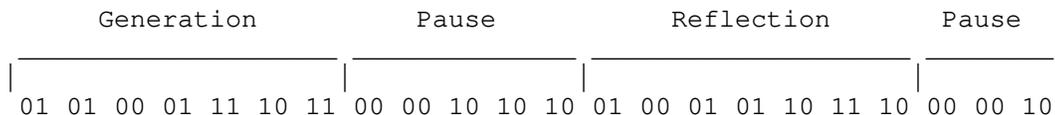
Figure 8: Round Trip Loss signal example

Note that 5 marked packets have been generated of which 4 have been
reflected.

4.1.4. Loss Coverage and Signal Timing

   A cycle of the round Trip loss signaling algorithm contains 2 RTTs of
   Generation phase, 2 RTTs of Reflection phase, and two Pause phases at
   least 1 RTT in duration each. Hence, the loss signal is delayed by
   about 6 RTTs since the loss events.

   The observer can only detect loss of marked packets that occurs after
   its initial observation of the Generation phase and before its
   subsequent observation of the Reflection phase. Hence, if the loss
   occurs on the path that sends packets at a lower rate (typically ACKs
   in such asymmetric scenarios), "2/6" ("1/3") of the packets will be
   sampled for loss detection.

   If the loss occurs on the path that sends packets at a higher rate,
   "lowPacketRate/(3*highPacketRate)" of the packets will be sampled for
   loss detection. For protocols that use ACKs, the portion of packets
   sampled for loss in the higher rate direction during unidirectional
   data transfer is "1/(3*packetsPerAck)", where the value of
   "packetsPerAck" can vary by protocol, by implementation, and by
   network conditions.

4.2. Q Bit - Square Bit

   The sQuare bit (Q bit) takes its name from the square wave generated
   by its signal. Every outgoing packet contains the Q bit value, which
   is initialized to the 0 and inverted after sending N packets (a
   sQuare Block or simply Q Block). Hence, Q Period is 2*N. The Q bit
   represents "packet color" as defined by [AltMark].

   Observation points can estimate upstream losses by watching a single
   direction of the traffic flow and counting the number of packets in
   each observed Q Block, as described in Section 4.2.2.

4.2.1. Q Block Length Selection

   The length of the block must be known to the on-path network probes.
   There are two alternatives to selecting the Q Block length. The
   first one requires that the length is known a priori and therefore
   set within the protocol specifications that implements the marking
   mechanism. The second requires the sender to select it.

   In this latter scenario, the sender is expected to choose N (Q Block
   length) based on the expected amount of loss and reordering on the
   path. The choice of N strikes a compromise - the observation could
   become too unreliable in case of packet reordering and/or severe loss
   if N is too small, while short flows may not yield a useful upstream
   loss measurement if N is too large (see Section 4.2.2).

   The value of N should be at least 64 and be a power of 2. This
   requirement allows an Observer to infer the Q Block length by
   observing one period of the square signal. It also allows the
   Observer to identify flows that set the loss bits to arbitrary values
   (see Section 7).

   If the sender does not have sufficient information to make an
   informed decision about Q Block length, the sender should use N=64,
   since this value has been extensively tried in large-scale field
   tests and yielded good results. Alternatively, the sender may also
   choose a random power-of-2 N for each flow, increasing the chances of
   using a Q Block length that gives the best signal for some flows.

   The sender must keep the value of N constant for a given flow.

4.2.2. Upstream Loss

   Blocks of N (Q Block length) consecutive packets are sent with the
   same value of the Q bit, followed by another block of N packets with
   an inverted value of the Q bit. Hence, knowing the value of N, an
   on-path observer can estimate the amount of upstream loss after
   observing at least N packets. The upstream loss rate ("uloss") is
   one minus the average number of packets in a block of packets with
   the same Q value ("p") divided by N ("uloss=1-avg(p)/N").

   The observer needs to be able to tolerate packet reordering that can
   blur the edges of the square signal, as explained in Section 4.2.3.

```
                 =====================>
                 **********     -----Obs---->     **********
                 * Client *                       * Server *
                 **********     <------------      **********

             (a) in client-server channel (uloss_up)

                 **********     ------------>      **********
                 * Client *                       * Server *
                 **********     <----Obs-----      **********
                                <=====================

             (b) in server-client channel (uloss_down)
```
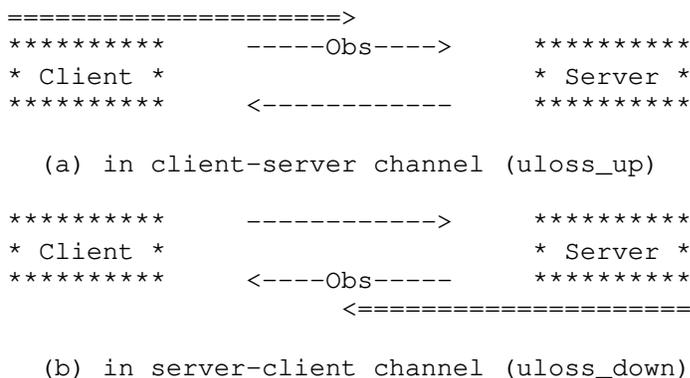
                       Figure 9: Upstream loss

4.2.3. Identifying Q Block Boundaries

   Packet reordering can produce spurious edges in the square signal.
   To address this, the observer should look for packets with the
   current Q bit value up to X packets past the first packet with a
   reverse Q bit value. The value of X, a "Marking Block Threshold",
   should be less than "N/2".

   The choice of X represents a trade-off between resiliency to
   reordering and resiliency to loss. A very large Marking Block
   Threshold will be able to reconstruct Q Blocks despite a significant
   amount of reordring, but it may erroneously coalesce packets from
   multiple Q Blocks into fewer Q Blocks, if loss exceeds 50% for some Q
   Blocks.

4.3. L Bit - Loss Event Bit

   The Loss Event bit uses an Unreported Loss counter maintained by the
   protocol that implements the marking mechanism. To use the Loss
   Event bit, the protocol must allow the sender to identify lost
   packets. This is true of protocols such as QUIC, partially true for
   TCP and SCTP (losses of pure ACKs are not detected) and is not true
   of protocols such as UDP and IP/IPv6.

   The Unreported Loss counter is initialized to 0, and L bit of every
   outgoing packet indicates whether the Unreported Loss counter is
   positive (L=1 if the counter is positive, and L=0 otherwise).

   The value of the Unreported Loss counter is decremented every time a
   packet with L=1 is sent.

   The value of the Unreported Loss counter is incremented for every
   packet that the protocol declares lost, using whatever loss detection
   machinery the protocol employs. If the protocol is able to rescind
   the loss determination later, a positive Unreported Loss counter may
   be decremented due to the rescission, but it should NOT become
   negative due to the rescission.

   This loss signaling is similar to loss signaling in [ConEx], except
   the Loss Event bit is reporting the exact number of lost packets,
   whereas Echo Loss bit in [ConEx] is reporting an approximate number
   of lost bytes.

For protocols, such as TCP ([TCP]), that allow network devices to
change data segmentation, it is possible that only a part of the
packet is lost. In these cases, the sender must increment Unreported
Loss counter by the fraction of the packet data lost (so Unreported
Loss counter may become negative when a packet with L=1 is sent after
a partial packet has been lost).

Observation points can estimate the end-to-end loss, as determined by
the upstream endpoint, by counting packets in this direction with the
L bit equal to 1, as described in Section 4.3.1.

4.3.1. End-To-End Loss

The Loss Event bit allows an observer to estimate the end-to-end loss
rate by counting packets with L bit value of 0 and 1 for a given
flow. The end-to-end loss rate is the fraction of packets with L=1.

The assumption here is that upstream loss affects packets with L=0
and L=1 equally. If some loss is caused by tail-drop in a network
device, this may be a simplification. If the sender's congestion
controller reduces the packet send rate after loss, there may be a
sufficient delay before sending packets with L=1 that they have a
greater chance of arriving at the observer.

4.3.2. Loss Profile Characterization

In addition to measuring the end-to-end loss rate, the Loss Event bit
allows an observer to characterize loss profile, since the
distribution of observed packets with L bit set to 1 roughly
corresponds to the distribution of packets lost between 1 RTT and 1
RTO before (see Section 4.4.1). Hence, observing random single
instances of L bit set to 1 indicates random single packet loss,
while observing blocks of packets with L bit set to 1 indicates loss
affecting entire blocks of packets.

4.4. L+Q Bits - Upstream, Downstream, and End-to-End Loss Measurements

Combining L and Q bits allows a passive observer watching a single
direction of traffic to accurately measure:

* upstream loss: sender-to-observer loss (see Section 4.2.2)

* downstream loss: observer-to-receiver loss (see Section 4.4.1.1)

* end-to-end loss: sender-to-receiver loss on the observed path (see
   Section 4.3.1) with loss profile characterization (see
   Section 4.3.2)

4.4.1. Correlating End-to-End and Upstream Loss

   Upstream loss is calculated by observing packets that did not suffer
   the upstream loss (Section 4.2.2). End-to-end loss, however, is
   calculated by observing subsequent packets after the sender's
   protocol detected the loss. Hence, end-to-end loss is generally
   observed with a delay of between 1 RTT (loss declared due to multiple
   duplicate acknowledgments) and 1 RTO (loss declared due to a timeout)
   relative to the upstream loss.

   The flow RTT can sometimes be estimated by timing protocol handshake
   messages. This RTT estimate can be greatly improved by observing a
   dedicated protocol mechanism for conveying RTT information, such as
   the Spin bit (see Section 3.1) or Delay bit (see Section 3.2).

   Whenever the observer needs to perform a computation that uses both
   upstream and end-to-end loss rate measurements, it should use
   upstream loss rate leading the end-to-end loss rate by approximately
   1 RTT. If the observer is unable to estimate RTT of the flow, it
   should accumulate loss measurements over time periods of at least 4
   times the typical RTT for the observed flows.

   If the calculated upstream loss rate exceeds the end-to-end loss rate
   calculated in Section 4.3.1, then either the Q Period is too short
   for the amount of packet reordering or there is observer loss,
   described in Section 4.4.1.2. If this happens, the observer should
   adjust the calculated upstream loss rate to match end-to-end loss
   rate, unless the following applies.

   In case of a protocol like TCP and SCTP that does not track losses of
   pure ACK packets, observing a direction of traffic dominated by pure
   ACK packets could result in measured upstream loss that is higher
   than measured end-to-end loss, if said pure ACK packets are lost
   upstream. Hence, if the measurement is applied to such protocols,
   and the observer can confirm that pure ACK packets dominate the
   observed traffic direction, the observer should adjust the calculated
   end-to-end loss rate to match upstream loss rate.

4.4.1.1. Downstream Loss

   Because downstream loss affects only those packets that did not
   suffer upstream loss, the end-to-end loss rate ("eloss") relates to
   the upstream loss rate ("uloss") and downstream loss rate ("dloss")
   as "(1-uloss)(1-dloss)=1-eloss". Hence, "dloss=(eloss-
   uloss)/(1-uloss)".

4.4.1.2. Observer Loss

   A typical deployment of a passive observation system includes a
   network tap device that mirrors network packets of interest to a
   device that performs analysis and measurement on the mirrored
   packets. The observer loss is the loss that occurs on the mirror
   path.

   Observer loss affects upstream loss rate measurement, since it causes
   the observer to account for fewer packets in a block of identical Q
   bit values (see Section 4.2.2). The end-to-end loss rate
   measurement, however, is unaffected by the observer loss, since it is
   a measurement of the fraction of packets with the L bit value of 1,
   and the observer loss would affect all packets equally (see
   Section 4.3.1).

   The need to adjust the upstream loss rate down to match end-to-end
   loss rate as described in Section 4.4.1 is an indication of the
   observer loss, whose magnitude is between the amount of such
   adjustment and the entirety of the upstream loss measured in
   Section 4.2.2. Alternatively, a high apparent upstream loss rate
   could be an indication of significant packet reordering, possibly due
   to packets belonging to a single flow being multiplexed over several
   upstream paths with different latency characteristics.

4.5. R Bit - Reflection Square Bit

   R bit requires a deployment alongside Q bit. Unlike the square
   signal for which packets are transmitted into blocks of fixed size,
   the Reflection square signal (being an alternate marking signal too)
   produces blocks of packets whose size varies according to these
   rules:

   * when the transmission of a new block starts, its size is set equal
      to the size of the last Q Block whose reception has been
      completed;

   * if, before transmission of the block is terminated, the reception
      of at least one further Q Block is completed, the size of the
      block is updated to the average size of the further received Q
      Blocks. Implementation details follow.

   The Reflection square value is initialized to 0 and is applied to the
   R-bit of every outgoing packet. The Reflection square value is
   toggled for the first time when the completion of a Q Block is
   detected in the incoming square signal (produced by the opposite node
   using the Q-bit). When this happens, the number of packets ("p"),
   detected within this first Q Block, is used to generate a reflection

square signal which toggles every "M=p" packets (at first). This new
signal produces blocks of M packets (marked using the R-bit) and each
of them is called "Reflection Block" (R Block).

The M value is then updated every time a completed Q Block in the
incoming square signal is received, following this formula:
"M=round(avg(p))".

The parameter "avg(p)" is the average number of packets in a marking
period computed considering all the Q Blocks received since the
beginning of the current R Block.

To ensure a proper computation of the M value, endpoints implementing
the R bit must identify the boundaries of incoming Q Blocks. The
same approach described in {#endmarkingblock} should be used.

Looking at the R-bit, unidirectional observation points have an
indication of losses experienced by the entire unobserved channel
plus those occurred in the path from the sender up to them.

Since the Q Block is sent in one direction, and the corresponding
reflected R Block is sent in the opposite direction, the reflected R
signal is transmitted with the packet rate of the slowest direction.
Namely, if the observed direction is the slowest, there can be
multiple Q Blocks transmitted in the unobserved direction before a
complete R Block is transmitted in the observed direction. If the
unobserved direction is the slowest, the observed direction can be
sending R Blocks of the same size repeatedly before it can update the
signal to account for a newly-completed Q Block.

4.5.1. R+Q Bits - Using R and Q Bits for Passive Loss Measurement

Since both sQuare and Reflection square bits are toggled at most
every N packets (except for the first transition of the R-bit as
explained before), an on-path observer can count the number of
packets of each marking block and, knowing the value of N, can
estimate the amount of loss experienced by the connection. An
observer can calculate different measurements depending on whether it
is able to observe a single direction of the traffic or both
directions.

Single directional observer:

*   upstream loss in the observed direction: the loss between the
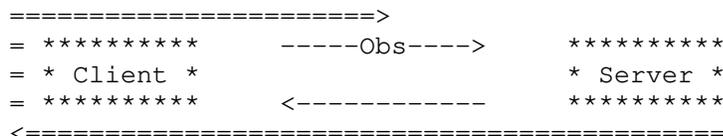   sender and the observation point (see Section 4.2.2)

   * "three-quarters" connection loss: the loss between the receiver
      and the sender in the unobserved direction plus the loss between
      the sender and the observation point in the observed direction

   * end-to-end loss in the unobserved direction: the loss between the
      receiver and the sender in the opposite direction

   Two directions observer (same metrics seen previously applied to both
   direction, plus):

   * client-observer half round-trip loss: the loss between the client
      and the observation point in both directions

   * observer-server half round-trip loss: the loss between the
      observation point and the server in both directions

   * downstream loss: the loss between the observation point and the
      receiver (applicable to both directions)

4.5.1.1. Three-Quarters Connection Loss

   Except for the very first block in which there is nothing to reflect
   (a complete Q Block has not been yet received), packets are
   continuously R-bit marked into alternate blocks of size lower or
   equal than N. Knowing the value of N, an on-path observer can
   estimate the amount of loss occurred in the whole opposite channel
   plus the loss from the sender up to it in the observation channel.
   As for the previous metric, the "three-quarters" connection loss rate
   ("tqloss") is one minus the average number of packets in a block of
   packets with the same R value ("t") divided by "N"
   ("tqloss=1-avg(t)/N").

```
                  =======================>
             = *********      -----Obs---->      *********
             = * Client *                        * Server *
             = *********      <------------       *********
             <=========================================

                 (a) in client-server channel (tqloss_up)


                 =========================================>
             *********      ------------>      ********* =
             * Client *                        * Server * =
             *********      <----Obs-----       ********* =
                             <=======================

                 (b) in server-client channel (tqloss_down)
```
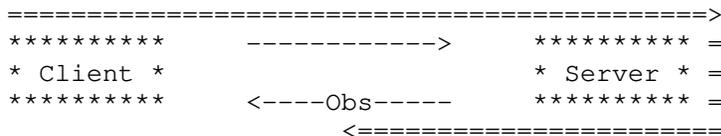
                Figure 10: Three-quarters connection loss

   The following metrics derive from this last metric and the upstream
   loss produced by the Q Bit.

4.5.1.2. End-To-End Loss in the Opposite Direction

   End-to-end loss in the unobserved direction ("eloss_unobserved")
   relates to the "three-quarters" connection loss ("tqloss") and
   upstream loss in the observed direction ("uloss") as
   "(1-eloss_unobserved)(1-uloss)=1-tqloss". Hence,
   "eloss_unobserved=(tqloss-uloss)/(1-uloss)".
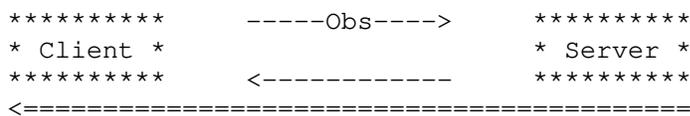
```
              **********     -----Obs---->     **********
              * Client *                       * Server *
              **********     <------------      **********
              <=========================================

          (a) in client-server channel (eloss_down)

              =========================================>
              **********     ------------>     **********
              * Client *                       * Server *
              **********     <----Obs-----      **********

          (b) in server-client channel (eloss_up)
```
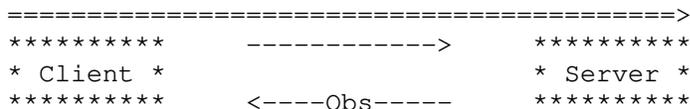
         Figure 11: End-To-End loss in the opposite direction

4.5.1.3. Half Round-Trip Loss

   If the observer is able to observe both directions of traffic, it is
   able to calculate two "half round-trip" loss measurements - loss from
   the observer to the receiver (in a given direction) and then back to
   the observer in the opposite direction. For both directions, "half
   round-trip" loss ("hrtloss") relates to "three-quarters" connection
   loss ("tqloss_opposite") measured in the opposite direction and the
   upstream loss ("uloss") measured in the given direction as
   "(1-uloss)(1-hrtloss)=1-tqloss_opposite". Hence,
   "hrtloss=(tqloss_opposite-uloss)/(1-uloss)".

```
                 ========================>
            = **********      ------|----->      **********
            = * Client *          Obs            * Server *
            = **********      <-----|------       **********
              <=======================
```
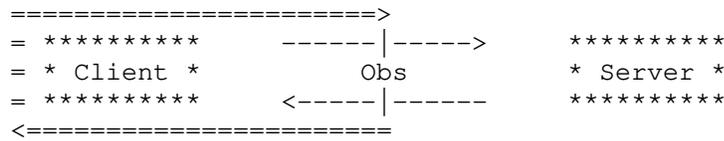
        (a) client-observer half round-trip loss (hrtloss_co)

```
                                   ========================>
                 **********      ------|----->      ********** =
                 * Client *          Obs            * Server * =
                 **********      <-----|------       ********** =
                                   <=======================
```
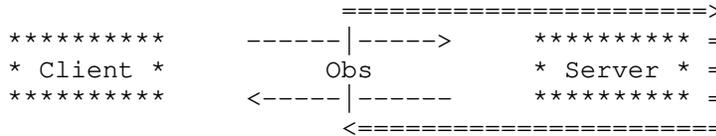
        (b) observer-server half round-trip loss (hrtloss_os)

            Figure 12: Half Round-trip loss (both direction)

4.5.1.4. Downstream Loss

   If the observer is able to observe both directions of traffic, it is
   able to calculate two downstream loss measurements using either end-
   to-end loss and upstream loss, similar to the calculation in
   Section 4.4.1.1 or using "half round-trip" loss and upstream loss in
   the opposite direction.

   For the latter, "dloss=(hrtloss-uloss_opposite)/(1-uloss_opposite)".

```
                               =====================>
                 **********      ------|----->      **********
                 * Client *          Obs            * Server *
                 **********      <-----|------       **********
```
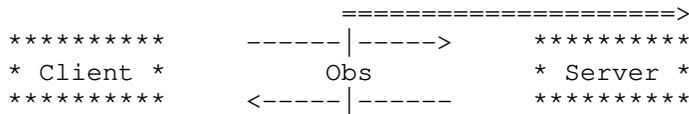
            (a) in client-server channel (dloss_up)

```
                 **********      ------|----->      **********
                 * Client *          Obs            * Server *
                 **********      <-----|------       **********
              <=====================
```
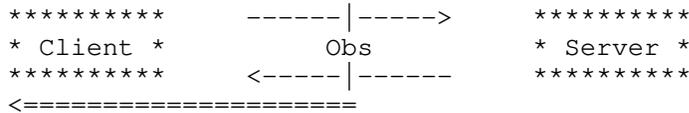
            (b) in server-client channel (dloss_down)

                    Figure 13: Downstream loss

4.5.2. Enhancement of R Block Length Computation

   The use of the rounding function used in the M computation introduces
   errors that can be minimized by storing the rounding applied each
   time M is computed, and using it during the computation of the M
   value in the following R Block.

   This can be achieved introducing the new "r_avg" parameter in the
   computation of M. The new formula is "Mr=avg(p)+r_avg; M=round(Mr);
   r_avg=Mr-M" where the initial value of "r_avg" is equal to 0.

4.5.3. Improved Resilience to Packet Reordering

   When a protocol implementing the marking mechanism is able to detect
   when packets are received out of order, it can improve resilience to
   packet reordering beyond what is possible using methods described in
   Section 4.2.3.

   This can be achieved by updating the size of the current R Block
   while this is being transmitted. The reflection block size is then
   updated every time an incoming reordered packet of the previous Q
   Block is detected. This can be done if and only if the transmission
   of the current reflection block is in progress and no packets of the
   following Q Block have been received.

5. Summary of Delay and Loss Marking Methods

   This section summarizes the marking methods described in this draft.

   For the Delay measurement, it is possible to use the spin bit alone
   or combined with the delay bit. A unidirectional or bidirectional
   observer can be used.

Method	# of bits	Available Delay Metrics		Impairments Resiliency
		UNIDIR Observer	BIDIR Observer	
S: Spin Bit	1	RTT	x2 Half RTT	lower
SD: Spin Bit + Delay Bit	2	RTT	x2 Half RTT	higher

   x2 Same metric for both directions.

                       Figure 14: Delay Comparison

   For the Loss measurement, each row in the table of Figure 15
   represents a loss marking method. For each method the table
   specifies the number of bits required in the header, the available
   metrics using an unidirectional or bidirectional observer, applicable
   protocols, measurement fidelity and delay.

Method	Bits	Available Loss Metrics		Prto	Measurement Aspects	
		UNIDIR Observer	BIDIR Observer		Fidelity	Delay
T: Round Trip Loss Bit	$ 1	RT	x2 Half RT	*	Rate by sampling 1/3 to 1/(3*ppa) of pkts over 2 RTT	~6 RTT
Q: Square Bit	1	Upstream	x2	*	Rate over N pkts (e.g. 64)	N pkts (e.g. 64)
L: Loss Event Bit	1	E2E	x2	#	Loss shape (and rate)	Min: RTT Max: RTO
QL: Square + Loss Ev. Bits	2	Upstream Downstream E2E	x2 x2 x2	#	-> see Q -> see Q|L -> see L	Up: see Q Others: see L
QR: Square + Ref. Sq. Bits	2	Upstream 3/4 RT !E2E	x2 x2 E2E Downstream Half RT	*	Rate over N*ppa pkts (see Q bit for N)	Up: see Q Others: N*ppa pk (see Q for N)

   *   All protocols
   #   Protocols employing loss detection (w/ or w/o pure ACK loss
       detection)
   $ Require a working spin bit
   !   Metric relative to the opposite channel
   x2 Same metric for both directions
   ppa Packets-Per-Ack
   Q|L See Q if Upstream loss is significant; L otherwise

                       Figure 15: Loss Comparison

6. ECN-Echo Event Bit

   While the primary focus of the draft is on exposing packet loss and
   delay, modern networks can report congestion before they are forced
   to drop packets, as described in [ECN]. When transport protocols
   keep ECN-Echo feedback under encryption, this signal cannot be
   observed by the network operators. When tasked with diagnosing
   network performance problems, knowledge of a congestion downstream of
   an observation point can be instrumental.

   If downstream congestion information is desired, this information can
   be signaled with an additional bit.

   * E: The "ECN-Echo Event" bit is set to 0 or 1 according to the
      Unreported ECN Echo counter, as explained below in Section 6.1.

6.1. Setting the ECN-Echo Event Bit on Outgoing Packets

   The Unreported ECN-Echo counter operates identically to Unreported
   Loss counter (Section 4.3), except it counts packets delivered by the
   network with CE markings, according to the ECN-Echo feedback from the
   receiver.

   This ECN-Echo signaling is similar to ECN signaling in [ConEx]. ECN-
   Echo mechanism in QUIC provides the number of packets received with
   CE marks. For protocols like TCP, the method described in
   [ConEx-TCP] can be employed. As stated in [ConEx-TCP], such feedback
   can be further improved using a method described in [ACCURATE].

6.2. Using E Bit for Passive ECN-Reported Congestion Measurement

   A network observer can count packets with CE codepoint and determine
   the upstream CE-marking rate directly.

   Observation points can also estimate ECN-reported end-to-end
   congestion by counting packets in this direction with a E bit equal
   to 1.

   The upstream CE-marking rate and end-to-end ECN-reported congestion
   can provide information about downstream CE-marking rate. Presence
   of E bits along with L bits, however, can somewhat confound precise
   estimates of upstream and downstream CE-markings in case the flow
   contains packets that are not ECN-capable.

7. Protocol Ossification Considerations

   Accurate loss and delay information is not critical to the operation
   of any protocol, though its presence for a sufficient number of flows
   is important for the operation of networks.

   The delay and loss bits are amenable to "greasing" described in
   [RFC8701], if the protocol designers are not ready to dedicate (and
   ossify) bits used for loss reporting to this function. The greasing
   could be accomplished similarly to the Latency Spin bit greasing in
   [QUIC-TRANSPORT]. Namely, implementations could decide that a
   fraction of flows should not encode loss and delay information and,
   instead, the bits would be set to arbitrary values. The observers
   would need to be ready to ignore flows with delay and loss
   information more resembling noise than the expected signal.

8. Examples of Application

8.1. QUIC

   The binding of the delay bit signal to QUIC is partially described in
   [QUIC-TRANSPORT], which adds the spin bit to the first byte of the
   short packet header, leaving two reserved bits for future
   experiments.

   To implement the additional signals discussed in this document, the
   first byte of the short packet header can be modified as follows:

   * the delay bit (D) can be placed in the first reserved bit (i.e.
      the fourth most significant bit _0x10_) while the loss bit (L) in
      the second reserved bit (i.e. the fifth most significant bit
      _0x08_); the proposed scheme is:

```
        0 1 2 3 4 5 6 7
       +-+-+-+-+-+-+-+-+
       |0|1|S|D|L|K|P|P|
       +-+-+-+-+-+-+-+-+
```

                        Figure 16: Scheme 1

   * alternatively, a two bits loss signal (QL or QR) can be placed in
      both reserved bits; the proposed schemes, in this case, are:

```
        0 1 2 3 4 5 6 7
       +-+-+-+-+-+-+-+-+
       |0|1|S|Q|L|K|P|P|
       +-+-+-+-+-+-+-+-+
```

                      Figure 17: Scheme 2A

            0 1 2 3 4 5 6 7
           +-+-+-+-+-+-+-+-+
           |0|1|S|Q|R|K|P|P|
           +-+-+-+-+-+-+-+-+

                      Figure 18: Scheme 2B

8.2. TCP

   The signals can be added to TCP by defining bit 4 of byte 13 of the
   TCP header to carry the spin bit, and eventually bits 5 and 6 to
   carry additional information, like the delay bit and the round-trip
   loss bit, or a two bits loss signal (QL or QR).

9. Security Considerations

   Passive loss and delay observations have been a part of the network
   operations for a long time, so exposing loss and delay information to
   the network does not add new security concerns for protocols that are
   currently observable.

   In the absence of packet loss, Q and R bits signals do not provide
   any information that cannot be observed by simply counting packets
   transiting a network path. In the presence of packet loss, Q and R
   bits will disclose the loss, but this is information about the
   environment and not the endpoint state. The L bit signal discloses
   internal state of the protocol's loss detection machinery, but this
   state can often be gleamed by timing packets and observing congestion
   controller response.

   Hence, loss bits do not provide a viable new mechanism to attack data
   integrity and secrecy.

9.1. Optimistic ACK Attack

   A defense against an Optimistic ACK Attack, described in
   [QUIC-TRANSPORT], involves a sender randomly skipping packet numbers
   to detect a receiver acknowledging packet numbers that have never
   been received. The Q bit signal may inform the attacker which packet
   numbers were skipped on purpose and which had been actually lost (and
   are, therefore, safe for the attacker to acknowledge). To use the Q
   bit for this purpose, the attacker must first receive at least an
   entire Q Block of packets, which renders the attack ineffective
   against a delay-sensitive congestion controller.

A protocol that is more susceptible to an Optimistic ACK Attack with
the loss signal provided by Q bit and uses a loss-based congestion
controller, should shorten the current Q Block by the number of
skipped packets numbers. For example, skipping a single packet
number will invert the square signal one outgoing packet sooner.

Similar considerations apply to the R Bit, although a shortened R
Block along with a matching skip in packet numbers does not
necessarily imply a lost packet, since it could be due to a lost
packet on the reverse path along with a deliberately skipped packet
by the sender.

10. Privacy Considerations

   To minimize unintentional exposure of information, loss bits provide
   an explicit loss signal - a preferred way to share information per
   [RFC8558].

   New protocols commonly have specific privacy goals, and loss
   reporting must ensure that loss information does not compromise those
   privacy goals. For example, [QUIC-TRANSPORT] allows changing
   Connection IDs in the middle of a connection to reduce the likelihood
   of a passive observer linking old and new sub-flows to the same
   device. A QUIC implementation would need to reset all counters when
   it changes the destination (IP address or UDP port) or the Connection
   ID used for outgoing packets. It would also need to avoid
   incrementing Unreported Loss counter for loss of packets sent to a
   different destination or with a different Connection ID.

11. IANA Considerations

   This document makes no request of IANA.

12. Change Log

   TBD

13. Contributors

   The following people provided valuable contributions to this
   document:

   * Marcus Ihlar, Ericsson, marcus.ihlar@ericsson.com

   * Jari Arkko, Ericsson, jari.arkko@ericsson.com

   * Emile Stephan, Orange, emile.stephan@orange.com

14. Acknowledgements

   TBD

15. References

15.1. Normative References

   [ConEx]   Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx)
              Concepts, Abstract Mechanism, and Requirements", RFC 7713,
              DOI 10.17487/RFC7713, December 2015,
              <https://www.rfc-editor.org/info/rfc7713>.

   [ConEx-TCP]
              Kuehlewind, M., Ed. and R. Scheffenegger, "TCP
              Modifications for Congestion Exposure (ConEx)", RFC 7786,
              DOI 10.17487/RFC7786, May 2016,
              <https://www.rfc-editor.org/info/rfc7786>.

   [ECN]   Ramakrishnan, K., Floyd, S., and D. Black, "The Addition
              of Explicit Congestion Notification (ECN) to IP",
              RFC 3168, DOI 10.17487/RFC3168, September 2001,
              <https://www.rfc-editor.org/info/rfc3168>.

   [IP]   Postel, J., "Internet Protocol", STD 5, RFC 791,
              DOI 10.17487/RFC0791, September 1981,
              <https://www.rfc-editor.org/info/rfc791>.

   [IPM-Methods]
              Morton, A., "Active and Passive Metrics and Methods (with
              Hybrid Types In-Between)", RFC 7799, DOI 10.17487/RFC7799,
              May 2016, <https://www.rfc-editor.org/info/rfc7799>.

   [IPv6]   Deering, S. and R. Hinden, "Internet Protocol, Version 6
              (IPv6) Specification", STD 86, RFC 8200,
              DOI 10.17487/RFC8200, July 2017,
              <https://www.rfc-editor.org/info/rfc8200>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

   [RFC8558] Hardie, T., Ed., "Transport Protocol Path Signals",
              RFC 8558, DOI 10.17487/RFC8558, April 2019,
              <https://www.rfc-editor.org/info/rfc8558>.

   [TCP]   Postel, J., "Transmission Control Protocol", STD 7,
              RFC 793, DOI 10.17487/RFC0793, September 1981,
              <https://www.rfc-editor.org/info/rfc793>.

15.2. Informative References

   [ACCURATE] Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More
              Accurate ECN Feedback in TCP", Work in Progress, Internet-
              Draft, draft-ietf-tcpm-accurate-ecn-12, 28 October 2020,
              <http://www.ietf.org/internet-drafts/draft-ietf-tcpm-
              accurate-ecn-12.txt>.

   [AltMark] Fioccola, G., Ed., Capello, A., Cociglio, M., Castaldelli,
              L., Chen, M., Zheng, L., Mirsky, G., and T. Mizrahi,
              "Alternate-Marking Method for Passive and Hybrid
              Performance Monitoring", RFC 8321, DOI 10.17487/RFC8321,
              January 2018, <https://www.rfc-editor.org/info/rfc8321>.

   [ANRW19-PM-QUIC]
              Bulgarella, F., Cociglio, M., Fioccola, G., Marchetto, G.,
              and R. Sisto, "Performance measurements of QUIC
              communications", DOI 10.1145/3340301.3341127, Proceedings
              of the Applied Networking Research Workshop, July 2019,
              <https://doi.org/10.1145/3340301.3341127>.

   [I-D.trammell-ippm-spin]
              Trammell, B., "An Explicit Transport-Layer Signal for
              Hybrid RTT Measurement", Work in Progress, Internet-Draft,
              draft-trammell-ippm-spin-00, 9 January 2019,
              <http://www.ietf.org/internet-drafts/draft-trammell-ippm-
              spin-00.txt>.

   [I-D.trammell-tsvwg-spin]
              Trammell, B., "A Transport-Independent Explicit Signal for
              Hybrid RTT Measurement", Work in Progress, Internet-Draft,
              draft-trammell-tsvwg-spin-00, 2 July 2018,
              <http://www.ietf.org/internet-drafts/draft-trammell-tsvwg-
              spin-00.txt>.

   [IPv6AltMark]
              Fioccola, G., Zhou, T., Cociglio, M., Qin, F., and R.
              Pang, "IPv6 Application of the Alternate Marking Method",
              Work in Progress, Internet-Draft, draft-ietf-6man-ipv6-
              alt-mark-02, 13 October 2020, <http://www.ietf.org/
              internet-drafts/draft-ietf-6man-ipv6-alt-mark-02.txt>.

   [QUIC-TRANSPORT]
             Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed
             and Secure Transport", Work in Progress, Internet-Draft,
             draft-ietf-quic-transport-32, 20 October 2020,
             <http://www.ietf.org/internet-drafts/draft-ietf-quic-
             transport-32.txt>.

   [RFC8517] Dolson, D., Ed., Snellman, J., Boucadair, M., Ed., and C.
             Jacquenet, "An Inventory of Transport-Centric Functions
             Provided by Middleboxes: An Operator Perspective",
             RFC 8517, DOI 10.17487/RFC8517, February 2019,
             <https://www.rfc-editor.org/info/rfc8517>.

   [RFC8701] Benjamin, D., "Applying Generate Random Extensions And
             Sustain Extensibility (GREASE) to TLS Extensibility",
             RFC 8701, DOI 10.17487/RFC8701, January 2020,
             <https://www.rfc-editor.org/info/rfc8701>.

   [SPIN-BIT] Trammell, B., Vaere, P., Even, R., Fioccola, G., Fossati,
             T., Ihlar, M., Morton, A., and S. Emile, "Adding Explicit
             Passive Measurability of Two-Way Latency to the QUIC
             Transport Protocol", Work in Progress, Internet-Draft,
             draft-trammell-quic-spin-03, 14 May 2018,
             <http://www.ietf.org/internet-drafts/draft-trammell-quic-
             spin-03.txt>.

   [TRANSPORT-ENCRYPT]
             Fairhurst, G. and C. Perkins, "Considerations around
             Transport Header Confidentiality, Network Operations, and
             the Evolution of Internet Transport Protocols", Work in
             Progress, Internet-Draft, draft-ietf-tsvwg-transport-
             encrypt-17, 8 September 2020, <http://www.ietf.org/
             internet-drafts/draft-ietf-tsvwg-transport-encrypt-
             17.txt>.

   [UDP-OPTIONS]
             Touch, J., "Transport Options for UDP", Work in Progress,
             Internet-Draft, draft-ietf-tsvwg-udp-options-08, 12
             September 2019, <http://www.ietf.org/internet-drafts/
             draft-ietf-tsvwg-udp-options-08.txt>.

   [UDP-SURPLUS]
             Herbert, T., "UDP Surplus Header", Work in Progress,
             Internet-Draft, draft-herbert-udp-space-hdr-01, 8 July
             2019, <http://www.ietf.org/internet-drafts/draft-herbert-
             udp-space-hdr-01.txt>.

Authors' Addresses

   Mauro Cociglio
   Telecom Italia
   Via Reiss Romoli, 274
   10148 Torino
   Italy

   Email: mauro.cociglio@telecomitalia.it

   Alexandre Ferrieux
   Orange Labs

   Email: alexandre.ferrieux@orange.com

   Giuseppe Fioccola
   Huawei Technologies
   Riesstrasse, 25
   80992 Munich
   Germany

   Email: giuseppe.fioccola@huawei.com

   Igor Lubashev
   Akamai Technologies

   Email: ilubashe@akamai.com

   Fabio Bulgarella
   Telecom Italia
   Via Reiss Romoli, 274
   10148 Torino
   Italy

   Email: fabio.bulgarella@guest.telecomitalia.it

   Isabelle Hamchaoui
   Orange Labs

   Email: isabelle.hamchaoui@orange.com

Massimo Nilo
Telecom Italia

Email: massimo.nilo@telecomitalia.it

Riccardo Sisto
Politecnico di Torino

Email: riccardo.sisto@polito.it

Dmitri Tikhonov
LiteSpeed Technologies

Email: dtikhonov@litespeedtech.com

TLS Working Group   P. Urien
Internet Draft   Telecom Paris
Intended status: Experimental

                                                       July 30 2020

Expires: January 2020

                  Identity Module for TLS Version 1.3
                      draft-urien-tls-im-03.txt

Abstract

   TLS 1.3 will be deployed in the Internet of Things ecosystem. In
   many IoT frameworks, TLS or DTLS protocols, based on pre-shared key
   (PSK), are used for device authentication. So PSK tamper resistance,
   is a critical market request, in order to prevent hijacking issues.
   If DH exchange is used with certificate bound to DH ephemeral public
   key, there is also a benefit to protect its signature procedure. The
   TLS identity module (im) MAY be based on secure element; it realizes
   some HKDF operations bound to PSK, and cryptographic signature if
   certificates are used. Secure Element form factor could be
   standalone chip, or embedded in SOC like eSIM.

Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119.

   .

Copyright Notice

Table of Contents

1 Overview

   TLS 1.3 [RFC8446] will be deployed in the Internet of Things
   ecosystem. In many IoT frameworks, TLS or DTLS protocols, based on
   pre-shared key (PSK), are used for device authentication. So PSK
   tamper resistance, is a critical market request, in order to prevent
   hijacking issues. If DH exchange is used with certificate bound to
   DH ephemeral public key, there is also a benefit to protect its
   signature procedure. The TLS identity module (im) MAY be based on
   secure element [ISO7816]; it realizes some HKDF [RFC5869] operations
   bound to PSK, and cryptographic signature if certificates are used.
   Secure Element form factor could be standalone chip or embedded in
   SOC like eSIM.

```
            +-----------+        +----------+
            | Processor |        | Identity |
            |  TLS 1.3  +------+  Module    |
            |           |      |  |    im    |
            +-----------+        +----------+
```
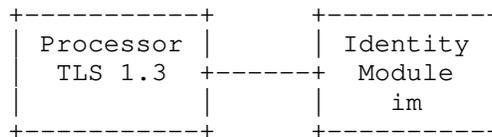
   Figure 1. TLS 1.3 Identity Module (im)

2 Protecting the Key Schedule for PSK

2.1 Context

   According to [RFC8446] external PSKs MAY be provisioned outside of
   TLS.

   The Early Secret (ESK) is computed according to relation:
   ESK =HKDF-Extract(salt=0s,PSK) = HMAC(salt=0s,PSK)

   The Binder Key (BSK) for outside provisioning is computed according
   to the relation:
   BSK = Derive-Secret(ESK, "ext binder", "")

   The Derived Secret (DSK) is computed according to the relation:
   DSK= Derive-Secret(ESK, "derived", "")

   The Finished External Key (FEK) is computed according to the
   relation:
   FEK = KDF-Expand-Label(BSK, "finished", "", Hash.length)

   For Derive-Secret procedures, "" is equivalent to the value
   hash(empty), whose size is hash-length.

2.2 Identity Module Procedures

   The identity module MUST provide a KSGS (Keys Secure Generation and
   Storage) procedure, which computes and securely stores ESK, BSK and
   FEK keys.

   This procedure MUST require administrative rights.

   A set IMKP (Identity Module Key Procedures) of four procedures is
   required, in order to protect from public exposure ESK, BSK, and
   FEK:
   - CETS: Client Early Traffic Secret
   - EEMS: Early Exporter Master Secret
   - HEDSK: HKDF-Extract from Derived Secret Key
   - HBSK: HMAC from Binder Key Secret

   These procedures MAY require user rights.

2.3 KSGS: Keys Secure Generation and Storage

   The Identity module MUST provide a KSGS procedure, requiring
   administrative rights, which computes and securely stores ESK, BSK,
   DSK, and FEK

   Input: salt and PSK
   Output: Success or Failure

   ESK, DSK, and BSK secret values are stored in the identity module

   HL16 : hash Length, 16 bits
   HL8 : hash length, 8 bits
   H0   : hash(empty)

   ESK= HMAC(salt=0s,PSK)
   DSK= HMAC(ESK,HL16||0d746c7331332064657269766564||HL8||H0||01)
   BSK= HMAC(ESK,HL16||10746c7331332065787874742062696e646572||HL8||H0||01)
   FEK= HMAC(BSK,HL16||0E746C7331332066696E69736865640001)

2.4 Identity Module Key Procedures (IMKP)

  2.4.1 CETS: Client Early Traffic Secret

   Input: Length, Message
   Output: Client Early Traffic Secret or Failure

   CETS(ClientHello) = Derive-Secret(ESK, "c e traffic", Message)
   = HMAC(ESK, Length || 11746c733133206320652074726166666963 ||
   Message || 01)

   Message is a hash value.

   2.4.2 EEMS: Early Exporter Master Secret

    Input: Length, Message
    Output: Early Exporter Master Secret or Failure

    EEMS(ClientHello) = Derive-Secret(ESK, "e exp master", Message)
    = HMAC(ESK, Length || 12746c73313320652065787020206d6173746572 ||
    Message || 01)

    Message is a hash value

   2.4.3 HEDSK: HKDF-Extract from Derived Secret Key

    Input: DHE
    Output: Handshake Secret or Failure

    HEDSK(DHE)= HKDF-Extract(salt=DSK,DHE) = HMAC(DSK, DHE)

   2.4.4 HBSK: HMAC from Binder Key Secret

    Input: data
    Output: HMAC(BSK, data) or Failure

    HBSK(data) = HMAC(FEK, data)

    Data is a hash value

3. Asymmetric Signature

    The identity module MUST provide a GENKEY (GENKEY: Generate Key)
    procedure, in order to store or generate private asymmetric key and
    associated public key.

    This procedure MUST require administrative rights.

    The procedure GETPUB (GETPUB: Get Public Key) is required in order
    to read the public key value.

    This procedure MAY require user rights.

    The procedure SIGN (SIGN: Signature) is required in order to perform
    a raw signature for a digest value, computed from certificate.

    This procedure MAY require user rights.

3.1 GENKEY

    Input: None
    Output: Success or Failure

   A private key is generated and store in the identity module. A
   public key is computed from the private key.

3.2 GETPUB

   Input: None
   Output: Public Key Value or Failure

3.3 SIGN

   Input: DigestValue
   Output: Signature Value or Failure

4. Secure Element as Identity Module

   Secure elements are defined according to [ISO7816] standards. They
   support hash functions (sha256, sha384, sha512) and associated HMAC
   procedures. They also provide DH procedures in Z/pZ* groups, and
   elliptic curves. Open software can be released thanks to the
   Javacard standards, such as JC3.04, JC3.05.

   Below is an illustration of binary encoding rules for secure element
   according to the T=1 ISO7816 protocol.

   An ISO7816 command (TAPDU) is a set of bytes comprising a five byte
   header and an optional payload (up to 255 bytes)

   The header comprises the following five bytes
   - CLA, Class
   - INS, Instruction code
   - P1, P1 byte
   - P2, P2 byte
   - P3, length of the payload, or number of expected bytes

   The response comprises a payload (up to 256 bytes) and a two bytes
   status word (SW1, SW2), 9000 meaning successful operation.

4.1 Administrative mode

   The [ISO7816] command VERIFY (INS=0x20) SHOULD be used to enter the
   administrative mode

   Tx: CLA=00 INS=20 P1=00 P2=Adm P3=PIN-Length [PIN-Value]
   Rx: 9000

4.2 User Mode

   The [ISO7816] command VERIFY SHOULD be used to enter the user mode

   Tx: CLA=00 INS=20 P1=00 P2=User P3=PIN-Length [PIN-Value]
   Rx: 9000

4.3 KSGS: Keys Secure Generation and Storage

   Length= 2 + Salt-Length + PSK-Length

   Tx: CLA=00 INS=TLS13 P1=0 P2=KSGS P3=Length Salt-Length [Salt-Value]
   PSK-Length [PSK-Value]
   Rx: 9000

   This procedure computes and stores ESK, BSK DSK and FEK.

   4.3.1 Example

   PSK=0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20

   Tx: CLA=00 INS=85 P1=00 P2=0A P3=23 01 00 20
   0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20
   Rx:9000

   Sha256(empty) =
   E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855

   ESK= HMAC-SHA256(0,PSK)
   ESK=
   23499E7EDF0FBE6BAA137DF0F23BECAEFA722AD19FC262855409DE8CD8B3C897

   DSK= HMAC-SHA256(ESK,0020 0d746c7331332064657269766564 20
   E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855 01)
   DSK=
   E8E7AC087158FC8440E41A12989F9194783764CD5FC36564028037F2C8206E96

   BSK = HMAC-SHA256(ESK,0020 10746c733133206578742062696e646572 20
   E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855 01)
   BSK=
   4351F8A53AA85AC394AB04C516464CAB96E9340C269632D09899537887EE651F

   FEK= HMAC-256(BSK, 0020 0E746C7331332066696E6973686564 00 01)

4.4 CETS: Client Early Traffic Secret

   Length = 2 + Messages-Length
   Hash-Length: the hash length (2 bytes)

   Tx: CLA INS=TLS13 P1=CETS P2=ESK P3=Length Hash-Length Messages-
   Length [Messages]
   Rx:[Client Early Traffic Secret] 9000

   4.4.1 Example

   Tx: CLA=00 INS=85 P1=00 P2=0B P3=03 0020 00
   Rx: 0738A2B6F6FAA2AF5CDD9B6F0F2B232F19B3256A5926EAC600B911F91E98D2D4
   9000

   Message= NULL = 0s
   [Client Early Traffic Secret] =
   HMAC-SHA256(ESK, 0020 11746c7331332063206520747261666669636 00 01)

4.5 EEMS: Early Exporter Master Secret

   Length = 2 + Messages-Length
   Hash-Length: the hash length (2 bytes)

   Tx: CLA INS=TLS13 P1=EEMS P2=ESK P3=Length Hash-Length Messages-
   Length [Messages]
   Rx: [Early Exporter Master Secret] 9000

  4.5.1 Example

   Tx: CLA=00 INS=85 P1=01 P2=0B P3=03 0020 00
   Rx: 9B7FC6A8F854C16A301DFC566859931DB5EE9A22793142A0C67159C445E7BEAB
   9000

   Message= NULL = 0s
   [Early Exporter Master Secret] =
   HMAC-SHA256(ESK, 0020 12746c7331332065207870206d6173746572 00 01)

4.6 HEDSK: HKDF-Extract from Derived Secret Key

   Tx: CLA INS=TLS13 P1=0 P2=HEDSK P3=Data-Length [Data]
   Rx: [HMAC(Data,DSK)] 9000

  4.6.1 Example

   Tx: CLA=00 INS=85 P1=00 P2=0E P3=01 00
   Rx: 7092C2117D67E6AEB5C5FDF5E6D9C70FBDC69B374E914C26AB08A122483D0E73

   DHE=NULL=0s
   HMAC-256(DSK,DHE)= HMAC-256(DSK,0s)

4.7 HBSK: HMAC from Binder Key Secret

   Tx: CLA INS=TLS13 P1=0 P2=HBSK P3=Data-Length [Data]
   Rx: [HMAC(FEK,data)] 9000

  4.7.1 Example

   Tx: CLA=00 INS=85 P1=00 P2=0C P3=01 00
   Rx: 3E015D850B89C2470D4C49D4BD8E7C76F2B74175DDD85F393569315DA15480A4

   Data=NULL=0s
   HMAC-256(FEK,Data)= HMAC-256(DSK,0s)

4.8 Signature Procedures

  4.8.1 Keys Generation

   Select Identity Module Application (AID= 010203040500)
   Tx: CLA=00 INS=A4 P1=04 P2=00 P3=06 01 02 03 04 05 00
   Rx: 9000

 Verify Administrator PIN (PIN= "00000000")
 Tx: CLA=00 INS=20 P1=00 P2=01 P3=08 30 30 30 30 30 30 30 30
 Rx: 9000

 Clear Key (P2=KeyIndex=0)
 Tx: CLA=00 INS=81 P1=00 P2=00 P3=00
 Rx: 9000

 Init Curve secp256r1 (P1 = idCurve, P2=KeyIndex)
 Tx: CLA=00 INS=89 P1=00 P2=00 P3=00
 Rx: 9000

 GenKey (P2=IndexKey)
 Tx: CLA=00 INS=82 P1=00 P2=00 P3=00
 Rx:9000

 Read PublicKey (P2=IndexKey)
 Tx: CLA=00 INS=84 P1=06 P2=00 P3=00
 Rx: 0041049E92726E24A548BB69ADA51103F265AA9B9F304E25971427D79EFAF471
     889CCC52FD8B05A729A400105C06AF99592535A4EDF338B5A37BB6089D3B11E7
     1B847B 9000

 Read PrivateKey (P2= IndexKey)
 Tx: CLA=00 INS=84 P1=07 P2=00 p3=00
 Rx: 00208E8793D5C399659D8A35B585534B5D9D0FAB37AD3FC7E8B43373C4BAD81E
     9000

4.8.2 Keys Setting

 Select Identity Module Application (AID= 010203040500)
 Tx: CLA=00 INS=A4 P1=04 P2=00 P3=06 01 02 03 04 05 00
 Rx: 9000

 Verify Administrator PIN (PIN= "00000000")
 Tx: CLA=00 INS=20 P1=00 P2=01 P3=08 30 30 30 30 30 30 30 30
 Rx: 9000

 Clear Key (P2=KeyIndex=0)
 Tx: CLA=00 INS=81 P1=00 P2=00 P3=00
 Rx: 9000

 Init Curve secp256r1 (P1 = idCurve, P2=KeyIndex)
 Tx: CLA=00 INS=89 P1=00 P2=00 P3=00
 Rx: 9000

 Set PrivateKey (P2=KeyIndex)
 Tx: CLA=00 INS=88 P1=07 P2=00 P3=20
 2e86bdd6d3b241ddbd00999f6a0ac1cb546d2bfb55744dca40f0268ac2bf7338
 Rx: 9000

 Set PublicKey (P2=KeyIndex)

```
 Tx: CLA=00 INS=88 P1=06 P2=00 P3=41
 045c8c90d0859dd96c722a589c4b62047ff01323cc74383e0e8eb80bea4ea45e55b8
 5499abd39d719885e874ed3f6327960d519ba25423c3fbdc14e6fd0cd5edee
 Rx: 9000
```

4.8.3 Signature

```
 Select Identity Module Application (AID= 010203040500)
 Tx: CLA=00 INS=A4 P1=04 P2=00 P3=06 01 02 03 04 05 00
 Rx: 9000
 Verify User PIN (PIN= "0000")
 CLA=00 INS=20 P1=00 P2=00 P3=04 30 30 30 30

 ECDSA secp256r1 Signature (P2=KeyIndex)
 Tx: CLA=00 INS=80 P1=00 P2=00 P3=20
 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
 Rx: 0047304502206BB1B02742C90B5FEAD3EF34F87B49D2A87F846F0368D0DBB3A
     0E9D9F3ABC450022100A0178CDE84FB9ACA4662ECC68638437D46EC27B69657
     8F8080E43ACCA4B35586
```

5. A simple Identity Module code for Javacard 3.04

```
package im;

import javacard.framework.*;
import javacard.security.* ;
import javacardx.crypto.*  ;

public class im extends Applet
{
final static byte  INS_SIGN            = (byte)  0x80 ;
final static byte  INS_CLEAR_KEYPAIR   = (byte)  0x81 ;
final static byte  INS_GEN_KEYPAIR     = (byte)  0x82 ;
final static byte  INS_GET_KEY_PARAM   = (byte)  0x84 ;
final static byte  INS_HMAC            = (byte)  0x85 ;
final static byte  INS_GET_STATUS      = (byte)  0x87 ;
final static byte  INS_SET_KEY_PARAM   = (byte)  0x88 ;
final static byte  INS_INIT_CURVE      = (byte)  0x89 ;
final static byte  INS_SELECT          = (byte)  0xA4 ;
public final static byte INS_VERIFY    = (byte)  0x20 ;
public final static byte INS_CHANGE_PIN = (byte)  0x24 ;

public final static short N_KEYS     = (short) 16;
public final static byte[] VERSION= {(byte)1,(byte)0};

KeyPair[] ECCkp      = null ;
Signature ECCsig     = null ;
MessageDigest sha256 = null ;

short status=0  ;
byte [] DB = null                               ;
public final static short DBSIZE = (short)320 ;

private static OwnerPIN UserPin=null;

private static final  byte[]   MyPin =
{(byte)0x30,(byte)0x30,(byte)0x30,(byte)0x30,
 (byte)0xFF,(byte)0xFF,(byte)0xFF,(byte)0xFF};

private static OwnerPIN AdminPin=null;

private static final byte[]  OpPin =
{(byte)0x30,(byte)0x30,(byte)0x30,(byte)0x30,
 (byte)0x30,(byte)0x30,(byte)0x30,(byte)0x30};

private final static short SW_VERIFICATION_FAILED  = (short)0x6300;
private final static short SW_PIN_VERIFICATION_REQUIRED  =
(short)0x6380;
final static  short SW_KPUB_DEFINED    = (short)0x6401;
final static  short SW_KPRIV_DEFINED   = (short)0x6402;
final static  short SW_KPRIV_UNDEFINED = (short)0x6403;
```

```
final static   short SW_GENKEY_ERROR    = (short)0x6D10;
final static   short SW_SIGN_ERROR      = (short)0x6D20;
final static   short SW_DUMP_KEYS_PAIR  = (short)0x6D30;
final static   short SW_SET_KEY_PARAM   = (short)0x6D40;

private final static byte [] ParamA1 =
{(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x01,(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x00,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,
(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,
(byte)0xff,(byte)0xfc};

private final static byte [] ParamB1 =
{(byte)0x5a,(byte)0xc6,(byte)0x35,(byte)0xd8,(byte)0xaa,(byte)0x3a,
(byte)0x93,(byte)0xe7,(byte)0xb3,(byte)0xeb,(byte)0xbd,(byte)0x55,
(byte)0x76,(byte)0x98,(byte)0x86,(byte)0xbc,(byte)0x65,(byte)0x1d,
(byte)0x06,(byte)0xb0,(byte)0xcc,(byte)0x53,(byte)0xb0,(byte)0xf6,
(byte)0x3b,(byte)0xce,(byte)0x3c,(byte)0x3e,(byte)0x27,(byte)0xd2,
(byte)0x60,(byte)0x4b};

private final static byte [] ParamField1=
{(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x01,(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x00,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,
(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,
(byte)0xff,(byte)0xff};

private final static byte [] ParamG1=
{(byte)0x04,(byte)0x6b,(byte)0x17,(byte)0xd1,(byte)0xf2,(byte)0xe1,
(byte)0x2c,(byte)0x42,(byte)0x47,(byte)0xf8,(byte)0xbc,(byte)0xe6,
(byte)0xe5,(byte)0x63,(byte)0xa4,(byte)0x40,(byte)0xf2,(byte)0x77,
(byte)0x03,(byte)0x7d,(byte)0x81,(byte)0x2d,(byte)0xeb,(byte)0x33,
(byte)0xa0,(byte)0xf4,(byte)0xa1,(byte)0x39,(byte)0x45,(byte)0xd8,
(byte)0x98,(byte)0xc2,(byte)0x96,(byte)0x4f,(byte)0xe3,(byte)0x42,
(byte)0xe2,(byte)0xfe,(byte)0x1a,(byte)0x7f,(byte)0x9b,(byte)0x8e,
(byte)0xe7,(byte)0xeb,(byte)0x4a,(byte)0x7c,(byte)0x0f,(byte)0x9e,
(byte)0x16,(byte)0x2b,(byte)0xce,(byte)0x33,(byte)0x57,(byte)0x6b,
(byte)0x31,(byte)0x5e,(byte)0xce,(byte)0xcb,(byte)0xb6,(byte)0x40,
(byte)0x68,(byte)0x37,(byte)0xbf,(byte)0x51,(byte)0xf5};

private final static short ParamK1 = (short) 0x0001;

private final static byte [] ParamR1=
{(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0x00,(byte)0x00,
(byte)0x00,(byte)0x00,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,
(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xbc,(byte)0xe6,
(byte)0xfa,(byte)0xad,(byte)0xa7,(byte)0x17,(byte)0x9e,(byte)0x84,
(byte)0xf3,(byte)0xb9,(byte)0xca,(byte)0xc2,(byte)0xfc,(byte)0x63,
(byte)0x25,(byte)0x51};
```

```
private byte []   ESK = new byte[32]; // Early Secret Key
private byte []   HSK = new byte[32]; // Handshake Secret Key
private byte []  eBSK = new byte[32]; // Binder Secret Key
private byte []  rBSK = new byte[32]; // Binder Secret Key
private byte [] feBSK = new byte[32]; // Finished Binder Secret Key
private byte [] frBSK = new byte[32]; // Finished Binder Secret Key

private final static byte  EXTRACT_EARLY  =  (byte)0x0A;
private final static byte  EXPAND_EARLY   =  (byte)0x0B;
private final static byte  HMAC_EBSK      =  (byte)0x0C;
private final static byte  HMAC_RBSK      =  (byte)0x0D;
private final static byte  EXTRACT_HANDSHAKE =  (byte)0x0E;

private byte [] derived =
{(byte)0x00,(byte)32,(byte)13,(byte)'t',(byte)'l',(byte)'s',
(byte)'1',(byte)'3',(byte)' ',(byte)'d',(byte)'e',(byte)'r',
(byte)'i',(byte)'v',(byte)'e',(byte)'d',
(byte)0x20,(byte)0xE3,(byte)0xB0,(byte)0xC4,(byte)0x42,(byte)0x98,
(byte)0xFC,(byte)0x1C,(byte)0x14,(byte)0x9A,(byte)0xFB,(byte)0xF4,
(byte)0xC8,(byte)0x99,(byte)0x6F,(byte)0xB9,(byte)0x24,(byte)0x27,
(byte)0xAE,(byte)0x41,(byte)0xE4,(byte)0x64,(byte)0x9B,(byte)0x93,
(byte)0x4C,(byte)0xA4,(byte)0x95,(byte)0x99,(byte)0x1B,(byte)0x78,
(byte)0x52,(byte)0xB8,(byte)0x55,(byte)1};

private byte [] ext_binder =
{(byte)0x00,(byte)32,(byte)16,(byte)'t',(byte)'l',(byte)'s',
(byte)'1',(byte)'3',(byte)' ',(byte)'e',(byte)'x',(byte)'t',
(byte)' ',(byte)'b',(byte)'i',(byte)'n',(byte)'d',(byte)'e',
(byte)'r',(byte)0x20,(byte)0xE3,(byte)0xB0,(byte)0xC4,(byte)0x42,
(byte)0x98,(byte)0xFC,(byte)0x1C,(byte)0x14,(byte)0x9A,(byte)0xFB,
(byte)0xF4,(byte)0xC8,(byte)0x99,(byte)0x6F,(byte)0xB9,(byte)0x24,
(byte)0x27,(byte)0xAE,(byte)0x41,(byte)0xE4,(byte)0x64,(byte)0x9B,
(byte)0x93,(byte)0x4C,(byte)0xA4,(byte)0x95,(byte)0x99,(byte)0x1B,
(byte)0x78,(byte)0x52,(byte)0xB8,(byte)0x55,(byte)0x01};

private byte [] res_binder   =
{(byte)0x00,(byte)32,(byte)16,(byte)'t',(byte)'l',(byte)'s',
(byte)'1',(byte)'3',(byte)' ',(byte)'r',(byte)'e',(byte)'s',
(byte)' ',(byte)'b',(byte)'i',(byte)'n',(byte)'d',(byte)'e',
(byte)'r',(byte)0x00,(byte)0x01};

private byte [] c_e_traffic  =
{(byte)17,(byte)'t',(byte)'l',(byte)'s',(byte)'1',(byte)'3',
(byte)' ',(byte)'c',(byte)' ',(byte)'e',(byte)' ',(byte)'t',
(byte)'r',(byte)'a',(byte)'f',(byte)'f',(byte)'i',(byte)'c'};
```

```
private byte [] c_exp_master =
{(byte)18,(byte)'t',(byte)'l',(byte)'s',(byte)'1',(byte)'3',
(byte)' ',(byte)'e',(byte)' ',(byte)'e',(byte)'x',(byte)'p',
(byte)' ',(byte)'m',(byte)'a',(byte)'s',(byte)'t',(byte)'e',
(byte)'r'};

private byte [] finished =
{(byte)0x00,(byte)32,(byte)14,(byte)'t',(byte)'l',(byte)'s',
(byte)'1',(byte)'3',(byte)' ',(byte)'f',(byte)'i',(byte)'n',
(byte)'i',(byte)'s',  (byte)'h',(byte)'e',(byte)'d',(byte)0,
(byte)1};


public void process(APDU apdu) throws ISOException
{ short adr=0,len=0,index=0,readCount=0;

byte[] buffer = apdu.getBuffer() ;

byte cla = buffer[ISO7816.OFFSET_CLA];
byte ins = buffer[ISO7816.OFFSET_INS];
byte P1  = buffer[ISO7816.OFFSET_P1] ;
byte P2  = buffer[ISO7816.OFFSET_P2] ;
byte P3  = buffer[ISO7816.OFFSET_LC] ;

adr = Util.makeShort(P1,P2)        ;
len = Util.makeShort((byte)0,P3)   ;

switch (ins)
{

case INS_SELECT:
readCount = apdu.setIncomingAndReceive();
return;

case INS_GET_STATUS:
Util.arrayCopyNonAtomic(VERSION,(short)0,buffer,(short)0,(short)VERS
ION.length);
Util.setShort(buffer,(short)VERSION.length,status);
apdu.setOutgoingAndSend((short)0,(short)(2+VERSION.length));
break;

case INS_VERIFY:
readCount = apdu.setIncomingAndReceive();
if (P2 == (byte)1)
{  if (readCount != (short)8)
   ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
   verify(AdminPin,buffer) ;
   if(AdminPin.isValidated()) UserPin.resetAndUnblock();
```

```
else if (P2 == (byte)0xFF)
{  if (readCount != (short)8)
   ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
   verify(AdminPin,buffer) ;
   if(AdminPin.isValidated())
   {  UserPin.resetAndUnblock();
      UserPin.update(MyPin,(short)0,(byte)8) ;
   }
}
}
else if (P2 == (byte)0)
{  if (readCount > (short)8)
   ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
   verify(UserPin,buffer);
}
else
ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
break;


case INS_CHANGE_PIN:
readCount = apdu.setIncomingAndReceive() ;
if (readCount != (short)16)
ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
buffer[4]=(byte)8;
if (P2 == (byte)1)
{  verify(AdminPin,buffer) ;
   AdminPin.update(buffer,(short)13,(byte)8);
}
else if  (P2 == (byte)0)
{  verify(UserPin,buffer)  ;
   UserPin.update(buffer,(short)13,(byte)8);
}
else
ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
break;


case INS_HMAC:


readCount = apdu.setIncomingAndReceive();
len = Util.makeShort((byte)0,buffer[(short)4]);

if (len != readCount)
ISOException.throwIt(ISO7816.SW_CORRECT_LENGTH_00);

else if ( (!AdminPin.isValidated()) && (!UserPin.isValidated()) )
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
```

```
if (P2 == (byte)2) // Compute HMAC
{  len = Util.makeShort((byte)0,buffer[(short)5]) ;
   hmac(buffer, (short)6, len, buffer, (short)(7+len),
        Util.makeShort((byte)0, buffer[(short)(6+len)]),
        sha256, buffer,(short)0,true);
   apdu.setOutgoingAndSend((short)0,(short)sha256.getLength());
}

else if (P2 == EXTRACT_EARLY)
{ len = Util.makeShort((byte)0,buffer[(short)5]); //HMAC: key-length
  hmac(buffer,(short)6,len,buffer,(short)(7+len),
        Util.makeShort((byte)0,buffer[(short)(6+len)]),
        sha256, buffer,(short)0,true);
   Util.arrayCopyNonAtomic(buffer,(short)0,ESK,(short)0,
                           (short)ESK.length);
   Util.arrayCopyNonAtomic(buffer,(short)0,buffer,(short)32,
                           (short)32);


   hmac(ESK,(short)0,(short)ESK.length,
        derived,(short)0,(short)derived.length,
        sha256,
        buffer,(short)0,true);
   Util.arrayCopyNonAtomic(buffer,(short)0,HSK,(short)0,
                           (short)HSK.length);
   Util.arrayCopyNonAtomic(buffer,(short)0,buffer,(short)64,
                           (short)32);


   hmac(ESK,(short)0,(short)ESK.length,
        ext_binder,(short)0,(short)ext_binder.length,
        sha256,
        buffer,(short)0,true);
    Util.arrayCopyNonAtomic(buffer,(short)0,eBSK,(short)0,
                           (short)eBSK.length);
    Util.arrayCopyNonAtomic(buffer,(short)0,buffer,(short)96,
                           (short)32);


    hmac(ESK,(short)0,(short)ESK.length,
         res_binder,(short)0,(short)res_binder.length,
         sha256,
         buffer,(short)0,true);
    Util.arrayCopyNonAtomic(buffer,(short)0,rBSK,(short)0,
                           (short)rBSK.length);
    Util.arrayCopyNonAtomic(buffer,(short)0,buffer,(short)128,
                           (short)32);
```

```
        hmac(eBSK,(short)0,(short)eBSK.length,
             finished,(short)0,(short)finished.length,
             sha256,
             buffer,(short)0,true);
        Util.arrayCopyNonAtomic(buffer,(short)0,feBSK,(short)0,
                                (short)feBSK.length);
        Util.arrayCopyNonAtomic(buffer,(short)0,buffer,(short)160,
                                (short)32);


        hmac(rBSK,(short)0,(short)rBSK.length,
                   finished,(short)0,(short)finished.length,
                   sha256,
                   buffer,(short)0,true);
        Util.arrayCopyNonAtomic(buffer,(short)0,frBSK,(short)0,
                                (short)frBSK.length);
        Util.arrayCopyNonAtomic(buffer,(short)0,buffer,(short)192,
                                (short)32);

        If (P1==(byte)0xFF)
        apdu.setOutgoingAndSend((short)32,(short)192);
        return ;
}

else if (P2 == EXPAND_EARLY)
{  len = Util.makeShort((byte)0,buffer[(short)7]);  // data length
   if (P1 == (byte)0)
   {
     Util.arrayCopyNonAtomic(buffer,(short)5,buffer,(short)0,
                                      (short)2);
     Util.arrayCopyNonAtomic(buffer,(short)7,
                                 buffer,(short)(2+ c_e_traffic.length),
                                 (short)(readCount-2));
     Util.arrayCopyNonAtomic(c_e_traffic,(short)0,buffer,(short)2,
                                 (short)c_e_traffic.length);
     buffer[(short)(readCount + c_e_traffic.length)] = (byte)0x01;
     hmac(ESK,(short)0,(short)ESK.length,
          buffer,(short)0,(short)(readCount+c_e_traffic.length+1),
          sha256,
          buffer,(short)0,true);
          apdu.setOutgoingAndSend((short)0,(short)32);
          return;
     }
     else if (P1 == (byte)1)
     {
      Util.arrayCopyNonAtomic(buffer,(short)5,buffer,(short)0,
                                 (short)2);
      Util.arrayCopyNonAtomic(buffer,(short)7,buffer,
                                 short)(2+ c_exp_master.length),
                                 (short)(readCount-2));
```

```
       Util.arrayCopyNonAtomic(c_exp_master,(short)0,buffer,(short)2,
                              (short)c_exp_master.length);
       buffer[(short)(readCount + c_exp_master.length)] = (byte)0x01;
       hmac(ESK,(short)0,(short)ESK.length,
            buffer,(short)0,(short)(readCount+c_exp_master.length+1),
            sha256,
            buffer,(short)0,true);
       apdu.setOutgoingAndSend((short)0,(short)32);
        return;
      }
      else
      ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);

else if ( P2 == HMAC_RBSK)
{   hmac(frBSK,(short)0,(short)rBSK.length,
         buffer,(short)5,readCount,
         sha256,
         buffer,(short)0,true);
         apdu.setOutgoingAndSend((short)0,(short)sha256.getLength());
}

else if (P2 == HMAC_EBSK)
{   hmac(feBSK,(short)0,(short)eBSK.length,
         buffer,(short)5,readCount,
         sha256,
         buffer,(short)0,true);
         apdu.setOutgoingAndSend((short)0,(short)sha256.getLength());
}

else if (P2 == EXTRACT_HANDSHAKE )
{   hmac(HSK,(short)0,(short)HSK.length,
         buffer,(short)5,readCount,
         sha256,
         buffer,(short)0,true);
         apdu.setOutgoingAndSend((short)0,(short)32);
}

else
ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
break;


case INS_SIGN:
readCount = apdu.setIncomingAndReceive();
if ( (!AdminPin.isValidated()) && (!UserPin.isValidated()) )
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

index= Util.makeShort((byte)0,P2);
if ( (index <0) || (index >= N_KEYS))
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if (!ECCkp[index].getPublic().isInitialized())
```

```
ISOException.throwIt(SW_KPUB_DEFINED);
if (!ECCkp[index].getPrivate().isInitialized())
ISOException.throwIt(SW_KPRIV_DEFINED);

switch (P1)
{
 case (byte)0: // RAW 256 bits
 case (byte)33:// ALG_ECDSA_SHA_256
                len= EccSign(ECCkp[index],buffer,P1) ;
                apdu.setOutgoingAndSend((short)0,len);
                break;
 default:
        ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
        break;
}
break;


case INS_CLEAR_KEYPAIR:

if ( !AdminPin.isValidated())
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
index= Util.makeShort((byte)0,P2);
if ( (index <0) || (index >= N_KEYS))
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if (ECCkp[index].getPublic().isInitialized())
ECCkp[index].getPublic().clearKey();
if (ECCkp[index].getPrivate().isInitialized())
ECCkp[index].getPrivate().clearKey();
break;

case INS_GEN_KEYPAIR: // Generate KeyPair

if ( !AdminPin.isValidated())
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
index= Util.makeShort((byte)0,P2);
if ( (index <0) || (index >= N_KEYS))
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if (ECCkp[index].getPublic().isInitialized())
ISOException.throwIt(SW_KPUB_DEFINED);
if (ECCkp[index].getPrivate().isInitialized())
ISOException.throwIt(SW_KPRIV_DEFINED);
len=this.GenECCkp(ECCkp[index]);
break;

case INS_GET_KEY_PARAM:

if ( (!AdminPin.isValidated()) && (!UserPin.isValidated())  )
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
index= Util.makeShort((byte)0,P2);
if ( (index <0) || (index >= N_KEYS))
```

```
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if ( (P1 == (byte)7) && !AdminPin.isValidated())
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
if ( (P1 == (byte)6) && !ECCkp[index].getPublic().isInitialized())
ISOException.throwIt(SW_KPUB_DEFINED);
if ( (P1 == (byte)7) && !ECCkp[index].getPrivate().isInitialized())
ISOException.throwIt(SW_KPRIV_DEFINED)
try
{  switch (P1)
   {  case 0:
        len= ((ECPublicKey)ECCkp[index].getPublic())
             .getA(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;

        case 1:
        len= ((ECPublicKey) ECCkp[index].getPublic())
             .getB(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;

        case 2:
        len= ((ECPublicKey) ECCkp[index].getPublic())
             .getField(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;

        case 3:
        len= ((ECPublicKey)ECCkp[index].getPublic())
              .getG(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;

        case 4:
        len= ((ECPublicKey) ECCkp[index].getPublic()).getK();
        Util.setShort(buffer,(short)2,len);
        Util.setShort(buffer,(short)0,(short)2);
        apdu.setOutgoingAndSend((short)0,(short)4);
        break;

        case 5:
        len= ((ECPublicKey) ECCkp[index].getPublic())
              .getR(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;
```

```
        case (byte)6:
        len= ((ECPublicKey) ECCkp[index].getPublic())
              .getW(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;

        case (byte)7:
        len= ((ECPrivateKey)ECCkp[index].getPrivate())
              .getS(buffer,(short)(2));
        Util.setShort(buffer,(short)0,len);
        apdu.setOutgoingAndSend((short)0,(short)(len+2));
        break;

        default:
        ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
        break;
    }
}
catch (CryptoException e)
{ISOException.throwIt(SW_DUMP_KEYS_PAIR);
 break;
}

break;


case INS_SET_KEY_PARAM:

readCount = apdu.setIncomingAndReceive();

if ( !AdminPin.isValidated())
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
index= Util.makeShort((byte)0,P2);
if ( (index <0) || (index >= N_KEYS))
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if ( (P1 == (byte)6) && ECCkp[index].getPublic().isInitialized())
ISOException.throwIt(SW_KPUB_DEFINED);
if ( (P1 == (byte)7) && ECCkp[index].getPrivate().isInitialized())
ISOException.throwIt(SW_KPRIV_DEFINED);

try
{  switch (P1)
    { case (byte)0:
      ((ECPublicKey)ECCkp[index].getPublic())
        .setA(buffer,(short)5,len);
      ((ECPrivateKey)ECCkp[index].getPrivate())
       .setA(buffer,(short)5,len);
      break;
```

```
      case (byte)1:
      ((ECPublicKey)ECCkp[index].getPublic())
       .setB(buffer,(short)5,len);
      ((ECPrivateKey)ECCkp[index].getPrivate())
       .setB(buffer,(short)5,len);
      break;

      case (byte)2:
      ((ECPublicKey)ECCkp[index].getPublic())
      .setFieldFP(buffer,(short)5,len) ;
      ((ECPrivateKey)ECCkp[index].getPrivate())
      .setFieldFP(buffer,(short)5,len);
      break;

      case (byte)3:
      ((ECPublicKey)ECCkp[index].getPublic())
       .setG(buffer,(short)5,len) ;
      ((ECPrivateKey)ECCkp[index].getPrivate())
       .setG(buffer,(short)5,len);
      break;

      case (byte)4:
      ((ECPublicKey)ECCkp[index].getPublic())
       .setK(Util.makeShort(buffer[5],buffer[6])) ;
      ((ECPrivateKey)ECCkp[index].getPrivate())
       .setK(Util.makeShort(buffer[5],buffer[6]));
      break;

      case (byte)5:
      ((ECPublicKey)ECCkp[index].getPublic())
       .setR(buffer,(short)5,len);
      ((ECPrivateKey)ECCkp[index].getPrivate())
       .setR(buffer,(short)5,len);
      break;

      case (byte)6:
      ((ECPublicKey)ECCkp[index].getPublic())
       .setW(buffer,(short)5,len) ;
      break;

      case (byte)7:
      ((ECPrivateKey)ECCkp[index].getPrivate())
       .setS(buffer,(short)5,len);
      break;

      default:
      ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
      break;
     }
 }
```

```
catch (CryptoException e)
{ISOException.throwIt(SW_SET_KEY_PARAM);
 break;
}

break;


case INS_INIT_CURVE:

if ( !AdminPin.isValidated())
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
index= Util.makeShort((byte)0,P2);
if ( (index <0) || (index >= N_KEYS))
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if ( (P1 == (byte)6) && ECCkp[index].getPublic().isInitialized() )
ISOException.throwIt(SW_KPUB_DEFINED);
if ((P1 == (byte)7) && ECCkp[index].getPrivate().isInitialized())
ISOException.throwIt(SW_KPRIV_DEFINED);

switch((byte)P1)
{  case (byte)0:
   case (byte)1:
   (ECPublicKey)ECCkp[index].getPublic())
    .setA(ParamA1,(short)0,(short)ParamA1.length) ;
   ((ECPrivateKey)ECCkp[index].getPrivate())
    .setA(ParamA1,(short)0,(short)ParamA1.length);
   ((ECPublicKey)ECCkp[index].getPublic())
    .setB(ParamB1,(short)0,(short)ParamB1.length) ;
   ((ECPrivateKey)ECCkp[index].getPrivate())
    .setB(ParamB1,(short)0,(short)ParamB1.length);
    ((ECPublicKey)ECCkp[index].getPublic())
    .setFieldFP(ParamField1,(short)0,(short)ParamField1.length);
   ((ECPrivateKey)ECCkp[index].getPrivate())
    .setFieldFP(ParamField1,(short)0,(short)ParamField1.length);
   ((ECPublicKey)ECCkp[index].getPublic())
    .setG(ParamG1,(short)0,(short)ParamG1.length) ;
   ((ECPrivateKey)ECCkp[index].getPrivate())
    .setG(ParamG1,(short)0,(short)ParamG1.length);
   ((ECPublicKey)ECCkp[index].getPublic())
    .setK(ParamK1) ;
   ((ECPrivateKey)ECCkp[index].getPrivate())
    .setK(ParamK1);
   ((ECPublicKey)ECCkp[index].getPublic())
    .setR(ParamR1,(short)0,(short)ParamR1.length) ;
   ((ECPrivateKey)ECCkp[index].getPrivate())
    .setR(ParamR1,(short)0,(short)ParamR1.length);
    break;
```

```
   default:
    ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
    break;
   }
break;

default:
ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
}

}

public short EccSign(KeyPair ECCkeyPair, byte [] buf, byte mode)
{ short len,sLen=(short)0;
  len= Util.makeShort((byte)0,buf[4]);
  Util.arrayCopy(buf,(short)5,buf,(short)2,len) // Sign
  try
  {  if (mode == (byte)0)// default
       { ECCsig.init(ECCkeyPair.getPrivate(),Signature.MODE_SIGN);
         sLen = ECCsig.signPreComputedHash(buf,(short)2,len buf,
              (short)(2+len));
       }
     else
       { ECCsig.init(ECCkeyPair.getPrivate(),Signature.MODE_SIGN);
         sLen = ECCsig.sign(buf, (short)2, len, buf, (short)(2+len));
       }
   }
  catch (CryptoException e)
  {ISOException.throwIt(SW_SIGN_ERROR);
   return (short)0;
   }

Util.arrayCopy(buf,(short)(2+len),buf,(short)2,sLen);
Util.setShort(buf,(short)0,sLen);
return(short)(sLen+2);
}

public short GenECCkp(KeyPair ECCkeyPair)
{ short len;
  try
  { ECCkeyPair.genKeyPair(); }
  catch (CryptoException e)
  { ISOException.throwIt(SW_GENKEY_ERROR);
    return (short)0;
   }
return 0;
}
```

```
public void verify(OwnerPIN pin,byte [] buffer) throws ISOException
  {short i,x;
   x = Util.makeShort((byte)0,buffer[4]);
   for(i=x;i<(short)8;i=(short)(i+1))
   buffer[(short)(5+i)]=(byte)0xFF;
   if ( pin.check(buffer, (short)5,(byte)8) == false )
   ISOException.throwIt((short)((short)SW_VERIFICATION_FAILED |
                                (short)pin.getTriesRemaining()));
  }

public static final short DB_off = (short)0 ;

public void hmac
( byte []  k,short k_off, short lk, // Secret key
  byte []  d,short d_off,short ld,  // data
  MessageDigest md,
  byte out[], short out_off, boolean init)
{
short i,DIGESTSIZE, DIGESTSIZE2=(short)64,BLOCKSIZE=(short)128;
DIGESTSIZE=(short)md.getLength();
if (md.getAlgorithm() == md.ALG_SHA_512)
{ DIGESTSIZE2= (short)64; BLOCKSIZE  = (short)128; }
else if (md.getAlgorithm() == md.ALG_SHA_256)
{ DIGESTSIZE2= (short)32; BLOCKSIZE = (short)64;}

if (init)
{ if (lk > (short)BLOCKSIZE )
  { md.reset();
    md.doFinal(k,k_off,lk,k,k_off);
    lk = DIGESTSIZE ;
  }
  for (i = 0 ; i < lk ; i=(short)(i+1))
  DB[(short)(i+DB_off+BLOCKSIZE+DIGESTSIZE2)] =
  (byte)(k[(short)(i+k_off)] ^ (byte)0x36) ;
  Util.arrayFillNonAtomic (
  DB,(short)(BLOCKSIZE+DIGESTSIZE2+lk+DB_off),
  (short)(BLOCKSIZE-lk),(byte)0x36);
  for (i = 0 ; i < lk ; i=(short)(i+1))
  DB[(short)(i+DB_off)] = (byte)(k[(short)(i+k_off)] ^ (byte)0x5C);
  Util.arrayFillNonAtomic(DB,(short)(lk+DB_off),
                          (short)(BLOCKSIZE-lk),(byte)0x5C);
}
  md.reset();
  md.update(DB,(short)(DB_off+BLOCKSIZE+DIGESTSIZE2),BLOCKSIZE);
  md.doFinal(d, d_off,ld,DB,(short)(DB_off+BLOCKSIZE));
  md.reset();
  md.doFinal(DB,DB_off,(short)(DIGESTSIZE+BLOCKSIZE),out,out_off);
}
```

```
protected im(byte[] bArray,short bOffset,byte bLength)
{ init();
  register();
}


public void init()
{ short i=0;
  status = (short)0;
  ECCkp  = new KeyPair[N_KEYS];
  UserPin   = new OwnerPIN((byte)3,(byte)8);  // 3 tries, 4=Max Size
  AdminPin  = new OwnerPIN((byte)10,(byte)8); // 10 tries 8=Max Size
  UserPin.update(MyPin,(short)0,(byte)8) ;
  AdminPin.update(OpPin,(short)0,(byte)8);
  for(i=0;i<N_KEYS;i++)
  {
    try{
        ECCkp[i] = new
        KeyPair(KeyPair.ALG_EC_FP,KeyBuilder.LENGTH_EC_FP_256);
        status =(short)(status + (short)1);
       }
    catch (CryptoException e){}
  }
  try {
      ECCsig =
      Signature.getInstance(Signature.ALG_ECDSA_SHA_256, false);
      status =(short)(status | (short)0x0100);
     }
  catch (CryptoException e){}
  try {
      sha256 =
      MessageDigest.getInstance(MessageDigest.ALG_SHA_256, false);
      status =(short)(status | (short)0x2000);
     }
  catch (CryptoException e){}
  DB = JCSystem.makeTransientByteArray(DBSIZE,
                                    JCSystem.CLEAR_ON_DESELECT);
}

public static void install(byte[] bArray, short bOffset,
                           byte bLength )
{ new im(bArray,bOffset,bLength);}


public boolean select()
{ if (UserPin.isValidated()) UserPin.reset();
  if (AdminPin.isValidated()) AdminPin.reset();
  return true;
}
```

6 IANA Considerations

   TODO

7 Security Considerations

   TODO

8 References

8.1 Normative References

   [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol
   Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018,
   https://www.rfc-editor.org/info/rfc8446.

   [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-
   Expand Key Derivation Function (HKDF)", RFC 5869, DOI
   10.17487/RFC5869, May 2010,
   <https://www.rfc-editor.org/info/rfc5869>.

   [ISO7816] ISO 7816, "Cards Identification - Integrated Circuit Cards
   with Contacts", The International Organization for Standardization
   (ISO).

8.2 Informative References

8 Authors' Addresses

   Pascal Urien
   Telecom Paris
   19 place Marguerite Perey
   91120 Palaiseau   Phone: NA
   France   Email: Pascal.Urien@telecom-paris.fr

                   Secure Element for TLS Version 1.3
                        draft-urien-tls-se-01.txt

Abstract

   This draft presents ISO7816 interface for TLS1.3 stack running in
   secure element. It presents supported cipher suites and key exchange
   modes, and describes embedded software architecture. TLS 1.3 is the
   de facto security stack for emerging Internet of Things (IoT)
   devices. Some of them are constraint nodes, with limited computing
   resources. Furthermore cheap System on Chip (SoC) components usually
   provide tamper resistant features, so private or pre shared keys are
   exposed to hacking. According to the technology state of art, some
   ISO7816 secure elements are able to process TLS 1.3, but with a
   limited set of cipher suites. There are two benefits for TLS-SE;
   first fully tamper resistant processing of TLS protocol, which
   increases the security level insurance; second embedded software
   component ready for use, which relieves the software of the burden
   of cryptographic libraries and associated attacks. TLS-SE devices
   may also embed standalone applications, which are accessed via
   internet node, using a routing procedure based on SNI extension.

Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119.

Status of this Memo

Copyright Notice

Table of Contents

1 Overview

   This draft presents ISO7816 interface for TLS1.3 stack running in
   secure element (see Figure 1), it presents supported cipher suites
   and key exchange modes, and describes embedded software
   architecture. TLS 1.3 [RFC8446] is the de facto security stack for
   emerging Internet of Things (IoT) devices. Some of them are
   constraint nodes, with limited computing resources. Furthermore
   cheap System on Chip (SOC) components don't usually provide tamper
   resistant features, so private or pre shared keys are exposed to
   hacking. The identity module (im) detailed in [IM] protects identity
   credentials. The TLS identity module [IM] MAY be based on secure
   element [ISO7816]. According to the technology state of art, some
   secure elements are able to process TLS 1.3, but with a limited set
   of cipher suites. There are two benefits for TLS-SE; first fully
   tamper resistant processing of TLS protocol, which increases the
   security level insurance; second embedded software component ready
   for use, which relieves the software of the burden of cryptographic
   libraries and associated attacks.
   Multiple TLS-SE devices, embedding standalone applications, can be
   hosted by an internet node. In this case SNI extension [RFC6066] MAY
   be used in order to select the right secure element (see Figure 2).

```
        +----------+ recv +-----------+ RECV +----------+
        |    IP    + ---> |  TCP/IP   | ---> |  TLS 1.3 |
        |  Network +------+ Constraint +------+ Secure   |
        |          | <--- |   Node    | <--- | Element  |
        +----------+ send +-----------+ SEND +----------+
                     |                  |
              Network Interface    ISO7816 interface
```
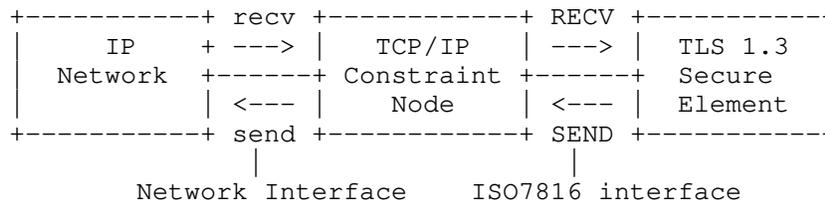
Figure 1. TLS 1.3 Secure Element (TLS-SE)

```
                                      +----------+
                                      |  TLS-SE  |
                          TLS-SE Name +---+ Secure  |
                                      |  | Element |
                                      |  +----------+
                 SNI= TLS-SE Name     |
   +----------+     +-----------+     |  +----------+
   |    IP    +     |  TCP/IP   |     |  |  TLS-SE  |
   |  Network +-----+   Node    +--+---+ Secure   |
   |          |     |SN Extension|  |  | Element |
   +----------+     +-----------+  |  +----------+
                                   |
                                   |  +----------+
                                   |  |  TLS-SE  |
                          TLS-SE Name +---+ Secure  |
                                      |  | Element |
                                      |  +----------+
```
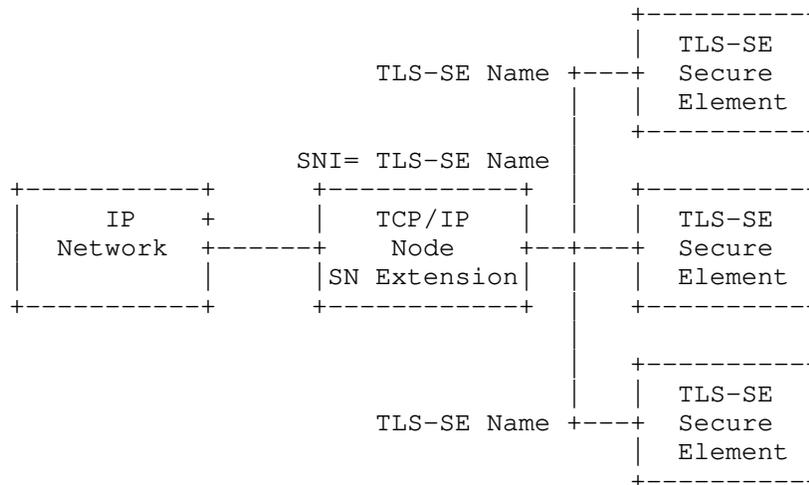
Figure 2. Routing procedure based on SNI for TLS-SE devices

2 About Secure Elements

   Secure elements are defined according to [ISO7816] standards. They
   support hash functions (sha256, sha384, sha512) and associated HMAC
   procedures. They also provide signatures and DH procedures in Z/pZ*
   groups, or elliptic curves (for example secp256r1). Open software
   can be released thanks to JavaCard (JC) standards, such as JC3.04,
   or JC3.05. Most of secure elements use 8 bits Micro Controller Unit
   (MCU) and embedded cryptographic accelerator. Non volatile memory
   size is up to 100KB, and RAM size is up to 10KB.

   Below is an illustration of binary encoding rules for secure element
   according to the T=0 ISO7816 protocol.
   An ISO7816 request is a set of bytes comprising a five byte header
   and an optional payload (up to 255 bytes)
   The header comprises the following five bytes:
   - CLA, Class
   - INS, Instruction code
   - P1, P1 byte
   - P2, P2 byte
   - P3, length of the optional payload, or number of expected bytes

   The response comprises an optional payload (up to 256 bytes) and a
   two bytes status word (SW1, SW2), SW1=90, SW2=00 (SW=9000) meaning
   successful operation.

   The ISO7816 defines two main classes for data exchange (called
   transport protocol), T=0, and T=1.
   The T=0 transport protocol is a byte stream; a payload can be
   included in request or response, but not in both.
   The T=1 transport protocol is a frame stream; payload can be
   included both in request and response.

3 Software components for TLS-SE

```
                            +--------+
                            | Crypto +----------------+
        RECV +----------+   |  Lib   |                |
        ---> |   Data   |   +--------+   +---------+----------+
        <--- | Exchange |      +---------+    TLS Lib         |
        SEND +----+-----+      |         | Make  Record Packet |
             |            +-----+-----+  | Check Record Packet |
             |            | TLS State |  +--------------------+
           +--------+  Machine  |
                        +----------+
```
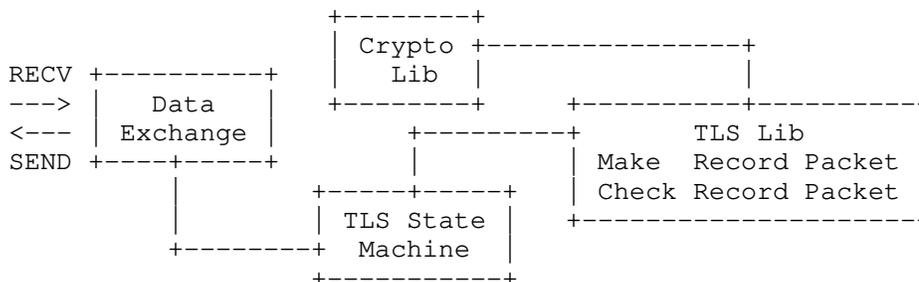
   Figure 3. Software Components for TLS-SE

3.1 Cryptographic resources

   Many secure elements support hash functions sha256, sha384 and
   sha512 used by TLS1.3. Associated HMAC, HKDF-Extract and Derive-
   Secret, MUST be implemented by a dedicated cryptographic library.

   Many secure elements support the secp256r1 elliptic curve. Diffie-
   Hellman (DH)calculation are performed according to [IEEE1363] using
   the ECKAS-DH1 scheme with the identity map as the key derivation
   function, (KDF), so that the shared secret is the x-coordinate of
   the ECDH shared secret elliptic curve point represented as an octet
   string. ECDSA signature is also available for 256,384 and 512 hash
   size.

   AES-128 is usually implemented, by not AES-CCM. So this AEAD
   algorithm SHOULD be implemented by a dedicated cryptographic
   library.

   In summary, according to the state of art TLS-SE supports the
   secp256r1 EC group, associated ECDSA signature computing and
   checking, and EC-DHE key establishment. It also implements the AES-
   128-CCM-SHA256 cipher suite.

3.2 Data exchange

   TLS record layer packets are received and sent from/to TCP/IP
   network thanks to well known socket procedures. TLS-SE processes
   these packets according to a dedicated state machine.

    3.2.1 Receiving Record Packet

   Dedicated ISO7816 requests (named RECV) push incoming record
   messages in secure element. A fragmentation mechanism splits the
   record packet in one a several ISO7816 requests, whose payload size
   is less than 255 bytes. A 2 bits fragmentation-flag field indicates
   the fragment status; bit F-First notifies the first fragment, and
   bit F-Last notifies the last fragment.

   The ISO7816 RECV request COULD be encoded as
   CLA=00, INS=D8, P1=0, P2=fragmentation-flag, P3=fragment-length
   F-First=b01, F-Last=b10, F-More=b00

   When application AEAD is opened a two bits flag (F-Encrypt, F-
   Decrypt) indicates the cryptographic operation:
   - P2=b01= F-Decrypt, decryption
   - P2=b10= F-Encrypt, encryption
   - P2=b00= Standalone embedded application.

If F-Last is not set, the ISO7816 response is always 9000 when no
error occurs. For the last fragment five cases may occur:
- sw-ok: no error, no record message returned, response = 9000.
- sw-open, no error, no record message returned, TLS application
AEAD is opened, for example response =9001.
- sw-close: no error, , no record message returned, TLS application
AEAD is closed, for example response =9002
- sw-error: error, no record message returned.
- sw-more(size): no error, a message or message fragment is ready.
For example sw-more(size)= 61xy, in which xy is the size of the
first fragment.

```
                  TCP/IP Node                  Secure Element
                      |                              |
                      |-RECV(F-First, Frag#1)-------->|
                      |<------------------sw-ok 9000-|
                      |-RECV(F-More,  Frag#i)-------->|
                      |<----------------sw-ok= 9000-|
                      |-RECV(F-Last,  Frag#n)-------->|
                      |<----------------sw-ok= 9000-|
                      |<--------------sw-open= 9001-|
                      |<-------------sw-close= 9002-|
                      |<---------sw-more(size)= 61xy-|
```

  Figure 4. Receiving record packet, segmentation mechanism.

3.2.2 Sending Record Packet

 A sending procedure starts by the reception of a sw-more(size)
 status, ending a response. This event may occur at the end of RECV
 procedure (see figure 6) or after TLS state machine reset (see
 figure 5).

 A RECV(F-First, No-Frag) request resets the TLS state machine. For
 TLS client a sw-more(size) status is returned. For TLS server the
 sw-ok status is returned.

```
      TCP/IP Node                    Secure Element
         |                              |
         |-RECV(F-First, No-Frag)------->|=> Reset State Machine
         |<----------------sw-ok= 9000-|  Server
         |<---------sw-more(size)= 61xy-|  Client
```

 Figure 5. Starting the SEND procedure after RESET request.

```
      TCP/IP Node                    Secure Element
         |                              |
         |-RECV(F-Last, Last-Fragment)-->| => End of Message
         |<---------sw-more(size)= 61xy-|  Client
```

 Figure 6. Starting SEND procedure after the end of RECV procedure.

The SEND(size) reads a record fragment, whose length is equal to
size. It MAY be necessary to adjust the SEND size (see figure 7).
Typically at the end of RECV procedure, the size indicated by the
sw-more(size) status is an expected fragment length. In that case
the status sw-retry status (for example 6Cxy) indicates the fragment
size.

```
    TCP/IP Node                        Secure Element
         │                                  │
         │-RECV(F-Last, Last-Frag)------->│  => End of Message
         │<----------sw-more(size)= 61xy-│
         │  SEND(size)------------------->│
         │  <------- sw-retry(size')=6Czt-│
         │  SEND(size')------------------>│
```
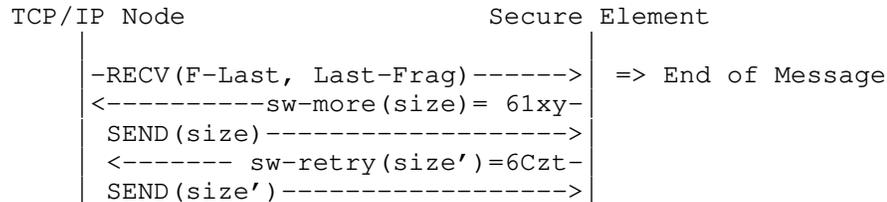
Figure 7. Adjusting SEND size.

The SEND(size) request is encoded as :
CLA=0, INS=C0, P1=0, P2=0, P3=size

The SEND procedure (see Figure 8) is a set of SEND requests, which
read record packet fragments.

```
   TCP/IP Node                              Secure Element
     │                                          │
     │<-------------- ---sw-more(size#1)= 61xy-│
     │-SEND(size#1)---------------------------->│
     │<--------------Frag#1 || sw-more(size#2)-│
     │-SEND(size#i)---------------------------->│
     │<----------Frag#i || sw-more(size#[i+1])-│
     │-SEND(size#n)---------------------------->│
     │<----------------------Frag#n || sw-ok)-│=> SEND End
     │<-----------Frag#n || sw-more(next-size)-│=> SEND Continue
     │<--------------------Frag#n || sw-open)-│=> Open
     │<--------------------Frag#n || sw-close-│=> Close
```
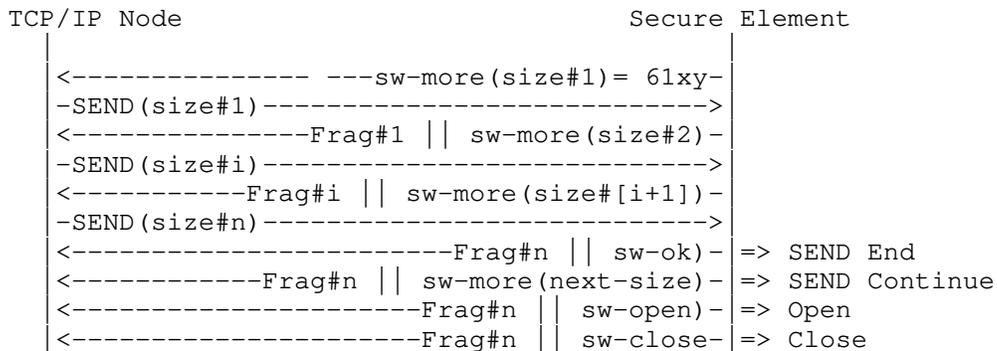
Figure 8. The SEND procedure

At the end of SEND procedure four events MAY occur:
- End of SEND procedure (status = sw-ok). No more record packets are
available.
- SEND procedure to be continued (status = sw-more(size)). Another
record packet is available.
- End of SEND procedure, application AEAD is ready for use (status =
sw-open)
- End of SEND procedure, application AEAD is closed (status = sw-
close)

   3.2.4 RECV and SEND procedure for open application AEAD

  When the application AEAD is opened RECV performs decryption and
  encryption operations (see figure 9).

  For decryption operation (RECV(F-Decrypt)) the RECV procedure pushes
  the incoming record packet. The returned payload by the SEND
  procedure is the decrypted message ended by the protocol byte.

  For encryption operation (RECV(F-Encrypt)) the RECV procedure pushes
  the content to encrypt ended by the associated protocol byte. The
  returned payload by the SEND procedure is a record packet, including
  the encrypted content.

```
    TCP/IP Node                                     Secure Element
       |                                                  |
       |-RECV(F-First, Frag#1)--------------------------->|
       |<--------------------------------sw-ok= 9000-|
       |-RECV(F-More,  Frag#i)--------------------------->|
       |<--------------------------------sw-ok= 9000-|
       |-RECV(F-Decrypt/F-Encrypt, F-Last, Frag#n)------>|
       |<-----------------------sw-more(size#1)= 61xy-|
       |-SEND(size#1)------------------------------------>|
       |<--------------------Frag#1 || sw-more(size#2)-|
       |-SEND(size#i)------------------------------------>|
       |<-----------------Frag#i || sw-more(size#[i+1])-|
       |-SEND(size#n)------------------------------------>|
       |<--------------------------Frag#n || sw-ok)-|=> SEND End
       |<--------------------------Frag#n || sw-close-|=> Close
```
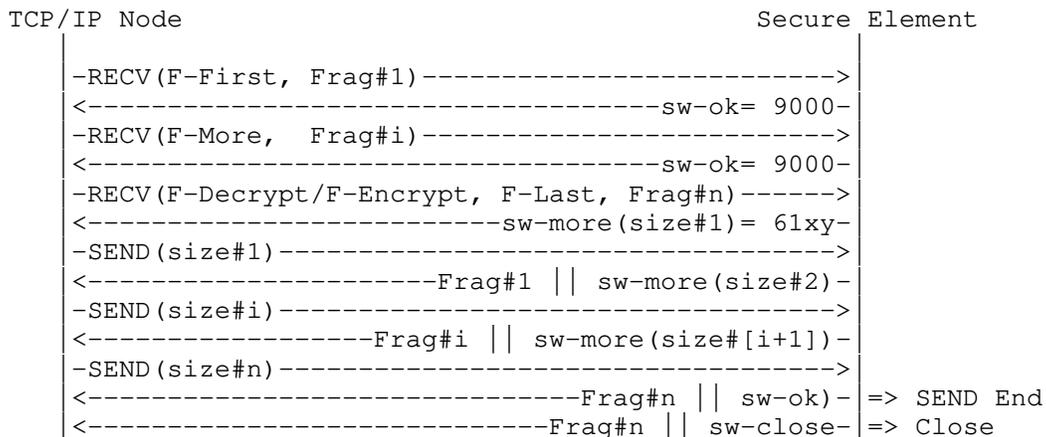
    Figure 9. Decryption/Encryption operations.

 3.3 TLS state machine

   The state machine manages TLS flights, it determines the next record
   packet to be received and checked, and the next record packet to be
   built and sent. The number of states and their order is dependent on
   the TLS-SE role (client or server), and on the supported working
   mode (pre shared key, server with certificate, server and client
   with certificate). Figure 10 details an example of state machine for
   TLS-SE server, using pre-shared key. The ordered list of states
   comprises: S-Ready, S-Extensions, S-SFinished, S-ClientCCS, S-
   CFinished and S-Open.

```
        TCP/IP Node                  Secure Element
            |                            |
            |-RESET----------------------->|
            |<----------------------sw-ok-|  state = S-Ready
   Client   |-RECV(F-First,Frag#1)--------->|
   Hello    |<----------------------sw-ok-|
            |-RECV(F-Last,Frag#2)---------->|
            |                            |  Check-ClientHello
            |                            |  Make-ServerHello
            |<---------------sw-more(size)-|  state= S-Extensions
            |-SEND(size)------------------->|
   Server   |<--------Packet||sw-more(size)-|
   Hello    |                            |
            |-SEND(size)------------------->|  Make-Extensions
   Server   |<--------Packet||sw-more(size)-|  state= S-SFinished
   Encrypted|                            |
   Extension|-SEND(size)------------------->|
            |                            |  Make-ServerFinished
   Server   |<---------------Packet||sw-ok-|  state= S-ClientCCS
   Finished |                            |
            |                            |
   ClientCCS|-RECV(F-First,F-Last,Packet)-->|  Check-ClientCCS
            |<----------------------sw-ok-|  state= S-CFinished
            |                            |
            |                            |
   Client   |-RECV(F-First,F-Last,Packet)->-|  Check-CFinished
   Finished |<---------------------sw-open-|  state= S-Open
            |                            |
   Packet   |-RECV(F-Decrypt,Packet)------->|  Decrypt Packet
            |<---------------sw-more(size)-|  Clear Form Message
            |-SEND(size)------------------->|
            |<------ -Message||ptcol||sw-ok-|
            |                            |
   Message  |-RECV(F-Encrypt,Message)------>|  Encrypt
            |<---------------sw-more(size)-|  Record Packet
            |-SEND(size)------------------->|
            |<------ -Record Packet ||sw-ok-|
            |                            |
```

       Figure 10. TLS-SE server with pre-share key state machine
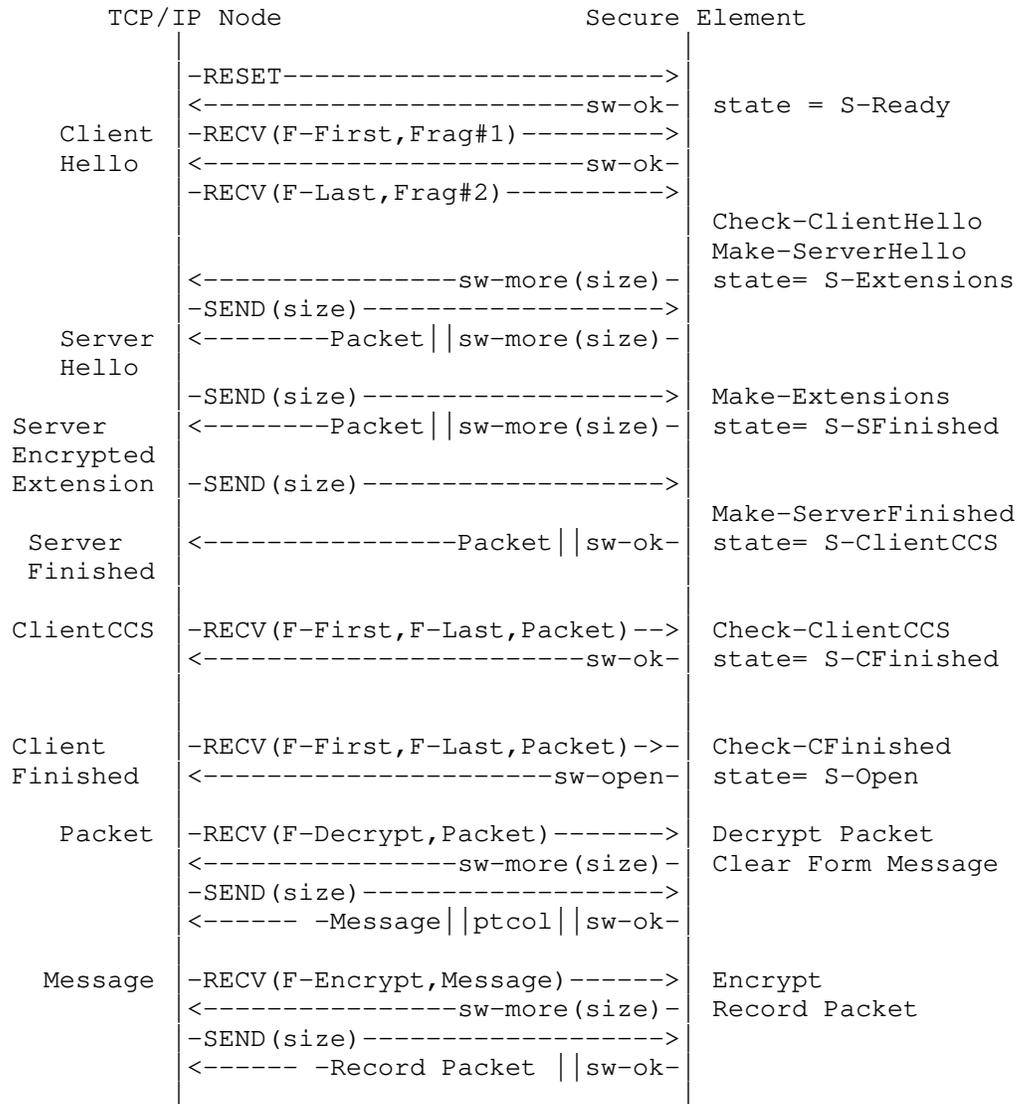

3.4 TLS library

   The TLS-SE library is a set of procedures that check, according to
   the state machine, incoming record packets and build outgoing record
   packets. In figure 10 the TLS library comprises the following
   elements: Check-ClientHello, Check-ClientCCS, Check-ClientFinished,
   Make-ServerHello, Make-EncryptedExensions, and MakeServerFinished.

4 ISO7816 interface

   The RECV and SEND binary encoding is shown by figure 11

```
+------+-----+-----+------------+-----------+----------+---------+
+ name | CLA | INS |     P1     |    P2     |    P3    | Payload |
+------+-----+-----+------------+-----------+----------+---------+
| RECV |  00 |  D8 | 01= Decrypt| 01= First | Fragment |         |
|      |     |     | 02= Encrypt| 02= Last  | Length   |   Yes   |
|      |     |     |            |           | 0= RESET |         |
+------+-----+-----+------------+-----------+----------+---------+
| SEND |  00 |  C0 |     00     |    00     | Incoming |   No    |
|      |     |     |            |           | Length   |         |
+------+-----+-----+------------+-----------+----------+---------+
```

   Figure 11. RECV and SEND ISO7816 requests binary encoding

   The status word binary encoding is shown by figure 12. Two binary
   encoding of sw-more MUST be supported. In the T=0 context, SE
   operating system returns the 61xy status when a request including a
   payload, induces a response with a payload. The status 9Fxy is
   managed by the application in order to notify response size to be
   returned. The TLS-SE application MAY use 61xy status, but this could
   induce interoperability issues.

```
+----------------+-----+------+
|      name      | SW1 | SW2  |
+----------------+-----+------+
|     sw-ok      |  90 |  00  |
+----------------+-----+------+
|  sw-more(size) |  61 | size |
|                |  9F | size |
+----------------+-----+------+
| sw-retry(size) |  6C + size |
+----------------+-----+------+
|    sw-open     |  90 |  01  |
+----------------+-----+------+
|    sw-close    |  90 |  02  |
+----------------+-----+------+
|    sw-error    |  6D |error |
|                |  6F |number|
+----------------+-----+------+
```

   Figure 12. ISO7816 status word binary encoding

5 ISO 7816 Use Case

   Below is an illustration of TLS-SE server, using a pre-shared key
   (PSK) with DHE over the secp256r1 curve, and the cipher suite AES-
   128-CCM-SHA256. The time consumed by handshake is about 1.4s.

   PSK=
   0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20
   DHE=
   037E6E633541EC03DB700A28E7DABB74F8E84D4A28E5F024B46F468A7821305D

   RECV(Client Hello)

   Tx: 00 D8 00 01 F0 16 03 03 00 F2 01 00 00 EE 03 03
       4E 65 53 05 52 AB 3E 83 14 0B 2F 9C 2F D7 BC 16
       F9 F5 C4 A9 86 CA 3F C8 8C 6E 8C D1 10 BB B1 57
       00 00 02 13 04 01 00 00 C3 00 2D 00 03 02 00 01
       00 2B 00 03 02 03 04 00 0D 00 1E 00 1C 06 03 05
       03 04 03 02 03 08 06 08 0B 08 05 08 0A 08 04 08
       09 06 01 05 01 04 01 02 01 00 33 00 47 00 45 00
       17 00 41 04 9A 1E 0A D8 40 88 D4 21 D1 55 D7 F2
       8F 78 4C 28 75 F5 19 CA 12 71 96 92 C4 07 8F B4
       35 42 57 E7 64 24 C1 BC 5D 89 0E F4 08 FD 25 8D
       24 F4 64 BB C3 F4 80 D3 BF 2C 23 A0 F9 2D A7 88
       0C 5B 44 53 00 0A 00 06 00 04 00 18 00 17 00 29
       00 3A 00 15 00 0F 43 6C 69 65 6E 74 5F 69 64 65
       6E 74 69 74 79 00 00 00 00 00 21 20 CC 05 4A 9F
       DE 70 E9 96 D6 01 69 61 F5 9A 78 20 D9 FC 6D ED
       4C C6 0A 7B 0D
   Rx: 90 00   [47 ms]
   Tx: 00 D8 00 02 07 4B 68 8F 4E B9 B2 CA

   Rx: 61 86   [879 ms]

   SEND(Server Hello)

   Tx: 00 C0 00 00 86
   Rx: 16 03 03 00 81 02 00 00 7D 03 03 5C 78 A4 E1 93
       34 D7 D9 64 B2 85 64 1B E4 76 63 94 39 1F 4A 15
       27 0A A4 C6 A0 C6 93 D9 E2 16 4D 00 13 04 00 00
       55 00 29 00 02 00 00 00 33 00 45 00 17 00 41 04
       25 C9 16 94 8B 39 51 D2 8E 88 70 F7 F5 4E 6C 31
       62 93 B1 65 55 2C 30 B2 5E 75 6C D8 FE AF DA A7
       67 D8 AD A7 BE 68 54 EA 3E A0 0B 4D CC 62 93 96
       38 07 68 29 3E D5 E6 0C 25 4A EA 12 C9 F8 99 7F
       00 2B 00 02 03 04 9F 1C   [32 ms]

SEND(Server Encrypted Extensions)

Tx: 00 C0 00 00 1C

Rx: 17 03 03 00 17 E6 04 4A 52 1A 50 B5 54 D8 73 5E
    00 F4 FD 66 BB B3 74 50 99 36 C8 08 9F 3A [78 ms]

SEND(Server Encrypted Finished)

Tx: 00 C0 00 00 3A

Rx: 17 03 03 00 35 CB CA 03 3E E4 34 7E D2 0C 7C 24
    C1 8F 39 A2 74 39 24 47 78 BE 94 95 7A 31 EC 03
    D5 0C A8 1C 46 04 05 F2 83 3E 99 0D AD D6 66 63
    60 23 F8 5D 7B 77 0F 95 18 35 90 00 [185 ms]

RECV(Client Encrypted Finished)

Tx: 00 D8 00 03 3A 17 03 03 00 35 BC 29 18 D1 B8 4B
    C0 3F 6F 81 79 D9 7E FD 58 E3 76 EA 61 13 9C 3E
    40 0F 34 CD 94 CE C1 44 CB 76 70 7D DA 8A 54 69
    41 D9 80 CD 5D 52 8F E5 38 D8 52 92 20 54 5E
Rx: 90 01 [389 ms]

TLS13 session is open

Decryption of incoming Record Packet
RECV(Decrypt, Packet)
Tx: 00 D8 01 03 24 17 03 03 00 1F 56 E2 D5 B5 C4 A6
    E2 3E 54 56 5A C4 2D E9 99 F3 58 22 34 15 15 A7
    96 FD 0E B0 61 60 4C 52 87
Rx: 61 0F [78 ms]

SEND(Message)

Tx: 00 C0 00 00 0F

Rx: 68 65 6C 6C 6F 20 77 6F 72 6C 64 21 0D 0A 17 90
    00 [15 ms]
Rx: hello world! ptcol=17

Encryption of message

RECV(Encrypt,Message)
Tx: 00 D8 02 03 0F 68 65 6C 6C 6F 20 77 6F 72 6C 64
    21 0D 0A 17

Rx: 61 24 [79 ms]

    SEND(Record Packet)

    Tx: 00 C0 00 00 24
    Rx: 17 03 03 00 1F 6F 78 FF 68 0F CA 9E 31 53 2C 96
        B3 FA D7 B0 51 1B 92 81 35 3D DB FE E9 18 A7 DF
        36 2F A5 27 90 00 [16 ms]

5 TLS-SE Name

    According to ISO7816 standards, secure elements return upon reset a
    set of bytes called Answer to Reset (ATR). ATR comprises at least
    two bytes (TS, T0). The LSB nibble of T0 indicates the number of
    historical bytes (ranging from 0 to 15). Historical bytes (HB) are
    located at the end of ATR. Historical bytes can be programmed by
    standardized API, and therefore MAY be used for secure element
    naming.

6 Server Name Indication

    According to [RFC6066] Server Name Indication extension is used to
    route TLS packets toward a virtual host.
    Multiple TLS-SE devices, embedding standalone applications, can be
    hosted by an internet node. In this case SNI extension MAY be used
    in order to select the right secure element, whose name, typically
    stored in historical bytes, is determined from SNI.

7 IANA Considerations

    This draft does not require any action from IANA.

8 Security Considerations

    This entire document is about security.

9 References

9.1 Normative References

    [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol
    Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018,
    https://www.rfc-editor.org/info/rfc8446.

    [RFC6066] Eastlake 3rd, D., "Transport Layer [RFC6066] Security
    (TLS) Extensions: Extension Definitions", RFC 6066, DOI
    10.17487/RFC6066, January 2011.

    [ISO7816] ISO 7816, "Cards Identification – Integrated Circuit Cards
    with Contacts", The International Organization for Standardization
    (ISO).

   [IEEE1363] IEEE, "IEEE Standard Specifications for Public Key
   Cryptography", IEEE Std. 1363-2000, DOI 10.1109/IEEESTD.2000.92292.

9.2 Informative References

   [IM] Urien, P., "Identity Module for TLS Version 1.3", draft-urien-
   tls-im-03.txt, July 2020.

10 Authors' Addresses

   Pascal Urien
   Telecom Paris
   19 place Marguerite Perey
   91120 Palaiseau   Phone: NA
   France   Email: Pascal.Urien@telecom-paris.fr