Internet Engineering Task Force
RAP Working Group
Expiration: May 2003
draft-ietf-rap-access-bind-02.txt

Walter Weiss
Ellacoya Networks
John Vollbrecht
David Spence
David Rago
InterLink Networks
Cees de Laat
Univ. of Amsterdam
Freek Dijkstra
Univ. of Utrecht
Leon Gommans
Enterasys
Amol Kulkarni
Ravi Sahita
Intel
Kwok Ho Chan
Nortel Networks

Framework for Binding Access Control to COPS Provisioning

Last Updated: 11/4/02

Status of this Memo

Conventions used in this document

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].

Abstract

There is an ever-growing distinction between edge and core functionality. While the core continues to focus on stability and pure forwarding functionality, the edges increasingly need to deal with dynamic capabilities like QoS management, VPN encapsulations, encryption, dynamic steering and service monitoring. The dynamic deployment of these functions is dependent on specific contextual knowledge such as the physical location of the data stream and the identity of the client or system generating the data.

In many environments, there is a requirement to both bind resource consumption to an identity or account, and also to quickly and efficiently provision the appropriate set of policies for that client or account. It is possible to provide this capability with a collection of currently available protocols. However, the synchronization of account and provisioning information between these protocols makes this approach extremely unwieldy.

This memo offers a solution in the form of a single COPS PIB capable not only of supporting all the above requirements but also offering a scalable means for extending the provisioning capabilities as new technologies are standardized. Specifically, we offer a mechanism for provisioning the criteria for initiating dynamic event notifications from the PEP as well as a means for propagating identity credentials received by the PEP to allow the PDP to validate a client identity or an account as part of the event notification process.

1. Introduction

There are two sides to access control. The one side is the negotiation between the client and the edge device (also known as

the policy enforcement point), for example between a workstation and an Ethernet switch supporting authentication protocols like 802.1x and PPPoE. The other side of a typical access control model is an interaction between the edge device (PEP) and a PDP, such that the PDP performs the client/account validation process and notifies the PEP of the result. This separation of access control into two parts is necessary because invariably the PEP does not have the capacity to store all possible identities and credentials. This information is typically stored in a server (PDP).

In reality access control can include authentication as one piece of a larger authorization process. As such, authorization has much in common with RSVP [RSVP]. When an RSVP service request is made, the PDP evaluates a set of criteria including what is being requested, what the availability of specific resources are, the identity of the requestor, and even the location of the requestor. All these criteria are taken into consideration before determining whether the

RSVP request should be honored. In addition, if the request is honored, specific provisioning actions may be taken to bind or manage the request. Similarly, the ability for a PDP to respond to a non-RSVP service request potentially requires all the same information of a traditional RSVP request in order to determine whether the request should be accepted or rejected.

It is also important to note that a service request should not just be restricted to network access. In practice, there are many instances where a determination of access privileges requires an explicit decision. For instance, there are scenarios where limited network access is granted based on the specific criteria, but subsequent authorization is required to access a premium resource that requires incremental authentication (via HTTP for example). Another possible scenario occurs when initial access is authorized based on one set of credentials, but usage of a specific resource requires an examination of an account balance. These authorizations will be referred to as _PEP Events_ to denote the fact that PEP is generating an event indicating a request for some type of service.

In order to support the broad variety of potential PEP Events, there must be a way of provisioning the criteria for generating the PEP Event. In the most common case the PEP Event is generated as the result of some type of packet oriented event such as activity on a link, traffic of a particular type or traffic from a new, unknown IP address.

This leads to a useful observation: In many cases, PEP Events need to be defined within the context of a network data path. In other words, the data path mechanisms defined in the DiffServ informal

model [MODEL] and the DiffServ PIB [DSPIB] provide a means for specifying the circumstances for generating a PEP Event by reusing elements from the model like the qosDatapathTable table and the qosClfrTable table in the DiffServ PIB.

The second circumstance for generating an event from the PEP is through a programmatic interface in between the PEP and an external service such as the policy control interface in an RSVP service. In these instances, this PIB is used to configure the PEP to accept specific events through the API. Using COPS provisioning, a PEP can be configured to generate events for one or more types of RSVP events such as a new PATH request or a new RESV request.

Another consideration is the variety of information that can potentially be included in a PEP Event. For instance, a PEP Event could pass information about the client (domain), the physical port (dial up interface), L2 headers, L3 headers, encapsulation headers (tunnels), cookies, and additional information already negotiated prior to generating the PEP Event. Given the amount of information that could be sent with the PEP Event, it is reasonable to allow the PDP to configure the PEP with the set of information the PDP would like to have included with a specific type of PEP Event.

PEP Events provide a convenient means for the PEP to signal an event that requires specific actions. A PDP authorization for access to specific resources (and the potential verification of identity) is one example of an event that not only requires a response but also some potential provisioning work. It is interesting to note how neatly RSVP decision support fits into this model. In the original COPS design [COPS], the RESV request was sent in a COPS request and a COPS response message determined whether the reservation should be accepted or not. The enhancements provided by this PIB not only allow RSVP messages to generate PEP events (also called access requests), but also explicitly provision QoS resources, using COPS-PR [COPSPR], to support the reservation. This generalizes COPS for RSVP and allows it to evolve to the COPS-PR model.

There are a number of situations where Events and associated provisioning need to be negotiated quickly. Mobile-IP applications in particular require speedy resolution of PEP Events. This implies that the combination of PEP Events and provisioning needs to be completed with the minimum number of communication legs (round trips).

2. Paradigm for the Access Bind PIB

There are several key aspects to this PIB. First there is the

ability to provisioning for future authorization events, known as PEP Events. Second, there is a set of tables that are used to notify the PDP of an attempt to access managed resources. These tables can also include credentials necessary to verify client identity. Finally, there are tables that determine how dialogs (COPS Request Handles) between the PEP and PDP should be grouped. In order to provide concurrency between competing events and provisioning requests, there must be a means for determining which PEP Events require a new COPS Request Handle and which should use existing handles.

2.1. Event Handler Concepts

This section introduces the concept of an Event Handler. Much of what is described in this paper is based on the Event Handler.

Event Handlers are implemented in PEPs and configured by PDPs. Event Handlers are provisioned by standard COPS-PR protocol sequences. A PEP will announce what Event Handlers are available in the capabilities table of the COPS-PR Request message. The PDP will provision the Event Handlers with Decision messages.

Once an Event Handler is provisioned it is responsible for identifying packets or API requests that require the PDP to be notified with an Event Message.

The general model for Event Message requests includes a client, a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP). In

this model, the PEP is the interface to the client, and the Event Handler is the part of the PEP that is responsible for recognizing the conditions for client authorization, generating the Event Message to the PDP, and communicating with the Client, if necessary, to get identity or other information.

The Event Handler takes a signal or message from the client and translates it into an Event Message to send to the PDP. It takes the provisioning Decision from the PDP and, in cases where the client is aware of the authorization process, does what is needed to communicate the Decision to the client.

The Event Message is sent from the PEP to the PDP. The PDP uses the Event Message to determine the appropriate provisioning steps. In some cases identity verification may require sending some intermediate messages to authenticate the client prior to provisioning the PEP with the policies appropriate to the client. The PEP then returns a Report to the PDP confirming what was provisioned by the Decision.

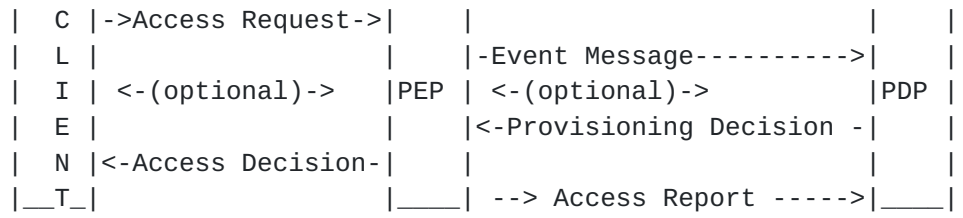


Figure 2.1: Access Control Protocol Sequence

This paper is primarily concerned with the function of the Event Handler and the communication between the PEP and PDP. Communication between the Client and PEP is assumed to be something like PPP or 802.1, and the capabilities described here should work with any Client/PEP communication method.

The PEP Event Message and PDP Provisioning Decision sequence is similar to the _classical_ COPS RSVP model. The Report confirming that the Decision was installed correctly on the PEP is an extra message beyond what is included in the RSVP sequence. We believe this is a good approach, but expect further discussion (It is interesting to note that RADIUS does not send an acknowledgement of Access Accepts/Rejects, and the DIAMETER drafts specify no acknowledgement, but do expect a negative message if the Reply cannot be processed correctly).

An Event Handler is a data path element in the PEP. Each Event Handler has a _selector_ that identifies packets that should cause Event Messages (See section 3.1.2). The selector essentially divides all packets into two categories, in the first, the Event Handler is responsible for generating Event Messages; in the other, it just passes the packets to the next data path element. For example, if an

Event Handler's selector is _All new source IP addresses_, an incoming packet's Source IP address is examined and if it is old, the packet is passed on to the next data path element without further processing. If the source IP address is new or unknown, an Event Message is generated and this packet may follow a different sequence of data path elements.

Event Messages are grouped by COPS Request Handles. Each Event Message may cause a new COPS Request Handle to be generated or a set of Event Messages may all share the same COPS Request Handle. The distinction between selector and Request Handle is spelled out in section 3.1.4.4). Attributes in the Access Bind PIB are provided to identify which COPS Request Handle a given Event Message should use.

Event Handlers are designed to detect conditions in the PEP that

result in the sending of Event Messages to the PDP. The Access Bind PIB defines a class to specify the criteria for generating an event. In some cases an event is appropriate every time the criteria is met. In other instances an event is appropriate only on the first occurrence. The provisioning of the event criteria using traditional classifiers can be difficult since it is often the case that the PDP can't anticipate what the PEP will see. For instance, when it is desirable to generate events every time a new user or device is recognized, the PDP can't anticipate which devices will be recognized or the order in which they will occur. Filter expressions can be constructed that enable the description of a set of packet fields that must match and a set of packet fields that collectively represent a new, unique combination. The expressive capability of the Access Bind PIB allows the PDP to indicate to the PEP that one event should be generated the first time a Src IP address has been seen by the PEP, but not generate events for subsequent packets with the same Src IP address.

One interesting problem associated with event driven provisioning is avoiding blocking of one event due to provisioning activity for a different event. On the other hand, there are situations where serialization or ordering of events is important. We use COPS Request Handles to address both these needs. However, this requires explicit provisioning to indicate when new handles should be provisioned and which events should be processed through which handles. The approach taken in this paper is that the scope of the COPS Request Handle is defined by one or more Filter entries. Some of the filters are defined in the COPS Framework PIB [FWPIB] as well as PIB modules defined in this document. For example, if a Filter object specifies SRC IP address (10.20.0.0) and SRC IP Mask (FF.FF.0.0) each new IP address within the range 10.20.0.0 and 10.20.255.255 will trigger the creation of a new Handle. For this example, any packet with a SRC IP address that generates a new Event Message will use the existing handle if that handle was already defined for that specific SRC IP address.

When a packet arrives at the Event Handler, it first checks if it meets the criteria for generating an Event (event criteria will be

discussed later). In the example above, a packet with a SRC IP address of 10.25.12.100 would not match the range criteria and would be passed to the next data path element. If it is selected, then a check is made to see if it matches the criteria for an existing COPS Request Handle.

If it does not match the criteria for an existing COPS Request Handle, then the PEP instantiates a new Request Handle and sends an Event Message to the PDP using the new Handle. In either case the

PDP analyzes the Event Message, possibly sending additional messages back to the PEP to support authentication and provisioning for the new address. If authentication was performed, a final Authentication Result object is sent to the PEP to indicate the authentication was successful or not. This is needed to allow the PAP, CHAP and EAP authentication processes to report success back to the authenticating user.

2.1.1. Example - Ethernet IP Address Authorization

This (relatively simple) example assumes an edge device has an Ethernet interface and wants to require each new Source IP address arriving at one Ethernet port to be authorized before getting general access to the network. Assume also that some clients are to get preferred access (via DiffServ Marking).

In the example, the PEP is configured with a classifier that has explicit entries for each source address that has already been authenticated and a default classifier element matching all addresses that has an Event Handler as the next element. Since the default classifier element is only used if none of the other classifier elements match, the Event Handler is only invoked for new Src IP addresses that have not yet been explicitly provisioned into the classifier. Each non-default classifier element points to another classifier that lists the policies uniquely for that Src IP address. The addresses of _premium_ users are assigned a high QoS while the addresses of _normal_ users are assigned best effort QoS. Since the Event Handler is not terminating any packets, the Event Handler passes all packets through to the Best Effort Queue.

When the PEP comes up it sends information about its Event Handlers to the PDP in a capabilities table. After capability negotiation is complete, the PDP provisions a set of policies that configure the Event Handlers behind the Ethernet interface's data path. Each Event Handler Table will have a pointer to a (tagged) set of EventHandlerElement Tables that provide Filter matching and COPS Request Handle matching rules. In this case, the EventHandlerElement table will be provisioned to generate unique Request Handles and Events the first time it matches a new _SourceIPAddress._

Once the Event Handler is setup, it is able to process packets arriving at the Ethernet Interface. The Event Handler looks at all packets with Src IP addresses that have not been explicitly been

defined in the upstream Classifier and uses the event matching rules to check if the packet contains an unknown Src IP address within the configured range. If the packet matches an event matching rule, the

Event Handler checks what information the PDP requires from the Client (e.g. username and credential), and collects this information. The PEP then checks to see if it should use an existing Request Handle or create a new one. In this example, each new address gets a unique COPS Request Handle so that all the address-specific (user specific) policies (and feedback information) are managed through a single COPS dialog. A unique handle also has the benefit of automatically removing all objects provisioned through the Handle when the Handle is deleted (the user ends their session). After the Request Handle is set up, an Event Message is sent to the PDP containing the user information including address, port, and credential information.

The PDP checks the information passed in the Event Message, authenticates the client (if required), and decides which policy should apply to that IP address. It sends a Provisioning Decision, containing the appropriate policy (add classifier element for the new address and set the next element of the classifier element to the _premium_ queue) to the PEP using the newly created Request Handle.

Additional examples using the Access Bind PIB to support RSVP, 802.11, and other protocols are described in section 8.

2.1.2. Context Data

As mentioned previously, Event Messages frequently require information beyond just the identity of the client. Information about the physical interface, the protocols being used, and the protocol bindings (VLANs, IP addresses, etc.) may also be required to provide enough information to the PDP to provide proper provisioning guidance. Therefore a mechanism is required that allows the PDP to specify what information is needed.

With the advent of Tunnels, the same headers can be repeated (nested) within a single client message. This makes identification of specific attributes such as IP Addresses difficult because it is unclear whether the PDP needs the IP Address in the innermost or outermost header. This gets even more complicated when more than two layers are involved (i.e. VLAN and MPLS label stacking). The ContextData class, described in more detail below, allows the PDP to explicitly specify the set of nested headers that it needs more details on. This can either be specified in from the outermost header or the innermost header, as well as all headers of a particular type.

Since the volume of information can be quite large and is very device and interface specific, it is appropriate to organize the information into manageable chunks. This approach was a compromise between two extremes. One extreme is one large data structure with

all possible information. The other extreme is specifying each attribute explicitly. The first extreme is not viable because it is difficult to adapt to new types of information. The second alternative is very configuration intensive, particularly for header data that must distinguish inner and outer headers. The choice to group context data into classes and request the data at the class level is not without problems. If the PDP is only interested in a single attribute within a given class, there is no way to specify this. Hence the PEP has to fill in the entire class and the PDP has to parse the entire class to find the appropriate attribute.

In order for the PDP to specify which chunks of context data it needs, this PIB defines a table called the ContextData class that allows the PDP to specify the tables it needs. This table is discussed in more detail in sections 3.1.3 and 3.1.4.5. The messages used to send ContextData are discussed in section 7

2.1.3. Policy Distribution and Management

One of the purposes of this paper is to demonstrate how authorization and authentication can be bound to traditional COPS provisioning. Stated somewhat differently, this paper provides the means for seamlessly integrating outsourcing with provisioning using only PIBs. Authorization, Authentication, and COPS/RSVP are all forms of outsourcing because they are all triggered by events in the PEP and depend on decision support from the PDP. Earlier sections have gone into fair detail in describing how the PEP can generate Event Messages. However, we have not yet discussed how these semantics integrate with traditional COPS-PR provisioning semantics.

There are two aspects to provisioning that need to be considered. First is the provisioning of the Event Handlers themselves. Section 2.1 went into some detail describing how Event Handlers are provisioned using policy decisions. More details on the Event Handler tables can be found in sections 3.1.1 and 3.1.4. In addition the provisioning messages used to configure Event Handlers are also described in section 7.1.

The second aspect of provisioning is the use of standard provisioning decisions to bind policies to authorized clients. The goal in binding events to policies was to minimize reconfiguration.

The process for this binding is as follows. An Event Handler can be configured to generate COPS Request Handles and trigger an Event Message based on specific criteria. These criteria explicitly scope the Request Handle. For example, if the criteria were one per unique source IP address, then there would be one Request Handle for each unique source IP address and all policies bound to that Request

Handle would typically operate on all traffic with that source IP address. Note that the criteria that scope a Request Handle could also be a unique protocol, unique VLAN, or each unique RSVP RESV message. It is also worth noting that the Request Handle bounding

criteria could also be a unique combination of field values such as a VLAN and TCP Port Number.

With the scope of a Handle specified, the Event Handler can instantiate new Handles in conjunction with the Event Message.

This PIB has been designed to provision Event Handlers as well as Policies once and bind them together dynamically. As described above, each Request Handle can manage a set of policies. However, in most cases, these policies reference data path elements that are shared by multiple Handles. For example, a new IP address may generate a unique Request Handle that in turn provisions one or more elements in the Classifier table. However, these elements may in turn point to other data path elements, such as queues or meters that are shared across multiple independent IP address classifier elements.

2.2. Event Handlers and Application-Specific PIBs

The Access Bind PIB is actually a modular set of PIBs. The Common PIB contains the Event Handler and it's associated structures. An extension PIB is also provided to support user authentication. This PIB is provided because only a subset of Events require identity management. Other PIBS are included in this document to support a variety of applications. In the future, these PIBS may be specified in independent documents. The Application-Specific PIBs minimize the number of COPS-PR classes that must be implemented in order to support Event Handler functionality for the many applications that require policy outsourcing.

2.3. Passive Monitoring vs. Programmatic API

The Event Handler is designed to operate in two specific scenarios. The first is a passive monitoring environment. In this mode, the Event Handler can be provisioned to detect specific types of traffic and generate events to the PDP based on the traffic. The Event Handler does not alter the packets in any way. However, packets may be sent to different packet processing engines depending on the decisions the PDP installs after responding to the event. The Passive Monitoring mode was designed to operate within the context of the DiffServ data path model. This model is discussed in more detail in Section 2.3.1.

In the second scenario, no packets are analyzed because some intermediate system is processing the packets and generating events.

In this mode, the system needs the help of the PDP to continue processing. Therefore, the system uses a signaling API to interact with the Event Handler, which in turn generates the events and receives the decisions. This mode is particularly useful for RSVP. Since the RSVP engine is processing the actual PATH and RESV messages, there are no packets for the Event Handler to process or analyze. In fact, the traditional COPS model defines a mapping of the RSVP policy engine to COPS messages. The Access Bind PIB is constructed to support a programmatic interface to the Event Handler. Further, the Programmatic API uses the same mechanisms as

the passive monitoring mode to configure new policies in intermediate systems. For instance, a RESV event received by the Event Handler through the programmatic interface and propagated to the PDP allows the PDP to generate data path decisions that can be installed in the intermediate system through the programmatic API. The concepts behind this model are discussed in greater depth in Section 2.3.2.

It is worth noting that both these modes are abstractions that may be equally applicable. It is possible to represent a service using the data path model and it is possible to represent a packet processing engine as a service with a programmatic interface to the PEP. It is a design decision that dictates the preferred approach for processing PEP events. The advantage of the passive monitoring approach is that it can more accurately represent the behavior of the system. However the API is more tolerant in the areas of filter definition and COPS request handle management.

2.3.1. Interactions with DiffServ data path model

The DiffServ model [MODEL] and PIB [DSPIB] allow for flexible addition of new Data Path Functional Elements. The Event Handler is one such new Data Path Functional Element. Previous sections have already described how this PIB extends the existing DiffServ Informal Model and the DiffServ PIB. However, it is worth describing how this PIB enhances the basic DiffServ model. First and foremost, this new PIB provides a means for scaling the basic DiffServ model to the edges of the network that can have many interfaces and many specialized services. Previous PIBs only supported the static configuration of data paths. This meant that dynamic events such as binding of new clients to existing or new services were difficult to support because there was no way to anticipate new clients. In addition most provisioning was managing Classifiers on a per client per service basis, which scales geometrically as the number of clients and services increases.

This PIB addresses this problem by preserving the basic data path

semantics but separating the creation of dynamic (event driven) policies into a new data path component. This provides a stable data path for the generation of authorizations while also supporting a stable data path for the services that various clients may make use of. The linchpin of this PIB is the Event Handler, a new type of demultiplexor, that separates streams of traffic into individually grouped triggers that in turn support dynamic authorization. The policy provisioning that results from these events can be bound back to pre-defined policies to minimize the changes required to support new clients. As a result, with these PIB modules, service policies can be added or removed at the session level rather than the raw data path level.

So far we have only discussed the value of authorizing a client when the link notices a new IP address. However, it is worth noting that because the Event Handler is part of the data path definition, it is

far more flexible. For instance, the Event Handler can be placed behind a Classifier to explicitly authorize access to a specific part of the network or specific services. The Event Handler can also be the FailNext element behind a meter resulting in an authorization for the use of out-of-profile traffic. Bandwidth Brokers can use this approach or an Event Handler trapping RSVP RESV messages to support dynamic bandwidth allocation decisions. MPLS LSRs and LERs can use this to detect label path addition or modification events.

The integration of Event Handler as a Data Path Functional Element allows seamless integration with DiffServ provisioning. DiffServ network device mechanism policy control continues to be supported with the use of DiffServ PIB [DSPIB] with added functionality at the edge of the network with usage of the Event Handler.

The Policy Server level interaction with DiffServ comes naturally with the integration of Event Handler as a Data Path Functional Element when the network data model is common and scoped appropriately in the schema level, with the Event Handler becoming stimuli for DiffServ provisioning.

2.3.2. The Programmatic API to the Access Bind PIB

The programmatic API to the Access Bind PIB is actually an implementation specific abstraction that allows intermediate systems to interact with the Event Handler. This PIB only defines the messages that enable the Event Handler to generate events on behalf of intermediate systems. Since implementations of intermediate systems such as RSVP vary greatly both in features and usage, the actual API that maps the RSVP policy engine to the Event Handler is

dependent on the actual outsourcing requirements of the intermediate engine. However, the Access Bind PIB is extremely flexible and can accommodate a broad range of events and policy decisions.

The typical programmatic API will have interfaces (methods) that parallel the sequence of messages seen in the data path mode of operation. The intermediate system will first invoke the API to register itself with the PDP. The COPS-PR side of the API would generate a Capabilities message indicating that the device supports the protocol or service represented by the intermediate system. The functionality of the API will be described by using RSVP as an example. It should be noted that any other service (such as SIP, H.323, or 3GPP-go) that needs to outsource policy requests would work the same way. In the case of RSVP, the API might be a function like EventHdlrRegister(RSVP-type). The API would in turn generate a COPS-PR capabilities message indicating that RSVP can be provisioned and monitored through the event handler.

The next step is a response from the PDP that provisions a set of controls. The first is the provisioning of the Event Handler that will interact with RSVP via the API. When the Event Handler is provisioned, the RSVP service is notified that it may now outsource

RSVP reservation requests to the PDP through the API. Second, there may be several provisioning tasks that are configured in the RSVP service through the API. In many implementations of RSVP there may be many reservations requiring varying levels of QoS but only a few Queues to support the scheduling of various RSVP flows. In these situations the PDP may provision some or all of these queues using the Framework [FWPIB] and DiffServ PIB [DSPIB]. The API can then be used to map these provisioning requests to the actual RSVP implementation within the RSVP service. This allows the pre-configuration of queues, schedulers.

When a reservation is processed by the RSVP service, the API is used to notify the event handler of a new RSVP event. When the event handler is provisioned some of the provisioned structures specify what events (RESV, PATH, etc.) the PDP will respond to and what information must be included in an event message. The API may be invoked with a method like EventGenReq() with parameters that describe the type of message and the context data that the PDP requires and the COPS request handle to use for this reservation.

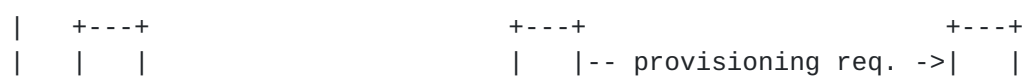
When the PDP completes the processing of the event, a set of decisions are sent back to the PEP. These decisions are handed to the RSVP service via the API. The decisions will determine how the reservation is to be processed. If several queues were pre-provisioned, the decisions may provision a classifier matching the

reservation's flow (4-tuple) and a meter that rate limits the traffic to the R-SPEC [RSVP]. The decisions would also indicate that the classifier should have the meter as the next element and that the meter would have the appropriate (pre-provisioned) queue as its next element. The API may be implemented as a synchronous or asynchronous interface. If the interface is synchronous, the API will return an indication back to the event handler as to whether the decisions were successfully installed or not. If the interface is asynchronous, the API will call the event handler explicitly to indicate success or failure. In either case, an explicit decision result message must be sent back to the PDP so that both the PEP and PDP are kept in sync on which policies have been installed in the PEP.

When the RSVP service determines that the reservation should be torn down, the API is used to close/terminate the COPS Request Handle. This action will cause both the PEP and the PDP to remove the decisions associated with this reservation (the classifier and meter).

3. Access Bind PIB

Figure 3.1 shows the basic operation of the Access Bind PIB. In the first phase, the Event Handler is provisioned in the PEP. This process enables the PEP to trap the configured events and pass them to the PDP. The process of trapping events may involve an intermediate step of authenticating the user or device. This step is represented in Figure 3.1 as an event message with possible authentication message exchanges between the PDP and the user/device. After the event (and optional authentication has been processed by the PDP, the PDP can provision the PEP with a set of policies specific to that user, device, or event. When the decision policies are installed, the PEP notifies the PDP that the install has succeeded or failed. In addition, if authentication was involved, the user or device may be notified that the authentication process has completed.



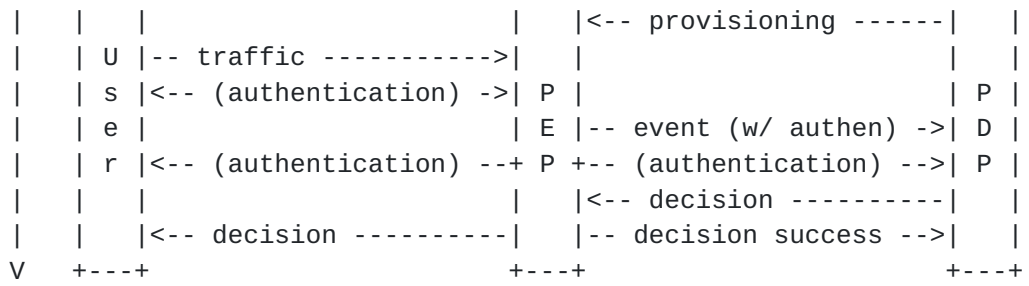


Figure 3.1: Typical message sequence when a trigger occurs.

In many scenarios, no identity management will be required and the authentication steps will not occur. This is why identity management classes have been place in a separate PIB, which is discussed in more detail in Section 4.

Section 3.1 will describe how the PDP installs an event handler into a PEP, and when the PEP should trigger an event. Section 3.2, about the event handling, will describe the actions that the PEP needs to perform when an action is triggered. This chapter will focus on generic event handlers; Section 5 will describe additional classes required to support event handlers within the context of specific signaling applications. Chapter 7 will describe the message sequence that follows the event request, as well as the message sequence associated with identity authentication.

3.1. The Event Handler

This Section will describe the concept of Events, Event Handlers, Event Handler Element, Event Handler Event Scopes, and Event Handler Handle Scopes.

In the framework described in this document, only events that match a predefined set of criteria can be sent from the PEP to the PDP. The main purpose of the Event Handler is to provision the PEP with that set of criteria. If the criteria provisioned by the PEP are met, the PEP must send an Event message to the PDP.

3.1.1. Functional Description

The PDP provisions the event handler onto the PEP. However, the PEP is the first machine to contact the PDP, typically at boot time of the PEP. The PEP and the PDP communicate with each other using common COPS-PR messages. After the PEP send a capability message indicating that it supports event handlers, the PDP will respond by provisioning the PEP with a set of configuration elements. These elements may include one or more instances (PRIs) of the Event

Handler class, the EventHdlrElement class, the EventHdlrScope class and optionally the ContextDataPointer class. The PEP will respond to this configuration request with a common COPS-PR report message indicating that these elements have been successfully provisioned.

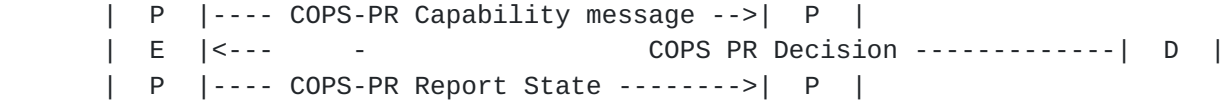


Figure 3.2: The PEP initialization sequence

The COPS Decision message that the PDP sends to the PEP should contain at least one EventHandler Instance. The EventHandler Entry is the base object which defines the behavior of the PEP when no event criteria are met. Each EventHandler is accompanied with one or more EventHandlerElements. Each EventHandlerElement describes the action which the PEP should take in case one of the event criteria is met.

The EventHandler and the set of EventHandlerElements are grouped together because each EventHandlerElement in the same group has the same TagId in the eventHdlrElementGrpId attribute. The EventHandler is associated with this group because it has the same value in the eventHdlrElementGrpId attribute.

The EventHandlerElement objects specify what actions should be taken if an event is triggered. To specify when an event must be triggered, the EventHandlerElement uses zero or more EventHdlrEventScopes. The Scopes are also grouped using a TagId, and have a precedence field. Each scope defines a simple condition, and all scopes from one group together form a complex boolean expression based on the eventHdlrEventScopePrecedence fields. If two scopes in the same group have a different precedence number, then the event criteria for the EventHandlerElement is met if one of the scopes condition is met. If two scopes in the same group have the same eventHdlrEventScopePrecedece fields, the event criteria are only met if BOTH conditions of the EventHdlrElements are met.

Take for example, an EventHandlerElement with an eventHdlrElementEventScope TagId (and thus the eventHdlrEventScopeGroup TagReferenceId of a certain group) set to 5, and there are 3 scopes in the group 5, with the eventHdlrEventScopePrecedence set to 2, 3, and 3 respectively for scope A, B and C. Then, only an event is triggered if (the criteria of scope A OR (the criteria of scope B or the criteria of scope C)) are met. The exact rules are explained in section 3.2.4.

3.1.2. Event Criteria behavior

One key aspect of the EventHdlrElement is the Event Criteria attribute. This attribute is used to describe whether unique events are one time events or repeatable events. For instance, every RSVP message may need to result in an Event Message. However, an Event Message may only be appropriate the first time a new Src IP address is seen. After that no events should be generated for that address. However other new addresses should still generate Event Messages. The Event Criteria attribute defines the frequency with which events should be generated. If the Event Criteria is set to 'One_Time', only one event will ever be generated for that EventHdlrElement when the first match occurs, irrespective of the number of subsequent matches. If the Event Criteria is set to 'Every_Time', each match will result in an Event Message. A hybrid case is defined called 'On_Change'. This option allows a subset of the filter attributes to be required for matching and a different set of attributes that must from a unique n-tuple in order to generate an Event Message. See Section 3.2.2 for more details of the behavior of the 'On_Change' attribute.

Whenever traffic arrives at the EventHandler for which an Event Message has not already been generated, it is compared against the FilterEntry objects of the EventHdlrEventScope objects referenced by the EventHdlrElement. If it matches the criteria specified in all of the FilterEntry objects, a new Event Message is generated and sent to the PDP. The value of EventHdlrElement's EventCriteria attribute in conjunction with the value of the Event Scope class's ChangeFlag attribute determine whether an Event Message will be generated.

For example, if a FilterEntry object specifies SRC IP address (10.20.0.0) and SRC IP Mask (FF.FF.0.0) and the EventCriteria is set to 'One_Time', the first address in the range of 10.20.0.0 and 10.20.255.255 will generate an event and no events will follow. If the EventCriteria is set to 'Every_Time' for the same attribute settings, each time a packet contains an IP address within the range an Event Message will be generated. If the EventCriteria is set to 'On_Change' and the eventHdlrEventScopeChangeFlag is set to True, each new IP address within the range 10.20.0.0 and 10.20.255.255 will trigger a new Event Message. However, as soon as a specific Src IP address like 10.20.15.109 has generated an Event Message, that specific address will no longer generate an event. If the EventCriteria is set to 'On_Change' and the eventHdlrEventScopeChangeFlag is set to False for the example

address range, than the eventHdlrEventScope instance with the ChangeFlag set to 'True' will determine uniqueness. In this scenario, the address range is acting as a strict filter that must be met without regard to which address in the range is responsible

or whether that address has been seen previously.

When multiple fields are specified for the filter and the ChangeFlag is set to 'True', each unique combination of field values generates an event. For example, if the SRC IP is assigned a range of 10.10.10.224 to 10.10.10.255 and DST Ports 80 to 90 then 10.10.10.240+80, 10.10.10.240+81, and 10.10.10.250+80 would all generate separate events. The combination of supporting multiple filters and being able to control precedence allows for the construction of both lists (10.10.x.x and 10.15.x.x) and combinations of disjoint headers in a single match criteria (any combination of 10.10.x.x and VLANs 100 to 120). See Section 3.2.4 for a detailed example of filter construction.

3.1.3. Context Data usage

For each application signaling protocol there are different pieces of information that the PDP needs in order to make a provisioning decision. In some cases the PDP may need IP header information. In other cases, it may need some state information internal to the PEP such as the activity timer for a connection or the number of bytes originating from a particular IP address. Since there are a huge number of potentially interesting pieces of information that the PDP may need, sending all the information can be expensive both in processor and bandwidth overhead. To address this issue, each EventHandlerElement instance can be independently provisioned with a list of classes that the PEP should send as part of an Event message. This list is constructed with a tag reference to a class called ContextData. Each instance of a ContextData class contains a class identifier (PRC). The class identifier specifies the type of ContextData class that should be passed with the Event message.

Typically a class will represent an autonomous structure such as an IP header or the fields of a RSVP reservation table entry. In some instances such as tunneling, there may be a list of entries that are applicable to the event. For this reason, the ContextData class has attributes that allow the PDP to indicate whether a specific entry in the list (relative to the beginning or end of the list) or all the entries in the list should be sent with an Event message. See Section 3.1.4.5 for details on organization of this class. Also see Sections 3.2.5 and 3.2.6.2.

3.1.4. Data Description

The following sections the classes defined in the Access Bind PIB.

3.1.4.1. EventHandler class

Instances of the Event Handler PRC are provisioned by the PDP on the PEP to catch specific events. The Event Handlers reference a group of eventHdlrElement PRIs that contain the scope of the event and specify the context data to send to the PDP when an event is caught.

The attributes of the EventHandler Class are as follows:

eventHandlerId (InstanceId)
Identifies an instance of this class.

eventHandlerElements (TagReferenceId)
A reference to a group of eventHdlrElement instances, each of which determines the scope (criteria for generating a new request) and what context information to send in a request.

eventHandlerNonMatchNext (Prid)
The data path for 'out of scope' traffic_ that is not matched by one of the eventHdlrElement's filter clauses.

3.1.4.2. EventHdlrElement class

The PDP installs EventHandlerElements as part of constructing the event handler. EventHandlerElements describe when events will be generated and which COPS request handles will be used to send the requests.

The purpose of the EventHdlrElement is to specify the characteristics of the EventHandler. The attributes in the EventHdlrElement provide maximal reuse by allowing multiple instances of an EventHandler to reuse the same EventHdlrElement instance. Each Instance of this PRC belongs to a group of eventHdlrElement PRIs. The group is identified by the eventHdlrElementGrpId attribute. These are provisioned by the PDP on the PEP to catch specific events. This PRC contains the scope of the event and specifies the context data type to send to the PDP when an event is caught.

Each EventHdlrElement constitutes a unique event semantic. Since the Event Scope, Handle Scope and Context Data are all bound to the EventHdlrElement, different EventHdlrElements can have different Event Scope (matching rules), Handle Scope (Handle generation rules), and Context Data (event specific data passed with the Event Message). This is why Event objects generated by the PEP reference both the Event Handler and the Event Handler Element that generated the event.

One key aspect of the EventHdlrElement is the Event Criteria attribute. This attribute is used to describe whether unique events are one time events or repeatable events. For instance, every RSVP message may need to result in a Event Message. However, an Event Message may only be appropriate the first time a new Src IP address

is seen. After that no events should be generated for that address. However other new addresses should still generate Event Messages.

The Event Criteria attribute defines the frequency with which events should be generated. If the Event Criteria is set to 'One_Time', only one event will ever be generated for that EventHdlrElement when the first match occurs, irrespective of the number of subsequent matches. If the Event Criteria is set to 'Every_Time', each match will result in an Event Message. A hybrid case is defined called 'On_Change'. This option allows a subset of the filter attributes to be required for matching and a different set of attributes that must from a unique n-tuple in order to generate an Event Message. See Section 3.2.4 for more details of the behavior of the 'On_Change' attribute.

The EventHdlrHandleScope is optional. If it is not specified, it's behavioral rules are taken from the EventHdlrEventScope objects associated with the relevant EventHdlrElement. In other words, the criteria for generating Request Handles will be identical to the criteria for generating Event Messages when the EventHdlrHandleScope is not explicitly specified.

The attributes of the EventHdlrElement class are:

eventHdlrElementId (InstanceId)
identifies the object

eventHdlrElementEventCriteria (Unsigned32)
Indicates when an event is generated. Valid options are 'one_time', 'every_time' and 'on_change'. This attribute allows event Handlers to distinguish one time events (ignore after the first match) from recurring events (generate an event every time a match occurs). An enum type is also define to specify that a new event should be generated when a specific set of fields change. This is important for protocols like RSVP because messages are sent both to demonstrate that the reservation is active and to notify hops of changes to reservations. Since only changes need to propagate to the PDP, the 'on_change' option indicates that those events should be generated selectively.

eventHdlrElementGrpId (TagId)
Group identifier. All instances with the same group identifier belong to one group and can be referenced collectively from an eventHandler instance.

eventHdlrElementEventScope (TagReferenceId)
Identifies a group of eventHdlrEventScope entries associated

with this eventHdlrElement instance.

eventHdlrElementHandleScope TagReferenceId)
Identifies a group of eventHdlrHandleScope entries associated with this eventHdlrElement instance. This is an optional attribute.

eventHdlrElementContext (TagReferenceId)

Identifies a list of ContextDataTable entries associated with this eventHdlrElement instance.

eventHdlrElementMatchNext (Prid)
The data path for traffic in scope.

3.1.4.3. EventHdlrEventScope class

This PRC defines the scope of an event handler element using references to filters defined in the Framework PIB or in some other PIBs. These filters may describe specific protocol properties for which events need to be generated. These filter references are grouped using a TagId, and this group is then referenced from the eventHdlrElement PRC.

Whenever traffic arrives at the EventHandler for which an Event Message has not already been generated, it is compared against the FilterEntry objects of the EventHdlrEventScope objects referenced by the EventHdlrElement. If it matches the criteria specified in all of the FilterEntry objects, a new Event Message is generated and sent to the PDP. The value of EventHdlrElement's EventCriteria attribute in conjunction with the value of the Event Scope class's ChangeFlag attribute determine whether an Event Message will be generated.

For example, if a FilterEntry object specifies SRC IP address (10.20.0.0) and SRC IP Mask (FF.FF.0.0) and the EventCriteria is set to One_Time, the first address in the range of 10.20.0.0 and 10.20.255.255 will generate an event and no events will follow. If the EventCriteria is set to Every_Time for the same attribute settings, each time a packet contains an IP address within the range an Event Message will be generated. If the EventCriteria is set to On_Change and the eventHdlrEventScopeChangeFlag is set to True, each new IP address within the range 10.20.0.0 and 10.20.255.255 will trigger a new Event Message. However, as soon as a specific Src IP address like 10.20.15.109 has generated an Event Message, that specific address will no longer generate an event. If the EventCriteria is set to On_Change and the eventHdlrEventScopeChangeFlag is set to False for the example address range, than the eventHdlrEventScope instance with the ChangeFlag set to True will determine uniqueness. In this scenario,

the address range is acting as a strict filter that must be met without regard to which address in the range is responsible or whether that address has been seen previously.

When multiple fields are specified for the filter and the ChangeFlag is set to true, each unique combination of field values generates an event. For example, if the SRC IP is assigned a range of 10.10.10.224 to 10.10.10.255 and DST Ports 80 to 90 then 10.10.10.240+80, 10.10.10.240+81, and 10.10.10.250+80 would all generate separate events. The combination of supporting multiple filters and being able to control precedence allows for the construction of both lists (10.10.x.x and 10.15.x.x) and combinations of disjoint headers in a single match criteria (any

combination of 10.10.x.x and VLANs 100 to 120). See Section 4.6 for a detailed example of filter construction.

The attributes of this class are:

eventHdlrEventScopeId (InstanceId)
Identifies this object

eventHdlrEventScopeGroup (TagId)
Contains the tag by which the EventHdlrElement references this object. This is the means by which a list of filters can be associated with an EventHandler.

eventHdlrEventScopeFilter (PRID)
Points to a FilterEntry object (as defined in the Framework PIB) that specifies the filter for this EventHdlrEventScope object

eventHdlrEventScopePrecedence (Integer)
Represents the precedence of this criterion with respect to other criteria within the same group. When the precedence is unique, the instance represents an alternative criterion (an OR function). When the precedence for two or more instances of the eventHdlrEventScope class is the same, the attributes within all the instances are treated collectively as a single filter criteria.

eventHdlrEventScopeChangeFlag (TruthValue)
Boolean value, if set to 'true' indicates that a new event should be generated if any of the assigned fields in the associated filter change.

3.1.4.4. EventHdlrHandleScope class

This PRC defines the scope of Request Handles generated by the PEP

due to events caught by the Event Handler Element. Each instance of this PRC references filters defined in the Framework PIB or some other signaling-protocol specific filter PRCs. These filters may describe specific protocol properties to which this Event Handler is sensitive. Essentially this table defines when a new COPS Request Handles must be created by the PEP based on protocol properties. The Event Handler may be set up to be sensitive to specific field values and/or the uniqueness of a set of values considered together. This accommodates various behaviors of signaling protocols. These filter references are grouped using a TagId, and this group is then referenced from the eventHdlrElement PRC via the eventHdlrElementHandleScope TagReference.

The behavior of the EventHdlrHandleScope class is identical to the behavior of the EventHdlrEventScope. The only difference is the EventHdlrEventScope determines when new Events are created and the EventHdlrHandleScope determines when new COPS Request Handles are created. It is important to note that the attributes determining

when a new Handle is created MUST be a subset of the filter attributes and filter values specified for the EventHdlrEventScope. The reason for this is that an Event Message MUST use one of the available Request Handles to notify the PDP of an Event. If the attributes and values used in the EventHdlrEventScope are not a superset of the attributes and values EventHdlrHandleScope, then there may be no valid Handle over which the Event Message can be sent to the PDP.

The EventHdlrHandleScope is optional. If it is not specified, it's behavioral rules are taken from the EventHdlrEventScope objects associated with the relevant EventHdlrElement. In other words, the criteria for generating Request Handles will be identical to the criteria for generating Event Messages when the EventHdlrHandleScope is not explicitly specified.

See Sections 3.1.2 and 3.2.4 for more details on the operational behavior of this class.

eventHdlrHandleScopeId (InstanceId)
An arbitrary integer index that uniquely identifies an instance of the eventHdlrHandleScopeTable class.

eventHdlrHandleScopeGroup (TagId)
Represents the binding between the eventHdlrElementEntry and the eventHdlrHandleScope entries. A group of eventHdlrHandleScope entries constitutes the criteria for defining the scope of the Handles generated.

eventHdlrHandleScopeFilter (Prid)
Pointer to a filter to be used as the criteria.

eventHdlrHandleScopePrecedence (INTEGER)
Represents the precedence of this criterion with respect to other criteria within the same group. When the precedence is unique, the instance represents an alternative criteria (an ORing function). When the precedence for two or more instances of the eventHdlrHandleScope class is the same, the attributes within all the instances are treated collectively as a single filter criteria.

eventHdlrHandleScopeChangeFlag (TruthValue)
Boolean value, if set to 'true' indicates that a new Handle should be generated if any of the assigned fields in the associated filter change.

3.1.4.5. ContextData class

This PRC specifies the context information to send to the PDP when an event is caught. The context information to send is described in terms of the PRC data types to include in the request, the level of encapsulated data and the interface information for that request.

The attributes of this class are:

ContextDataId (InstanceId)
Identifies this object

ContextDataGroup (TagId)
Defines the grouping of contextData instances that are applicable to a given eventHdlrElement.

ContextDataIfElement (PrcIdentifier)
The OID of a class whose instance is to be included with the PEP request or event-specific ContextData Response.

ContextDataEncapsulation (Integer)
This attribute allows one to distinguish between inner and outer headers when there are multiple encapsulated headers of the same type in a packet.

A value of:
0 means all headers,
positive number 'n' means the 'n'th header starting from the outermost,
negative number 'n' means the 'n'th header starting from the innermost.

3.2. Event Handling

As soon as the PEP detects a situation described by an event handler, it must trigger an event. If an event is triggered, the PEP must send a message to the PDP that installs the event handler. This section describes how this message looks like.

3.2.1. Functional Description

The event message that the PEP needs to send to the PDP is a common COPS-PR request message, containing exactly one event data structure, and optionally zero or more context data classes, depending on which eventHandlerElement triggered the event. Context Data classes are described in chapter 4.

An event message typically represents an access request of a client, the decision for which the PEP outsources to the PDP. See Figure 3.3.

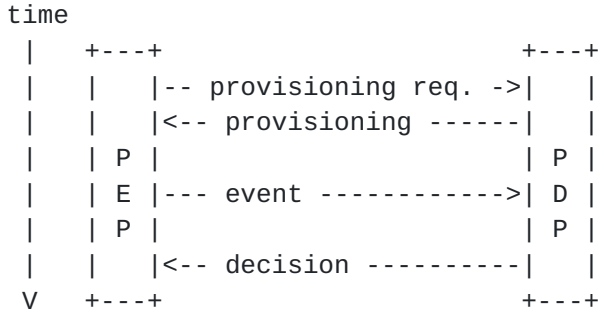


Figure 3.3: Simple message sequence for provisioning and events.

3.2.2. COPS Client Handle

The event and the decision are associated with each other using the COPS Client Handle, which is described in section 2.2.1 of RFC 2748. Each COPS message contains this Client Handle, which serves as an identifier to match request and response, and can later be used to remove an earlier install decision.

Because the COPS Client Handle serves as a connection between the request and the decision, there must not be more then one outstanding COPS request with the same Client Handle. It is possible to have multiple outstanding COPS requests, as long as their Client Handle is different.

For most applications, the PEP will generate a new unique COPS Client Handle for each event. However, in some situation it can be

useful if the same client handle is used for multiple events. This implies that the second even can only be sent after the PDP sent a response for the first event. The PDP can explicitly specify when the PEP must use a new COPS Client Handle, by using the eventHdlrElementHandleScope. This should point to a scope, similar to the eventHdlrElementEventScope. Only when the criteria in the eventHdlrElementHandleScope matches, the PEP must create a new COPS Client Handle.

3.2.3. DiffServ element

In case the Event Handler is part of a DiffServ model, the EventHandler acts as a Classifying DiffServ element. If no criteria are met, the ingress traffic for this element is forwarded to the DiffServ element specified by the eventHandlerNonMatchNext attribute in the EventHandler instance. If a criteria is met, traffic belonging to this ingress dataflow is dropped (or forwarded to the PDP, as is shown in chapter 7), until the PDP responds to the outstanding request. If the response is affirmative, the properties of the ingress dataflow, as specified by the eventHdlrElementEventCriteria and eventHdlrEventScopeChangeFlags are stored, typically in a lookup-table, and all traffic coming from this ingress dataflow is forwarded to the DiffServ element specified by the eventHdlrElementMatchNext attribute. If the response from the PDP is negative, the authorization failed, and the PEP can just forward the traffic to the eventHandlerNonMatchNext until an event is triggered.

An affirmative reply from the PDP is defined as a COPS-PR Decision message with the command in the Decision flag object set to Install (section 2.2.6 of RFC 2748). An example of how the EventCriteria and ChangeFlags specify a filter is given in 3.1.2, while section 3.2.4 further clarifies the scopes.

3.2.4. Behavior of the Event and Handle Scope classes

The rules for interpreting Handle Scope and Event Scope are the same. One is applied to Handles and the other is applied to Events.

Some of the classes used by the Access Bind PIB are the Filter classes described in the COPS-PR Framework PIB [FWPIB]. These classes allow a PDP to specify a set of 802.1 and IP header field values that can be matched against packets. The Event Scope and Handle Scope classes can use these filter classes as well as other filter classes to define the criteria for generating an event.

Each scope class (Event Scope or Handle Scope) instance has a

precedence value associated with it. When two or more scope class instances of the same type (event vs. handle) have the same precedence number, they are considered part of the same rule. For example, table 3.1 lists a set of Event Scope class instances, their precedence values and the filter field names and values associated with each instance (FName is the field name):

Instance	Precedence	FName/Val	FName/Val	FName/Val
1*	2	W/20	X/2-4	
2*	1	A/5-6	B/15	C/10-11
3	2	W/14		Y/500-550
4	2	Q/4-9	R/92	

Table 3.1: List of Filter rules

This example would result in the following two filter expressions:
1. (A=5 or A=6) and (B=15) and (C=10 or C=11 or C=12)
2. (W=20 or W=14) and (X=2-4) and (Y=500-550) and (Q=4-9) and (R=92)

If the EventCriteria was set to 'One Time', then if either 1 or 2 is matched, one event will be generated and this particular Event Handler Element will generate no further events. Note that if matches occur but the 'One Time' event has already been generated, the Event Hander Element's MatchNext attribute may still determine what the next forwarding action is for the packet event even though no event is generated.

If the EventCriteria was set to 'Every Time', then every matching packet will cause an event. If the EventCriteria was set to 'On Change', then events will be generated the first time a unique combination of attributes is seen. Setting the ChangeFlag to 'True' in the EventScope class (denoted by the asterisks next to the Instance number in Table 3.1), identifies the set of attributes for which unique combinations of values generated new events. The ChangeFlag is applied to the attribute, not the specific instance of the filter. Therefore, even though instance 3 does not have the ChangeFlag set, the values for attribute W specified in instance 3

will be treated as if the ChangeFlag was set for that attribute, as per example 8 below.

Continuing the example above the following table shows a stream of packets and whether an event will be generated.

1. A=5, B=19, C=10	No Event (B did not match)
2. A=5, B=15, C=10	Event (Unique pairing of A & C)
3. A=6, B=15, C=11	Event (Unique pairing of A & C)

4.	W=20, X=2, Y=500, Q=4, R=92	Event (Unique pairing of W & X)
5.	W=20, X=2, Y=505, Q=9, R=92	No Event (Already have a matched for W & X)
6.	A=5, B=15, C=11	Event (Unique pairing of A & C)
7.	A=5, B=15, C=10	No Event (Already have a matched for A & C)
8.	W=20, X=3, Y=502, Q=7, R=95	No Event (no match on R)
9.	W=14, X=2, Y=500, Q=4, R=92	Event (Unique pairing of W & X)

3.2.5. Context Data Entries

Section 3.1.3 described the ContextData class and how it is used to provision the set of event-specific information elements that must be included with each Event Message. This section provides an overview of the format of the actual information elements.

Each Context Data Entry is organized to logically describe a layer or grouping of attributes. The downside of this strategy is that when a specific entry is requested, all the fields in the entry must be filled before the entry can be sent to the PDP. This is a compromise between forcing the PDP to describe all the fields explicitly and making the PEP send all attributes of possible interest. Since a given PDP knows better what it needs to generate a decision than a PEP, the second alternative is extremely unwieldy. On the other hand, forcing the PDP to describe all the fields necessary for a given event, would create an explosion of object definitions.

In sections 3.2.6.2, there are two classes that are defined as part of the Access Bind PIB. These objects define the relevant fields of a Ethernet header and a IP header. It was determined that these headers existed in a large enough cross-section of application-specific signaling PIBs, that they belonged in the Access Bind PIB. This does not impact those application-specific signaling PIBs that don't use one or both headers. Since the PDP request only those headers relevant to each application specific event, these classes do not need to be implemented in order to meet the compliance requirements for this PIB.

3.2.6. Data Description

This section describes the behavior of all classes associated with the generation of event messages to the PDP.

3.2.6.1. Event class

Instances of this table represent events that occurred at the PEP. The events reference the event handler instance and the specific

event handler element that the event was caught by.

The attributes of this class are:

eventId (InstanceId)
Identifies this object

eventEventHdlr (ReferenceId)
This attribute allows a PEP to indicate to the PDP that this event was generated due to the referenced Event Handler. This attribute references an event handler via the indirection PRC frwkReference, since the event handler and event could potentially belong to a different PIB contexts.

eventCause (ReferenceId)
This attribute references the specific instance in a group of event Handler elements belonging to an event Handler that resulted in this event. This attribute references a specific event handler element via the indirection PRC frwkReference, since the event handler element and event could potentially belong to a different PIB contexts.

3.2.6.2. ContextData classes

This section contains examples of classes that might be referenced by the ContextData class as classes that must be included in the Event Message for various types of eventHdlrElements.

There are two kinds of ContextData classes depending on the type of PEP. Some PEPs receive traffic from many users over a shared port such as an Ethernet port. They recognize new users based on information in the headers of incoming packets. For them, the ContextData will come from packet headers. The L3HeaderData class is an example of this kind of ContextData class. Other PEPs receive traffic from one user per interface. For them, the context data will be information about the interface. The CtxtDialupInterfaceFramedProtocol class is an example of this kind of ContextData class.

3.2.6.2.1. CtxtL3Hdr class

This class specifies level three header data. This class is used to inform the PDP of the details of the IP header that caused the PEP Event Message to be generated.

The attributes of this class are:

CtxtL3HdrId (InstanceId)
identifies this object

CtxtL3HdrSrcAddrType (Enum)
specifies the type of the packet's layer 3 source address

CtxtL3HdrSrcAddr
the packet's layer 3 source address

CtxtL3HdrDstAddrType (Enum)
specifies the type of the packet's layer 3 destination address

CtxtL3HdrDstAddr
the packet's layer 3 destination address

CtxtL3HdrProtocol
the packet's protocol field

CtxtL3HdrSrcPort
the packet's source port field

CtxtL3HdrDstPort
the packet's destination port field

CtxtL3HdrDscp
the packet's Type of Service (Diffserv Code Point)field

CtxtL3HdrEcn (boolean)
Indicates whether this packet is ECN capable (True) or not (False)

CtxtL3HdrIpOpt
IP Options

CtxtL3HdrEncap
The Encap allows the PEP to indicate where this header is in relation to other IP headers found in the packet (with tunnels). This value can be either positive or negative depending on how the EventHandler (or the Session-specific Context Data request) was specified using negative or positive numbers.

A negative n means return the nth layer from the innermost header. A positive n means return the nth layer from the outermost header.

3.2.6.2.2. Ctxt802Hdr class

This class specifies IEEE 802.1 header data. This class is used to inform the PDP of the details of the 802 header that caused the PEP

Event Message to be generated.

The attributes of this class are:

Ctxt802HdrId (InstanceId)
identifies this object

Ctxt802HdrSrcAddr
the frame's source MAC address

Ctxt802HdrDstAddr
the frame's destination MAC address

Ctxt802HdrProtocol
the layer 2 frame's protocol field

Ctxt802HdrPriority
the layer 2 frame's priority field (only used if the frame is using the 802.q header extension)

Ctxt802HdrVlan
the layer 2 frame's VLAN field (only used if the frame is using the 802.q header extension)

Ctxt802HdrEncap
The Encap allows the PEP to indicate where this header is in relation to other IP headers found in the packet (with tunnels). This value can be either positive or negative depending on how the Event Handler (or the explicitly requested PDP Context Data request) was specified using negative or positive numbers.

A negative n means return the nth layer from the innermost header. A positive n means return the nth layer from the outermost header.

4. Identity Extensions PIB module

The Access Bind PIB provides a basic framework for processing PEP events. A subset of events require identity management of some type. In some cases this means that the PDP, PEP and end system are involved in some type of authentication process. An Identity Extensions PIB is provided that extends the EventHandler class and adds some Authentication Protocol specific classes. This PIB is described in detail below.

4.1. Functional Description

In the operational model for this PIB, the Authentication Server is a specific function of the PDP. The main purpose of the authentication portions of this PIB is to verify the validity of client credentials by an Authentication Server. The verification process itself may do this whilst ensuring some level of authenticity, confidentiality and integrity. Messages exchanged between a Client and Authentication Server (PDP) may remain confidential to PEP's and Proxy Servers. The message integrity may be ensured by some hashing algorithm so PEP's and Proxy's may inspect but not modify the content of authentication messages. Clients, PEP's, Proxy's and PDP's will always need some security method to ensure message authenticity.

Some authentication protocols explicitly consider proxies by allowing the payload to be carried over a variety of transports. Others depend on the termination point of the connection to explicitly proxy the authentication, when that is necessary. In order to demonstrate the general utility of this model, a variety of client authentication protocols will be considered in this document. For each protocol, the negotiation mechanism will be described and the mapping to this framework will be detailed.

4.1.1. Provisioning

The PEP will not start an authentication sequence with the client if it hasn't been told to do that. It will only do so when a specific event occurs. The PDP tells the PEP exactly when this event should be triggered. This process is called provisioning.

The provisioning starts with the initial Provisioning Request, which

is typically sent at boot time. The PEP sends up capability PRC's indicating the types of authentication it can handle. The PDP will reply by setting the following Access Bind PRC's:

a. identEventHandler (IdentityEventHandler)
b. identEventhdlrAuthProtocol
c. eventHdlrElement
d. eventHdlrEventScope
e. eventHdlrHandleScope
f. contextData

and an additional PRC instance referred to by the eventhdlrEventScopeFilter in the eventhdlrEventScope table, indicating how the signaling trigger is recognized.

In case the PDP wants the PEP to perform an authentication when an event is triggered, provisions an Identity Event Handler (identEventHandler) instead of the standard Event Handler. The Identity Event Handler has a few extra attributes in the class to allow the PDP to indicate what authentication protocols to use and whether authentication is mandatory. This is done by setting the identEventhdlrRequestAuth in the identEventHdlr to true and optionally letting the identEventHdlrAuthProtocol field reference a eventHdlrAuthProtocol tagid to indicate which authentication protocols should be used for the authentication.

As soon as the PDP has provisioned the PEP to watch for certain traffic that triggers an event, the PEP is ready to start an authentication.

4.1.2. EAP Authentication

The most significant aspect distinguishing EAP [EAP] from other authentication protocols is that EAP assumes the negotiation is between the client and the authentication server. In anticipation of the fact that the terminating point of a connection such as PPP or L2TP is not necessarily the same as the agent managing client authentication, EAP encapsulates it's negotiation process in a separate header that can be forwarded in entirety to the server. This mechanism provides extra security by preventing intermediate proxies from monitoring or managing authentication credentials.

EAP supports a number of different authentication mechanisms including MD5, TLS, and One-Time-Password authentication.

The terminology used in [EAP] differs from the terminology used in this document. In particular, the peer, as defined in section 1.2 of [EAP], is referred to as 'Client' in this document. Similarly, the

'authenticator' is called a PEP in this document and 'back-end server' is called the Authentication Server function of the PDP (or just PDP) in this document.

4.1.2.1. EAP Message sequence

The generic sequence of transmissions between the PEP and PDP has already been described in section 2. In particular, figure 2.1 gives an overview of the messages involved between the Client workstation, PEP and PDP. EAP messages are embedded in PPP packets in the communication between the Client and the PEP. In the communication between the PEP and PDP, the EAP messages are embedded in COPS Request, COPS Decision and COPS Report messages. Figure 4.1 shows how EAP may be used to retrieve credentials from the client workstation by the PDP.

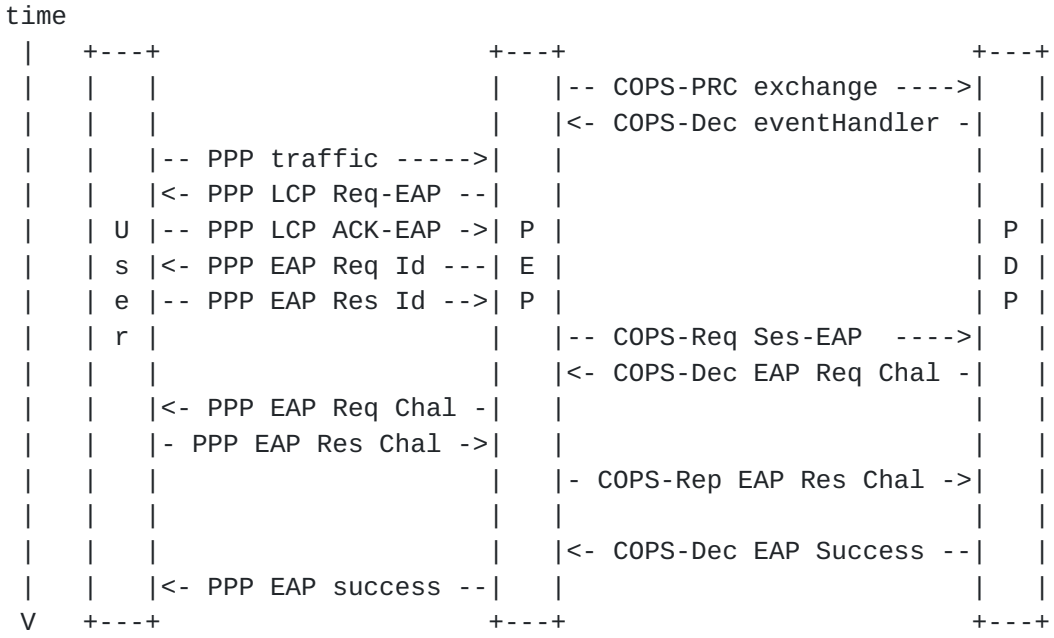


Figure 4.1: Embedding of EAP messages between the Client workstation and the PEP, and between the PEP and PDP. The EAP messages may be opaque to the PEP.

Typically, when the PEP boots up, it sends it's capabilities to the PDP in a COPS message and is than configured by the PDP with one or more datapathEventHandlers specifying the criteria for generating PEP Event Messages. The first message after this provisioning process from the PEP to the PDP is a new Event Message. The PEP sends a COPS request to the PDP containing a new instance of the Event table. The eventEventHdlr attribute in the Event table entry

is a ReferenceId that points to a dpeventHandler entry indicating (by means of the dpEventHdlrAuthProtocol) that EAP is a valid protocol to use for this Event. Also, the eventCause attribute in the Event table entry is a ReferenceId that points to an eventhdlrElement indication of which Filter (by means of the eventhdlrEventScope) triggered the event.

All EAP messages necessary to complete the authentication process will be forwarded to the PDP. All of the negotiation occurs between the Client and the PDP and should, except for the EAP message code field, not be examined by the PEP. In order to support multiple EAP protocols, this PIB supports a generic EAP request class and EAP response class. Each class has a single string attribute (authEapReqExtSpecific and authEapRespExtSpecific, respectively) within which the entire EAP message is passed.

Although figure 4.1 shows two EAP messages going from the PDP to the Client and two EAP messages being returned from the client to the PDP, the actual number of messages exchanged can be any amount. The PDP may continue to retrieve additional credentials from the client for as long as it wishes. As soon as the PDP has all the necessary

credentials from the client, the PDP may continue to provision the PEP with policies. This is action is not shown in figure 4.1.

The PDP should end the EAP negotiation with an EAP Success or an EAP Failure message. If the PDP sends a EAP Success, the PEP must from then on use the matchNext Prid to determine the next processing filter for data defined by the values described using the eventhdlrEventScope.

4.1.3. PAP Authentication

PAP (Password Authentication Protocol), as described in section 2 of [AUTH] is a very simple authentication mechanism used over PPP. It is not considered to be a secure mechanism, since it sends passwords over the wire in clear text format. However, where one-time passwords are used, this security concern is mitigated. It is described here nonetheless for illustration purposes and because it may still be used among ISPs, or in situations where another layer already performs encryption for security.

The terminology used in [AUTH] differs from the terminology used in this document. In particular, the peer as defined in section 1.2 of [AUTH] is referred to as 'Client' in this document. Similar, the 'authenticator' is called PEP in this document.

4.1.3.1. PAP Connection sequence

Figure 4.2 shows how PAP may be used to retrieve credentials from the client workstation by the PDP.

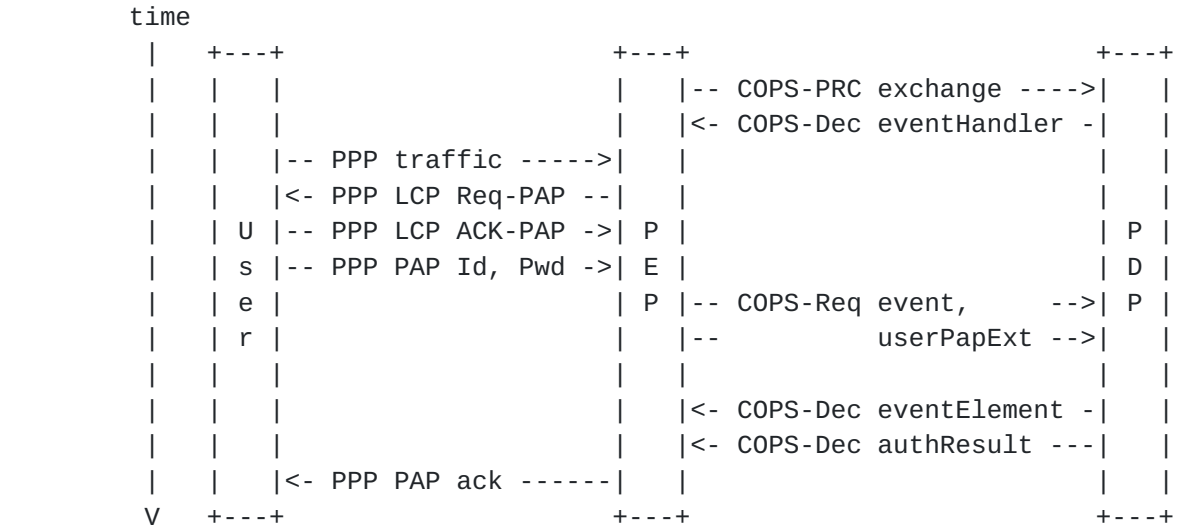


Figure 4.2: Embedding of PAP messages between the Client workstation and the PEP, and between the PEP and PDP.

When the dpEventHandler has been configured to require authentication, a PEP Event message will not be generated until

after a minimal set of credentials have been negotiated with the client. For PAP, this means that a PEP Event Message will not be generated until after the authRealm and authUsername have been determined. This means that that the PEP must receive a PAP Identity message before it can send the PEP Event Message.

The Client will send the Identity and Password to the PEP. The PEP will embed the password into the userPapExt datastructure and send this to the PDP. Since this datastructure inherits the fields of the userAuthExt data structure and the extAuth data structure, it will also contain the PAP identity attribute inserted into the authUsername attribute of this Instance.

The first connection from the PEP to the PDP is an alert that an event was triggered. The PEP sends an Event Message over COPS to the PDP containing a new instance of the Event table. The eventEventHdlr attribute in the Event table entry is a ReferenceId that points to a dpeventHandler entry indicating (by means of the dpEventHdlrAuthProtocol) that PAP is a valid protocol to use for this Event. Also, the eventCause attribute in the Event table entry is a ReferenceId that points to an eventhdlrElement indication which

Filter (by means of the eventhdlrEventScope) did trigger the event. Along with this new instance of the Event table, the PEP must also send an instance of the AuthPapExt table.

Besides these required instances, the PEP might have been configured by the PDP to sent additional information about the client to the PDP. For example in the case of a dialup connection between the Client and the PEP, the PDP might specify using a contextData instance that the PEP should also sent an instance of a ctxtDialupInterface.

The PDP performs the PAP authentication. When the authentication is complete and the PDP is ready to authorize the event, the PDP optionally provisions the PEP with policies. This sequence of messages should terminate with a PDP Provisioning Decision (a COPS-PR Decision message). The PDP Provisioning Decision contains an instance of the AuthExtResult table with the authExtAuthSuccess set to either TRUE or FALSE. The PEP must upon receipt of this COPS Decision message, send PAP ACK or NACK message to the client. Also, if the authExtAuthSuccess attribute was true, then the PEP should keep track of this particular data, defined by the unique values of the fields specified using the eventhdlrEventScope.

4.1.4. CHAP Authentication

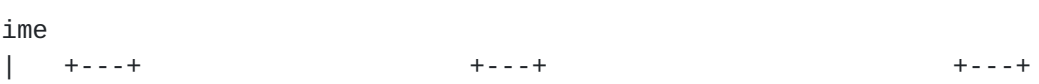
CHAP (Challenge Authentication Protocol) [CHAP] is a strong authentication mechanism, which eliminates the need to send passwords in the clear, like PAP does. With CHAP, the Authenticator generates a challenge key, sends it to the Peer (Client) and the client responds with a cryptographically hashed response that depends upon the Challenge and a secret key. The PDP checks the

secret key by performing the same encryption and comparing the results.

The terminology used in [CHAP] differs from the terminology used in this document. In particular, the peer as defined in section 1.2 of [CHAP] is referred to as 'Client' in this document. Similar, the 'authenticator' is called PEP in this document.

4.1.4.1. CHAP Connection sequence

Figure 4.3 shows how CHAP may be used to retrieve credentials from the client workstation by the PDP.



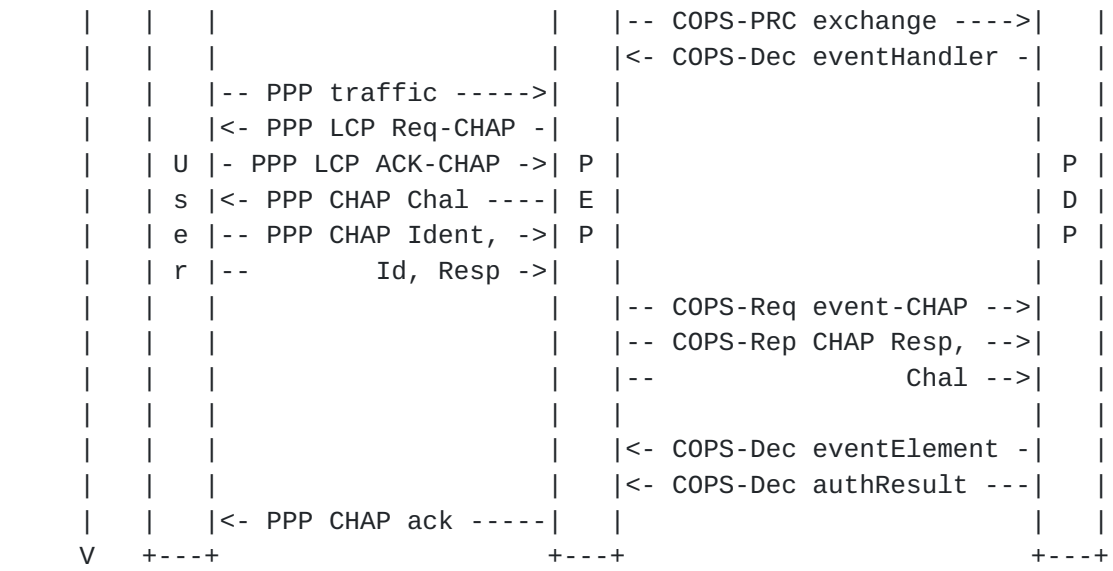


Figure 4.3: Embedding of CHAP messages between the Client workstation and the PEP, and between the PEP and PDP.

As soon as the PEP finished negotiating CHAP as the Authentication protocol, it generates a challenge itself, and sends this to the Client. The client will respond to this authentication request by sending his or her identity, an identifier and the response. The response is a cryptographically encrypted hash based on the challenge and secret key (password).

The identifier is only used to keep track of CHAP messages, and needs to be used by the PEP to recover the associated challenge.

The first connection from the PEP to the PDP is a notice of a new Event. The PEP sends an Event Message to the PDP containing a new instance of the Event Table. The eventEventHdlr attribute in the Event table entry is a ReferenceId that points to a dpeventHandler entry indicating (by means of the dpEventHdlrAuthProtocol) that CHAP is a valid protocol to use for this Event. Also, the eventCause

attribute in the Event table entry is a ReferenceId that points to an eventhdlrElement indication of which Filter (by means of the eventhdlrEventScope) did trigger the event. Along with this new instance of the Event table, the PEP must also send an instance of the AuthChapExt table.

Note that having the PEP issue the challenge allows the PEP to perpetrate fraud by issuing a replayed request (assuming that the PEP and PDP are in different domains). The only guard against this is for the PDP to check that multiple authentication requests for

the same client have unique challenges. This may be slow. PDP and Authentication server developers who feel this is a security issue may want to use EAP-MD5 authentication rather then CHAP authentication, since EAP-MD5 addresses this problem by letting the PDP generate the challenge.

Besides these required instances, the PEP might have been configured by the PDP to send additional information about the client to the PDP. For example in the case of a dialup connection between the Client and the PEP, the PDP might specify using a contextData instance that the PEP should also sent an instance of a ctxtDialupInterface.

The PDP performs the CHAP authentication. When the authentication is complete and the PDP is ready to authorize the client, the PDP may choose to provision the PEP with policies for this client, which was probably the intention of starting this authentication process in the first place. This sequence of messages should terminate with a PDP Provisioning Decision (a COPS-PR Decision message). The PDP Provisioning Decision contains an instance of the AuthExtResult table with the authExtAuthSuccess set to either TRUE or FALSE. The PEP must upon receipt of this COPS Decision message, send PAP ACK or NACK message to the client. Also, if the authExtAuthSuccess attribute was true, then the PEP should keep track of this particular data, defined by the unique values of the fields specified using the eventhdlrEventScope.

4.2. Data Description

This section describes each of the classes defined in the Identity Extension PIB module.

4.2.1. IdentityEventHdlr Class

This PRC is an extension of the EventHandler PRC. This extension illustrates the use of the EventHandler PRC concept for authentication usage. Instances of this PRC are provisioned by the PDP on the PEP to catch specific events that require authentication processing. This PRC references a group of eventHdlrAuthProtocol instances that define a set of Authentication mechanisms to use if an access event is caught by this event Handler. From its base class (Event Handler) this PRC also references a group of eventHdlrElement PRIs that contain the scope of the access event and specify the context data to send to the PDP when an access event is caught.

The attributes of this class are:

identityEventHdlrRequestAuth (TruthValue)

Boolean flag, if set to 'true' requires authentication data

to be sent in the Event Message sent to the PDP.

identityEventHdlrAuthProtocol (TagReferenceId)
References a group of identityHdlrAuthProtocol instances, each of which specifies an authentication mechanism.

4.2.2. EventHdlrAuthProtocol class

This class allows a PDP to configure the set of authentication mechanisms that are allowed for users or devices that must authenticate in order to have access control policies assigned to them.

The attributes of this class are:

eventHdlrAuthProtocolId (InstanceId)
Identifies this object

eventHdlrAuthProtocolGroup (TagId)
Represents a binding between an datapathEventHdlrTable instance and a list of eventHdlrAuthProtocolTable instances.

eventHdlrAuthProtocolAuthMechanism (Enum)
Specifies an authentication mechanism to be used in Event Messages triggered by the EventHandler referencing this EventHdlrAuthProtocol object. The value is from an enumerated list initially consisting of (PAP, CHAP, EAP-MD5, and EAP-TLS)

4.2.3. AuthExt class

This is an abstract PRC. This PRC can be extended by authentication PRCs that contain attributes specific to that authentication protocol. An instance of the extended class is created by the PEP and sent to the PDP. The PDP may send information back to the PEP or may use the information to authenticate the PEP's Event Message. This PRC itself should not be instantiated.

Typically this class and it's subclasses are included as part of an event message containing the Event class (Section 3.2.6.1).

The data in this class is passed between the PDP and the client with little or no involvement of the PEP except to forward it in the appropriate AuthExt class instance. The PEP is not meant to store AuthExt objects. As such, this class, along with all its extending classes, is meant to be 'transient'. Its instances are temporary and are deleted by the PEP after a certain time or event. The PDP, in its decisions, must not refer to instances of this class that are

sent by the PEP in its requests. Likewise, the PEP must not refer to instances sent by the PDP. Also, since instances are deleted, it is possible for InstanceIds to be reused.

The AuthExt class is extended for each authentication mechanism supported. As a base class, it is never instantiated.

The attributes of this class are:

AuthExtId (InstanceId)
identifies this object

4.2.4. UserAuthExt class

This is a concrete PRC used to contain user authentication fields. This PRC extends the base PRC authExtEntry.

The attributes of this class are:

userAuthExtRealm (OCTET STRING)
The user realm octet string.

userAuthExtUsername (OCTET STRING)
The Username octet string.

4.2.5. AuthExtResult class

All authentication message sequences conclude with an authentication result message sent from the PDP to the PEP. This message is usually accompanied by one or more provisioning decisions associated with the authenticated identity. The AuthExtResult class extends the AuthExt class. It contains the Authentication result Boolean flag.

The attributes that this class adds to the base class are:

AuthExtSuccessful (TruthValue)
A Boolean flag set to true if the authentication (via CHAP or PAP) was successful.

4.2.6. AuthEapReqExt and AuthEapRespExt classes

The EAP messages are embedded in COPS by sending an instance of the authEapReqExt or authEapRespExt table, which each have an attribute (Specific) to encapsulate the appropriate EAP messages necessary for the authentication mechanism. The authEapReqExt table is owned and managed by the PEP, while the authEapReqExt table is owned and managed by the PDP. Put another way, the PDP generates authEapReqExt instances that it sends in Decision messages and the PEP generates authEapRespExt instances that it sends in Report messages. Since neither the PEP nor the PDP needs to maintain the messages

permanently, the same instance of each class is used when more than one exchange is required in each direction.

Since both AuthEapReqExt and AuthEapRespExt are extensions of the AuthExt class, they both inherit the attributes of AuthExt.

AuthEapReXXExt table attributes	Attribute type
authExtId	InstanceId
authExtEvent	ReferenceId
authEapReXXExtSpecific	OCTET STRING

Figure 4.4: Data elements in AuthEapReqExt and AuthEapRespExt tables

The AuthEapReXXExt class contains three attributes. The instanceId is used to uniquely define the instance in the table. However, since EAP messages are meant to be opaque, they should not be referenced. Because the purpose of the classes is to carry EAP messages and each message is transient instances of these tables are temporary and should not be referred to. The Event attribute points to the Event table entry for which EAP is being negotiated. The format of EAP packages being passed by the AuthEapReXXExt classes is described in [EAP].

4.2.7. AuthPapExtEntry class

The PAP information is embedded in the PEP Event Message by sending an instance of the authPapExt table. Since the authPapExt table is an extension of the userAuthExt table, which is an extansion of the authExt table, the authPapExt inherits the attributes of these tables.

AuthPapExt table attributes	Attribute type
authExtId	InstanceId
authExtEvent	ReferenceId
authRealm	OCTET STRING
authUsername	OCTET STRING
authPapExtPwd	OCTET STRING

Figure 4.5: Atributes of the AuthPapExt table.

The AuthPapExt contains five attributes. The instanceId is used to uniquely define the instance in the table. However, since the PAP password is sent to the PDP once and is needed by neither the PDP nor the PEP after the client is authenticated, the instance should

not be referenced after it is used the first time. The Event attribute points to the Event table entry for which PAP is being negotiated.

The result of the authentication for PAP is sent in the AuthExtResult table. Since the authExtResult table is an extension of the AuthExt table, it inherits the attributes of AuthExt.

AuthExtResult table attributes	Attribute type
authExtId	InstanceId
authExtEvent	ReferenceId
authExtAuthSuccess	Truth Value

Figure 4.6: Atributes of the AuthExtResult table.

The AuthExtResult is sent by the PDP to the PEP. If the authentication was successful and the PEP should from now on use the matchNext Prid to determine the next processing filter (the next component down the internal datapath in the PEP) for all traffic defined by the values of the parameters as set in the eventhdlrHandlerScope.

4.2.8. AuthChapExtEntry class

The CHAP information is embedded in the Event Message by sending an instance of the authChapExt table. Since the authChapExt table is an extension of the userAuthExt table, which is an extansion of the authExt table, the authChapExt table inherits the attributes of these tables.

AuthChapExt table attributes	Attribute type
authExtId	InstanceId
authExtEvent	ReferenceId
authRealm	OCTET STRING
authUsername	OCTET STRING
authChapExtId	Unsigned32
authChapExtChal	OCTET STRING
authChapExtResp	OCTET STRING

Figure 4.7: Data elements of the AuthChapExtEntry datastructure.

The AuthChapExtId is generated by the PEP. The ID value is sent to

the client. When the client endpoint (Peer) generates a CHAP response it includes the same ID, and the ID is then included in this attribute when it is sent to the PDP.

The AuthChapExtEntry contains seven attributes. The instanceId is used to uniquely define the instance in the table. However, since the CHAP information is sent to the PDP once and is needed by neither the PDP nor the PEP after the client is authenticated, the instance should not be referenced after it is used the first time. The Event attribute points to the Event table entry for which PAP is being negotiated.

The authChapExtChal attribute is the challenge generated by the PEP. The PDP may check the challenge to see if it is different from challenges used earlier. This to provides an increased level of security. The Response and the Id is taken from the CHAP message sent by the client and put into the AuthChapExtEntry by the PEP.

The authChapExtResp is calculated by the client and forwarded by the PEP to the PDP.

5. Signal Handling

5.1 Functional Description

5.2 Data Description

6. Programmatic Interface Between Signal and Event Handling

The programmatic interface between signal and event handling (Access Bind API) allows flexible implementation of signal handling mechanisms loosely coupled to the event handling capability provided by the Access Bind. This flexibility allows multiple different types of signaling technology be used and integrated using Access Bind.

6.1. Functional Description

The Access Bind API consist of multiple phases of operation:
The notification and specification of Event Handling needs by the signal handling mechanism. We call this signal handling Registration with event handling.
The result of the Registration indicated by event handling of Access Bind.
The indication by signal handling for event generation.
The indication of event result by event handling.
Signal handling state information notification.
Registration session termination.

Each of the above phases uses PRCs defined in the Access Bind Framework PIB and COPS-PR functionality.

When the signaling mechanism initializes or realize it requires event handling assistance, it shall register with event handling using the EventHdlrReg() method. This acts as an indication to event handling what services signal handling need and can also indicate the capability of signal handling if necessary. These service requirement and capability are indicated using PRC definitions. Event handling uses these requirement and signal handling capability, together with event handling capability to generate the initial COPS REQ message that carry instances of the

capability and limitation PRC as they are defined in [FRWKPIB].

The Registration result is generated based on the COPS DEC message received by event handling. This sets up and creates corresponding entries in eventHandlerTable, eventHdlrElementTable, eventHdlrEventScopeTable, eventHdlrHandleScopeTable, and contextDataTable. Some of the content of these entries may be from information provided by signal handling during registration. Event handling shall use EventHdlrRegRsp() to provide the registration result to signal handling. The amount of information included in with the registration result is based on the functionality of the signal handling mechanism and may include provisioning information for the signal handling mechanism.

After the registration is completed, the event handling is ready for event requests from signal handling. This is provided by the EventGenReq() method. The signal handling mechanism must provide the corresponding information necessary for the event generation. This includes, in the form of PRIs, information necessary for event

handling and PDP to determine the event result. This may include information on what the event result signal handling is expecting for the specific event. The EventGenReq() shall cause event handling to create an entry in eventTable and send a COPS REQ message. Signal handling also have the responsibility for indicating how the COPS Request Handles are used and re-used for specific EventGenReq() invocation.

Upon receive of COPS DEC message from PDP, event handling uses EventRslt() to communicate to signal handling the event result. The result can be as simple as a yes/no response or as complex as indicating in detail everything signal handling need to do to process the event result. The degree of complexity is determined by the signal handling mechanism. Event handling and this API in Access Bind are flexible to handle this spectrum of complexity. The event result can also affect event handling for integration of the dynamic nature of event and the possibly more static setup of data path usage.

Signal handling is required to indicate to event handling its response for accepting/implementing the event result by using the EventStateUpd() method. This information provided by signal handling is again dependent on the signal handling mechanism and can be simple or complex. The EventStateUpd() method is also used by signal handling to indicate any asynchronous state changes or usage indications.

When either signal handling or event handling wants to end the event

handler session registration, the EventHdlrRegTerm() method is used.

6.2. Method Description

EventHdlrReg(), called by signal handling
Indicates to event handling the services needed by signal handling.

Parameters may include:
Signal Handling Type (COPS Client-Type).
Signal Handling Capability/Limitation (as PIB PRCs).

EventHdlrRegRsp(), called by event handling
Indicates the success/failure of the registration. For successful registration, may provision the signal handler for event generation and other signal handling tasks.

Parameters may include:
Registration Success/Failure
Signal handler provisioning PRCs

EventGenReq(), called by signal handling.

Event generation request. Used to request for service from event handling. Triggers the sending of COPS REQ message to PDP.

Parameters may include:
EventState, indicate create new or reuse existing event state (COPS Handle). When reuse, ID for existing event state.

AssociationInfo, the association for this event state. This can be interface or service identification.

SignalInfo, the signal handling specific information. The signaling information needed for correct event handling.

EventRslt(), called by event handling Event result.
Containing the information in a COPS DEC message for handling the event.

Parameters may include:
EventState, indicate for which event state (COPS Handle) this result is for.
ResultInfo, contains signal handling specific information.

EventStateUpd(), called by signal handling
For Event State update due to signal handling info changes, including to indicate the Event Result have been implemented. This may also be used for usage feedback purposes. This triggers the generation of COPS Report message.

Parameters may include:
EventState, indicate for which event state (COPS Handle) this update is for.
UpdateInfo, the state information being updated.

EventStateTerm(), called by either signal or event handling
Event state termination.
Removes the event state.

Parameters may include:
EventState, indicate which event state to terminate.
TermInfo, the termination information, e.g. reason code.

EventHdlrRegTerm(), called by either signal or event handling
Registration termination.
Ends the current registration.

6.3. Access Bind API Example

As an example, using RSVP as the signaling protocol for an unicast flow between sender S1 and receiver R1 through a RSVP and Policy aware router as in Figure 6.1.

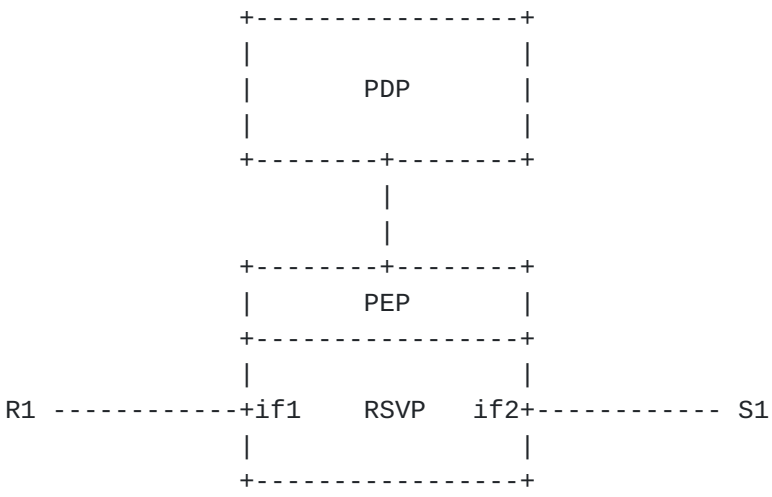


Figure 6.1: Signal/Event Handling API _ RSVP Example

When the signaling protocol is RSVP, the API is used as follows:

1. EventHdlrReg() called by Signal Handling

   RSVP Signal Handling indicates to the generic Event Handling its need to communicate to the PDP via COPS-PR.

   This will have Event Handling generate:
   COPS REQ msg containing PRCs:
   Framework PIB Capability/Limitation PRC for
   - Event Handler (eventHandler/HdlrElement/etc)
   - Signal Handler specific trigger filters
   - Signal Handler provisioning PRCs
   That indicate what signal handling needs from PDP

2. EventHdlrRegRsp() called by Event Handling

   Indication of success/failure of Event Handler initialization.

   This is caused by the event handler receiving:
   A COPS DEC msg, and may include instances of PRCs that:
   - Provisions the Signal Handler for event generation. (i.e. eventHandler/HdlrElement/HdlrEventScope and RSVP Filter PRC instances)
   - Provisions the Signal Handler if supported. (i.e. enable RSVP multicast, refresh timers)

3. EventGenReq() called by Signal Handling

   Event Generation Request.

   For RSVP, this is used as indication of PSB or RSB state changes, when receiving or about to send RSVP messages, requiring Policy Decisions.

   Parameters:

   EventState, indicate create new or reuse existing event state (COPS Handle). When reuse, ID for existing event state.
   AssociationInfo, the association for this event state. For RSVP this is the In/Out Interface identification.
   SignalInfo, the signal handling specific information. For RSVP, this contains PRCs for In/Out/Allocate Context, Path/ResV/PathErr/ResVErr signal message type, and the RSVP objects themselves (as carried by the COPS-RSVP ClientSI).

   This causes the sending of a COPS REQ message.

4. EventRslt() called by Event Handling
   Event Result.
   For RSVP, this is the decision from the PDP for a previously
   generated event.

   Parameters:
      EventState, indicate for which event state (COPS Handle)
        this result is for.
      ResultInfo, contains the signal handling specific
        information. For RSVP, this contains PRCs for
        In/Out/Allocate Context, Path/ResV/PathErr/ResVErr signal
        message type, and the RSVP objects themselves (as carried
        by the COPS-RSVP ClientSI).

5. EventStateUpd() called by Signal Handling
   For Event State update due to signal handling info changes. To
   indicate the Event Result have been implemented.
   For RSVP, this is used for generating the COPS Report message.

   Parameters:
      EventState, indicate for which event state (COPS Handle)
        this update is for.
      UpdateInfo, the state information being updated. For RSVP,
        used to indicate report type.

6. EventTerm() called by either Signal or Event Handling
   Event Terminiation.
     Removes the Event State.

   Parameters:
      EventState, indicate which event state is to be terminated.
      TermInfo, termination information. For RSVP, reason code.

7. Message Types

All PIB messages have some form of transactional semantics. Most all transactions consist of requests and responses. Typical provisioning PIBs have the PDP sending a provisioning decision to the PEP and the PEP responding with a success or fail. This PIB uses this paradigm in some cases, but it also uses a paradigm where the PEP initiates an event and the PDP responds with a success or fail. The specific

use of this paradigm is with the PEP Access Event Message, which is triggered by a PEP event and requires authentication success or failure semantics as part of the Provisioning Decision. This section discusses both paradigms and how the various classes defined in sections 3 and 4 are combined to form the various message interactions described in sections 2, 3 and 4.

Each message description in this section will include the purpose of the message, the COPS-PR message type, the direction of the message, and the class instances typically found in the message.

7.1. Event Handler Provisioning Decisions

The Event Handler Provisioning Decision message is a COPS-PR Decision message used by the PDP to provision each Event Handler in the PEP. It is likely to be a piece of a larger Decision message that provisions other data path components that occur either before or after the Event Handler in the data path. However, it could also be sent as a part of unrelated data path or other provisioning components. Event Handler provisioning typically includes the EventHandler class, the EventHdlrElement class, the EventHdlrEventScope class, often the EventHdlrHandleScope class and the ContextData class. An optional set of EventHdlrAuthProtocol class instances may be sent if an IdentityEventHdlr object is set up for Access Event Messages.

Because the EventHdlrElement, ContextData, EventHdlrEventScope, and the EventHdlrHandleScope classes all describe configuration details of the EventHandler, any of these class instances may be shared by multiple EventHandler instances. Therefore, in many cases, an EventHandler Provisioning Decision will contain only an EventHandler that references instances of the other classes defined in previous Provisioning Decisions. In addition, these classes can also be provisioned individually in anticipation of being applied to an EventHandler. However, because there is a relationship between the EventHandler and EventHdlrElement classes, there is an order dependency between the classes. For instance, an EventHdlrEventScope must be provisioned at the same time or before an EventHdlrElement making use of the EventHdlrEventScope. EventHdlrElement, ContextData and data path class instances referenced by an EventHandler must be provisioned at the same time or before the EventHandler is provisioned.

When the PEP receives an EventHandler Provisioning Decision, it MUST always respond with a Provisioning Report indicating success or failure.

Note that additional EventHdlrElements can simply be added to an existing EventHandler by using the TagId (group identifier) for the EventHandler to which the element is to be added. Additional EventHdlrEventScope or EventHdlrHandleScope instances can be added similarly by adding PRIs with the TagId value of the group these instances are to be added to. This allows incremental updates to be made to the Event Handlers.

7.2. Provisioning Report

A report MUST follow all provisioning decisions described in section 7.1. This report may not have any class instances in it. However, it explicitly notifies the PDP that the provisioning was successful or whether it failed. If many structures were simultaneously provisioned in the Provisioning Decision and a failure occurred, none of the class instances will be accepted by the PEP. Hence it is possible that subsequent Provisioning Decisions occur with a smaller subset of the class instances or an alternative set of class instances that can satisfy the service policies defined in the PDP.

7.3. PEP Event Message

A PEP Event Message is generated by the PEP to indicate that a new class of traffic has been identified by the Event Handler. This Event Message possibly uses a new COPS Request Handle. The decision to use a new COPS Request Handle or reuse an existing Handle is based on the EventHdlrHandleScope information configured in the Event Handler. The Handle Scope information is a set of criteria that is protocol specific, and specifies the set of fields in the protocol that the Event Handler is sensitive towards. The PEP Event Message is essentially a COPS-PR Request message. The PEP Event Message MUST always include an instance of the Event class. This Event instance references the EventHandlerElement instance and EventHandler instance that caught the event. This allows the PDP to identify events belonging to each Event Handler. Other Classes that may be a part of a PEP Event Message include one or more instances of protocol specific Context Data and Interface data classes and optionally an instance of one of the Authentication Extension classes (for example, if the Event is an access event).

When authentication protocols such as PAP or CHAP are in use, the PIB assumes that the UserId, Challenge, and Password will all be determined by the PEP prior to generating the PEP Access Event Message. EAP is an exception to this rule because EAP assumes a direct negotiation between the Endpoint and the Authentication server. For EAP, it is assumed that the Endpoint generates a response to the EAP Identity Request message before the PEP sends the Access Event Message. This allows the PEP to fill in the Username and Realm in the UserAuthExt table. However, for this scenario, it is also assumed that the PEP Access Event Message will

include the EAP Identity Response in the authEapRespExtSpecific attribute of the AuthEapRespExtEntry class. Subsequent EAP negotiation will be performed with the Opaque Decision and Opaque Report message types. When the negotiation is complete the PDP sends a Provisioning Decision message (that includes an instance of the AuthExtResult class specifying success or failure). Note that all interactions resulting from a given Event Message (including authentication negotiation) are performed within the context of a single COPS Request Handle. The COPS Request Handle provides an independent dialog between the PDP and the PEP to fully process an Access Event Message in a synchronous way.

7.3. PDP Provisioning Decision

When the PDP has all the necessary information to determine what policies to provision for the event that was generated by the PEP, and it has completed any intermediate data path provisioning that the event may be dependent on, the PDP SHOULD generate a PDP Provisioning Decision message. The PDP Provisioning Decision message only contains the instances of the classes the PDP wants to configure as a result of the event. In addition to this message the PDP MAY also send unsolicited Provisioning responses on other COPS handles to add policies that may be shared across events.

The PEP is the only entity that knows when traffic is no longer flowing through a particular session (either because of a timer expiring or because of a physical link termination). Therefore the lifetime of a COPS Request handle is always controlled by the PEP. The PDP MAY advise the PEP that the Handle is no longer valid via a provisioning update. However, the ultimate dispensation of the Request Handle and the associated tables are always determined by the PEP. The PDP MAY also indicate that a traffic flow may no longer have access to resources by changing the data path to drop packets arriving for that traffic flow. Since the PDP can modify the data path such that all packets for the flow will be dropped, both alternatives achieve the same semantics. Since a COPS-PR Provisioning Decision is used, the PEP MUST send a report back to the PDP to confirm that there are no problems with the data path change requested by the PDP.

The PEP MAY delete the COPS Request Handle simply by notifying the PDP via a Delete Request Message that the provisioned policies for that Handle are no longer valid.

When a COPS Request Handle is removed, all contained class instances MUST be removed as well. Typically these will include header and authentication table instances.

7.4. PDP fetching Event-specific ContextData

The ContextData class MAY be specified either during the configuration of the EventHandler to indicate what context data should be sent with each PEP Event Message or it MAY be used by the

PDP to get additional context data for an event after it receives an Event Message. In the latter case, the PDP MAY send a solicited decision that specifies ContextData for the last Event Message received on the same Request Handle. The ContextData message contains PRC names to retrieve the specific information. This information may be needed to either authorize a pending event, monitor a set of policies bound to the handle or get more context information regarding the event. Since each ContextData class only retrieves a specific subset of the information regarding the event within the context of a Request Handle, a single request message MAY contain multiple instances of the ContextData class, thereby supporting the retrieval of as much event-specific information as needed in a single message.

The COPS-PR message type used by the PDP to fetch Event-specific ContextData is a Provisioning Decision message. When the PEP receives a message from the PDP asking for Event-specific ContextData, it MUST send an Event-specific ContextData message in a COPS Request message back to the PDP. This request message MUST use the same COPS Request Handle. Since the TagId in the ContextData class is only used when the ContextData class is configured with an EventHandler, the TagId attribute should not be set when the class is used in an Event-specific ContextData Fetch.

The updated Event-specific ContextData Request from the PEP SHOULD contain a set of Header and Interface context data class instances. Since the updated request uses the same Request Handle, the PDP knows which event is being updated by more context data. Using PDP Fetched ContextData messages precludes the PDP from provisioning the PEP to allow multiple simultaneous Event Messages outstanding on the same Handle.

7.5. Event-specific ContextData Response

The Event-specific ContextData Response message is used to report specific interface and/or packet header information back to the PDP. This message is implemented as a COPS-PR Report message. A Report message MAY include any number of Interface or Header table instances. However, because Reference Identifiers to the Event table are not specified in the header or interface data tables, a Report message may contain header and interface data for one and only one Event or the most recent Event Message received on that specific

COPS Request Handle.

7.6. Opaque Decision

An Opaque Decision message is used to send specialized authentication messages from the PDP to the PEP. Specifically, this type of COPS-PR Decision message is used to pass EAP request messages via authEapReqExt table instances.

7.7. Opaque Report

An Opaque Report message is used to send specialized authentication messages from the PEP to the PDP. Specifically, this type of COPS-PR Report message is used to pass EAP response messages via authEapRespExt table instances.

7.8. Combining Data Structures in Messages

In the most degenerate case, the PDP provisions the EventHandler to only send the Event object when an event occurs. The PDP then requests Event-specific Context Data that the PEP will respond to with a Report Message. In addition, if EAP authentication is required, a sequence of Opaque Decisions and Opaque Reports are also required. Finally, if new data paths need to be provisioned (including specialized EventHandlers), normal Provisioning Decision and Report messages must also be exchanged. Note that these provisioning decisions may be on separate COPS Request Handles.

In some environments, for example authorization, it is essential to complete the transaction as quickly as possible. The way to accelerate this process is to combine as many messages into a single message as possible.

8. Access Bind Usage Examples

Following examples on how the Access Bind PIB PRCs are used provide some additional clarifications on the PRC definitions. But they by no means indicate all the PRCs needed for the application given by the example. And providing these examples here does not indicate where the application specific PRCs should be defined. These examples are provided only to assist better and easier understanding of the Access Bind PIB.

8.1 Wireless LAN (802.11 Access Point) Usage Example

A wireless LAN Access Point (AP) is pictured in Figure 7.1.1 below. This is based roughly on 802.11/802.1x concepts. The following is meant to give an indication of how the Access Bind PIB could be included in such an AP. Note that this is an exercise to see if the concepts fit together, not a proposal for exactly how they would fit.

The AP shown below includes a _Service Manager_ (SM), which interfaces with the wireless data interface. For incoming wireless data it separates management frames and level 2 frames. In the following we will deal particularly with Associate and ReAssociate Management Frames.

The SM (as interpreted here) takes Associate and Reassociate management frames and creates a temporary Port Access Entity PAE for the association. The PAE must then be authenticated and provisioned by an external Authentication Server (AS). Communication with the AS is assumed in this model to be mediated by a Policy Enforcement Point (PEP, which is part of the AP. The AS acts as a Policy Decision Point (PDP).

8.1.1 Wireless LAN Access Event Handler Provisioning

In a Access PIB implementation the figure shows the SM sending a REQ at boot time to tell the AS that it is up and what capabilities it has. The PDP returns a configuration to support the SM. In particular, this configuration includes provisioning information for how to instantiate a PAE and what trigger information should be sent by the instantiated PAE to the PDP.
The Event Handler Provisioning is supported by the Access Bind PIB by using the following PRCs in the decision (DEC) message:
- eventHandler
- eventhdlrElement
- eventhdlrEventScope

With eventhdlrEventScopeFilter indicating how the signaling protocol is recognized.

8.1.2 Wireless LAN Access Event Handling

When an event (here a Associate or ReAssociate) is detected the SM/event handler instantiates and initializes a PAE. The initial PAE instance includes an access port which splits internally into a controlled and uncontrolled port.

The controlled port is what is used to pass data from the access port to the external Ethernet. It is controlled in that there is a switch that must be turned on by the authenticator before data can flow. It may also have QoS parameters that can be controlled by the AS. In its initial state the controlled port drops all incoming frames.

The uncontrolled port connects to an internal authenticator. The authenticator creates the initial trigger. In some cases it may need to send an EAP frame back to the Station prior to sending the initial trigger, and other times it may have enough information from the initial Associate or ReAssociate to create the trigger immediately.

The Access Event Handling is supported by the Access Bind PIB.

The PEP creates an instance of event PRC, with eventEventhdlr referencing the eventHandler, and eventCasue referencing eventhdlrElement provisioned in Access Event Handler Provisioning above.

This event PRI will be sent by the PEP to the PDP in a REQ using a new COPS Request Handle. This REQ message may contain additional

PRIs as dictated by how a specific signaling protocol should be handled.

8.1.3 Wireless LAN Access Event Decision

The AS/PDP decides whether the trigger contains enough information to make an authentication decision. If not, it may initiate an EAP dialog through the authenticatior to the STATION.

Once it has enough information the PDP makes a decision and sends a Provisioning message to the AP that sets QoS parameters and _closes_ the switch on the controlled port.

The decision (DEC) message sent by the PDP to the PEP will be using the same COPS Request Handle created in Access Event Handling above. The content (PRCs) carried by the DEC message will depend on the functionality need to be provided. It may be command to _close_ the switch on the controlled port, it may contain QoS parameters. This step is very similar to, if not the same as, provisioning using the DiffServ PIB.

8.2 RSVP Usage Example

RSVP is a signaling protocol used for a variety of purposes including some call setup applications and MPLS label distribution for traffic engineering. RSVP uses a number of message types to negotiate both the hop-by-hop path and the service requirements between a sender and one or more receivers.

Some RSVP messages contain information that helps determine whether the reservation should be accepted or not. However, the router may not equipped with sufficient context to take advantage of the information in determining whether to accept or reject an RSVP message. COPS was designed to pass specific RSVP messages to a PDP (Policy Server). The PDP could then analyze the RSVP message and usually determine whether to accept or reject the reservation.

With the advent of COPS-PR, it became possible to construct more sophisticated policies beyond simple accept or reject messages. However, these more sophisticated policies were targeted for DiffServ rather than RSVP. With the definition of the AccessBind PIB, it becomes possible to provision a router not only to specify which RSVP messages should be sent to the PDP, but also to use existing PIBs to specify how the QoS requirements in a RSVP reservation should be supported in a specific router implementation.

Two types RSVP specific structures are added to AccessBind to

support RSVP. In order to provision the EventHandler class to detect RSVP messages, a number of filter classes must be defined. These filter classes are general purpose and could be used both by EventHandlers and by Classifiers although the semantics of the filter class are somewhat different for each. The other group of classes is the Context Data classes that pass some or all parts of the RSVP message to the PDP when the EventHandler generates an event.

Because COPS assumes that all RSVP message objects are sent to the PDP, each well known RSVP object will be assigned a unique Context Data PRC identifier and the rest of the RSVP object's attributes will be part of the PIB class in the same order and format as in the original RSVP object. The actual PRC mappings for these objects can be found in the PIB definition. For details on the operation of these objects refer to [RSVP] and [INTSERV]. In addition, a PIB class is also defined to support unrecognized RSVP objects.

A Context Data PIB class is also specified to describe the relevant RSVP common header attributes. The attributes in the common header that will be specified are:

1. The RSVP MsgType attribute, which distinguishes a PATH message from a RESV or PATHerr message.
2. The RSVP Flags attribute is used to indicate whether Refresh Reduction is possible or not.
3. The Send TTL (Time To Live) attribute, provides a easy mechanism for determining whether non-RSVP hops have been traversed by comparing this field with the IP TTL field.
4. The In Interface (if known)

5. The Out Interface (if known)

A special context data class, called AllRSVPMsgObjects, is defined to simplify the process of specifying the set of RSVP objects to be included with a COPS-PR Event message. Rather than explicitly specifying every context data class that should be included with the Event message, this class (when referenced by PRC through the ContextDataIfElement attibute of the ContextData class) indicates that all RSVP objects, including the common header class described above, should be encapsulated and propagated to the PDP. All Refresh Reduction related RSVP objects (MESSAGE_ID, MESSAGE_ID_ACK, and MESSAGE_ID_NACK) are explicitly excluded from being sent to the PDP when the AllRSVPMessageObjects attribute is set to True. These objects are specifically for purpose of synchronizing state between RSVP hops and bears no value in the policy decision process. However, a context data PIB object is defined for each of these classes in the event that a PDP determines that it needs these objects.

The EventScope classes have been specified to roughly follow the same mappings as the Context Data PIB classes. However, since the typical criteria for outsourcing a RSVP message are usually rather simple, only a subset of the RSVP objects require mappings to COPS-PR filter classes. If some implementations require support for filtering additional objects, it is trivial to extend the filters. Note that the filters bound to EventHandlers determine whether a matching packet should generate an Event or not.

The RSVP objects that will be mapped to filters in this specification will include the RSVP common header, the RSVP Dclass object, the RSVP session object, and the RSVP style object. The last three are used to describe various characteristics of the data traffic for which the reservation is being performed. Since the filters can describe both AND and OR semantics, the challenge is in organizing the fields of the objects to simplify filter expressions as much as possible. Since this is the primary goal the appropriate attributes of each object have been combined into a single PIB class. The RSVP filter PIB class contains the following attributes. The fields marked with asterisks will be represented as masked values (for IP addresses) and ranges (for UDP/TCP ports) to add flexibility.

RSVP MsgType
RSVP Flags
Send TTL
DCLASS DSCP
Session Dest IP*
Session Protocol
Session Dest Port*
Filter Src IP*
Filter Src Port*
Style value

This paper does not address reservation specification (TSPEC and FLOWSPEC) modifications that depend on the RSVP refresh model. RSVP refresh reduction [REFRESH] is assumed as a simplifying assumption for this application of the Access Bind PIB. However, if support for traditional RSVP refresh is desirable, it can be supported in this model by adding explicit filters for the RSVP FlowSpec and RSVP Tspec objects as specified in [IntServ].

In order to support RSVP outsourcing with the AccessBind PIB the Event Handler must be provisioned with the appropriate settings to recognize specific RSVP messages, create new request handles, and generate events (outsourcing requests). After we have described how

this is accomplished, we will show the actual message flows involved in the RSVP outsourcing process.

The specific PIB classes that need to be provisioned are the EventHandler, EventhdlrElemenet, ContextData, EventhdlrHandleScope, and EventhdlrEventScope. The EventHandler provides a termination point for processing RSVP messages. As RSVP messages arrive, they are directed to the EventHandler by a classifier. In this scenario the EventHandler as behaving as a termination point for all RSVP messages. Hence, the EventHandler class is provisioned with no data path elements following the EventHandler. Therefore, the attribute eventhdlrNonMatchNext is left unassigned.

Alternatively, the EventHandler can also be provisioned such that RSVP and non-RSVP packets alike pass through the EventHandler, but only RSVP messages invoke events. In this case, the attribute eventhdlrNonMatchNext would specify the next data path element that should process any packets not matching the EventHandler's criteria (non RSVP packets).

The EventhdlrElement class identifies a specific category of events. Suppose one wanted to generate different Events for PATH messages and RESV messages. This could be done by configuring one EventhdlrElement to only match PATH messages and another EventhdlrElement to only match RESV messages. Event messages contain a reference to the EventhdlrElement that generated the event. Therefore, it is possible to generate different events from the same EventHandler.

The EventhdlrElement contains two main semantics. First, it specifies the criteria for creating new Request Handles. Each Request Handle constitutes a unique dialog between the PEP and PDP. The second semantic is the criteria for generating events. In some situations, it is desirable to generate a one-time event and not generate events when similar messages are seen later. A good example of this is RSVP Refresh messages. When RSVP Refresh messages are used to indicate that the reservation is still active, generating events for each message is inappropriate. In contrast, when Refresh Reduction [Refresh] is active, only reservation changes are propagated as full RSVP messages. In this situation, every message may constitute an Event.

With RSVP it is usually appropriate to assign a unique COPS-PR Request Handle for every new RSVP session. Since EventHandlers are typically bound to an interface data path, the RSVP Path message and the Resv message will be processed on different data paths. Therefore, unique events and unique COPS-PR Request Handles will

typically be assigned for each message type. However, this is not significant since the provisioning objectives for Path messages are different from the provisioning objectives for Resv messages. For RSVP, the EventhdlrElement will use the Tag Reference EventhdlrElementHandleScope to describe the criteria for creating a unique handle. Each EventhdlrHandleScope object will contain pointers to the RSVP filter objects mentioned earlier to describe the various fields whose combination of values constitute a unique handle.

Typically, the Filter class used is the RSVP filter class. For this class, the session attributes (SessionDestIP, SessionDestPort, and SessionProtocol) will be assigned wildcard values and all other attributes assigned to NULL to indicate that any combination of these attributes constitutes a unique handle. When various messages arrive that require the generation of an event and that have a newly unique combination of the filter attribute values, a new request handle will be assigned. When a message arrives for which a previous message has already generated a handle, that handle is used to pass the appropriate event to the PDP.

The other class pointed to by the EventhdlrElement is the EventhdlrEventScope. This class describes the criteria for generating an event. Typically, the MsgType attribute in conjunction with the session attributes will be wildcarded and the other fields assigned to NULL to indicate that all RSVP messages should be sent to the PDP. This describes the criteria for an event: Every time a unique combination of all these attributes occurs, generate a new event.

In many cases it may make sense to assign the filter attributes (SessionDestIP and SessionDestPort) to the EventhdlrEventScope and/or EventhdlrHandleScope class. This would be done when it is desirable to notify of the PDP of the need to allocate additional resources to a set of reserved flows going to the same destination but originating from different sources.

A new COPS-PR Request Handle MUST only be created when a valid event occurs. If a packet matches the criteria described by EventhdlrHandleScope but does not match any EventhdlrEventScope criteria, a COPS-PR Request Handle must not be generated.

This version of the paper only describes how RSVP can be supported when Refresh Reduction [REFRESH] is being used. The complexity of addressing the distinction between RSVP refresh messages and reservation update messages is too great to be addressed in this version. Any RSVP message containing bundle messages (MsgType 12)

MUST be decomposed and each message in the bundle must be iteratively processed through the EventHandler as if individual RSVP messages were generated from the RSVP neighbor. Whether EventhdlrMatchNext applies to the individual sub-messages or the bundled message is beyond the scope of this paper.

Irrespective of whether Refresh Reduction is in use or not, the RSVP daemon is responsible for aging out reservations that are no longer valid. As with traditional COPS, when a reservation is aged out, the RSVP daemon or other entity responsible for aging out reservations MUST take responsibility for deleting COPS Request Handles. This allows the PDP to clean up state associated with the reservation and ensures the proper removal of any policies in the PEP specifically assigned through the COPS Request Handle.

Figure 7.1 shows how the various Event Handler objects would be provisioned in a router to ensure that an event is generated for every RSVP message.

Fig 8.1: Representation of an Event Handler for RSVP

Figure 8.1 represents the set of PIB classes that would be provisioned in order to indicate to the PEP that RSVP messages should generate unique events for any combination of filters or sessions. However, all messages using the same unique session will

share the same COPS Request Handle.

When an RSVP message arrives at the PEP with an new combination of session attribute values, the PEP will create a new COPS Request Handle. Following this, an Event message will be generated

containing an Event object with references to EventHandler EH1 and EventhdlrElement Elem1. These two pieces of information allow the PDP to determine which provisioned EventHandler and which specific event type generated the event. In addition the Event message also contains a set of Context Data objects. Since the AllRSVPMsgObjects class was specified in the ContextData class, all RSVP objects are encapsulated in COPS-PR PIB classes and sent to the PDP in the Event message.

When the PDP receives the Event message, it determines what policies to provision in the PEP. Suppose the RSVP message was a reservation request for a controlled load service with a bandwidth allocation of 1 Mbps and session object contained (SessionDestIP = 1.2.3.4, SessionProtocol=UDP, SessionDestPort=7788). If the router's implementation only supported 4 queues with respective bandwidth allocations of 20Mb, 40Mb, 30Mb, and 10Mb, the PDP may decide that allocating the reservation to queue 3 can satisfy the reservation request. Hence, a PDP might generate a provisioning policy as a result of the PEP's Event message that creates a new Classifier Element and Filter that matches all 1.2.3.4:7788 traffic and directs it to queue 3.

9. The AccessBind PIB Module

```
   --
   -- The AccessBind PIB Module
   --

   ACCESSBIND-PIB PIB-DEFINITIONS ::= BEGIN

      IMPORTS
          Unsigned32, Integer32, MODULE-IDENTITY,
          MODULE-COMPLIANCE, OBJECT-TYPE, OBJECT-GROUP, pib
                  FROM COPS-PR-SPPI
          InstanceId, Prid
                  FROM COPS-PR-SPPI-TC
          RoleCombination, PrcIdentifierOid
                  FROM FRAMEWORK-TC-PIB
          InetAddress, InetAddressType
                  FROM INET-ADDRESS-MIB
          TruthValue, PhysAddress
                  FROM SNMPv2-TC;

      accessBindPib  MODULE-IDENTITY
          SUBJECT-CATEGORIES { all }
          LAST-UPDATED "200202202002Z"
          ORGANIZATION "IETF RAP WG"
          CONTACT-INFO "
                        Walter Weiss
                        Ellacoya Networks
                        7 Henry Clay Drive
                        Merrimack, NH 03054
                        Phone: 603-879-7364
                        E-mail: wweiss@ellacoya.com
                        "
          DESCRIPTION
                "A PIB module containing the set of classes to
                configure generic event handlers, and outsource
                events as they occur. One application of this PIB is
                to bind authorization and authentication to COPS
                Provisioning."
```

```
    ::= { pib 4 }    -- xxx to be assigned by IANA


--
-- The branch OIDs in the AccessBind PIB
--

eventClasses OBJECT IDENTIFIER ::= { accessBindPib 1 }
eventHdlrClasses OBJECT IDENTIFIER ::= { accessBindPib 2 }
contextClasses OBJECT IDENTIFIER ::= { accessBindPib 3 }
```



```
--
-- Event Table
--
-- Instances of this table represent events that occurred at
-- the PEP. The events reference the event handler instance
-- and the specific event handler element that the event was
-- caught by.
eventTable OBJECT-TYPE
    SYNTAX         SEQUENCE OF EventEntry
    PIB-ACCESS     notify
    STATUS         current
    DESCRIPTION
        "An instance of this class is created by the PEP and sent
        to the PDP. As a result of this event, The PDP may send
        additional unsolicited decisions to the PEP after
        sending the mandatory solicited decision for the event."

    ::= { eventClasses 1 }

eventEntry OBJECT-TYPE
    SYNTAX         EventEntry
    STATUS         current
    DESCRIPTION
        "An instance of the eventTable PRC."

    PIB-INDEX { eventId }
    UNIQUENESS { }

    ::= { eventTable 1 }

EventEntry ::= SEQUENCE {
    eventId                 InstanceId,
    eventEventHdlr          ReferenceId,
    eventCause              ReferenceId
```

```
    }

    eventId  OBJECT-TYPE
        SYNTAX         InstanceId
        STATUS         current
        DESCRIPTION
            "An index to uniquely identify this event."

        ::= { eventEntry 1 }

    eventEventHdlr  OBJECT-TYPE
        SYNTAX         ReferenceId
        PIB-REFERENCES {  frwkReferenceEntry }
        STATUS         current
        DESCRIPTION
            "This attribute allows a PEP to indicate to the PDP that
            this event was generated due to the referenced Event
            Handler. This attribute references an event handler via
            the indirection PRC frwkReference, since the event
```



```
            handler and event could potentially belong to a different
            PIB contexts."


        ::= { eventEntry 2 }


    eventCause OBJECT-TYPE
        SYNTAX         ReferenceId
        PIB-REFERENCES { frwkReferenceEntry }
        STATUS         current
        DESCRIPTION
            "This attribute references the specific instance in a
            group of event Handler elements belonging to an event
            Handler that resulted in this event. This attribute
            references a specific event handler element via the
            indirection PRC frwkReference, since the event handler
            element and event could potentially belong to a different
            PIB contexts."


        ::= { eventEntry 3 }


    --
    -- EventHandler Table
    --
    -- Instances of this PRC are provisioned by the PDP on the PEP
    -- to catch specific events. The Event Handlers reference a
```

```
      -- group of eventHdlrElement PRIs that contain the scope of
      -- the event and specify the context data to send to the PDP
      -- when an event is caught.

      eventHandlerTable OBJECT-TYPE
          SYNTAX          SEQUENCE OF EventHandlerEntry
          PIB-ACCESS      install
          STATUS          current
          DESCRIPTION
              "The eventHandlerTable specifies for what events the PEP
              should send a request to the PDP. As a  result of this
              request, the PDP may send configuration changes to the
              PEP. An instance of this class defines the circumstances
              for generating a request, and provides the means for
              specifying the contents of the PEP Request. Hence, the
              eventHandlerTable can be said to create eventTable
              entries. "

          ::= {  eventHdlrClasses 1 }

      eventHandlerEntry OBJECT-TYPE
          SYNTAX  EventHandlerEntry
          STATUS  current
          DESCRIPTION
```



```
              "eventTable entry."
          PIB-INDEX { eventHandlerId }
          UNIQUENESS {    eventHandlerElements,
                          eventHandlerNonMatchNext
                     }

          ::= { eventHandlerTable 1}

      EventHandlerEntry ::= SEQUENCE {
          eventHandlerId                  InstanceId,
          eventHandlerElements            TagReferenceId,
          eventHandlerNonMatchNext        Prid
      }

      eventHandlerId OBJECT-TYPE
          SYNTAX    InstanceId
          STATUS        current
          DESCRIPTION
              "An arbitrary integer index that uniquely identifies
              an instance of the eventHandlerTable class."

          ::= { eventHandlerEntry 1}

      eventHandlerElements OBJECT-TYPE
```

```
            SYNTAX        TagReferenceId
            PIB-TAG       { eventHdlrElementGrpId }
            STATUS        current
            DESCRIPTION
                "A reference to a group of eventHdlrElement instances,
                each of which  determines the scope (criteria for
                generating a new  request) and what context information
                to send in a request."

            ::= { eventHandlerEntry 2}

        eventHandlerNonMatchNext  OBJECT-TYPE
            SYNTAX       Prid
            STATUS       current
            DESCRIPTION
                "The data path for 'out of scope' traffic."

            ::= { eventHandlerEntry 3}


        --
        -- EventHdlrElement Table
        --
        -- Each Instance of this PRC belongs to a group of
        -- eventHdlrElement PRIs. The group is identified by the
        -- eventHdlrElementGrpId attribute. These are provisioned by
        -- the PDP on the PEP to catch specific events. This PRC
        -- contain the scope of the event and specify the context data
        -- type to send to the PDP when an event is caught.
```



```
        eventHdlrElementTable OBJECT-TYPE
            SYNTAX          SEQUENCE OF EventHdlrElementEntry
            PIB-ACCESS      install
            STATUS          current
            DESCRIPTION
                "The eventHdlrElementTable specifies a single eventHdlr
                element's scope via a reference to a group of filters and
                the context data type and encapsulation meta-information
                that the PEP needs to send an event notification to the
                PDP."

            ::= { eventHdlrClasses 2 }

        eventHdlrElementEntry OBJECT-TYPE
            SYNTAX  EventHdlrElementEntry
            STATUS  current
            DESCRIPTION
                "eventTable entry."
```

```
        PIB-INDEX { eventHdlrElementId }
        UNIQUENESS {    eventHdlrElementEventCriteria,
                        eventHdlrElementGrpId,
                        eventHdlrElementEventScope,
                        eventHdlrElementHandleScope,
                        eventHdlrElementContext,
                        eventHdlrElementMatchNext
                   }

        ::= { eventHdlrElementTable 1}

    EventHdlrElementEntry ::= SEQUENCE {
        eventHdlrElementId                      InstanceId,
        eventHdlrElementEventCriteria           Unsigned32,
        eventHdlrElementGrpId                   TagId,
        eventHdlrElementEventScope              TagReferenceId,
        eventHdlrElementHandleScope             TagReferenceId,
        eventHdlrElementContext                 TagReferenceId,
        eventHdlrElementMatchNext               Prid
    }

    eventHdlrElementId OBJECT-TYPE
        SYNTAX    InstanceId
        STATUS        current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies
            an instance of the eventHdlrElementTable class."

        ::= { eventHdlrElementEntry 1}

    eventHdlrElementEventCriteria  OBJECT-TYPE
        SYNTAX        Unsigned32 {
                                   one_time(1),
                                   every_time(2),
```



```
                                   on_change(3)
                                 }
        STATUS        current
        DESCRIPTION
            "Indicates when an event is generated. Valid options are
            one_time, every_time and on_change. This attribute allows
            event Handlers to distinguish one time events (ignore
            after the first match) from recurring events (generate an
            event every time a match occurs).  A enum type was also
            define to specify that a new event should be generated
            when a specific set of fields change. This is important
            for protocols like RSVP because messages are sent both to
            demonstrate that the reservation is active and to notify
```

```
            hops of changes to reservations. Since only changes need
            to propagate to the PDP, the on_change option indicates
            that that events should be generated selectively.

            This criteria controls behavior of both, the EventScope
            and the HandleScope."

        ::= { eventHdlrElementEntry 2}

    eventHdlrElementGrpId OBJECT-TYPE
        SYNTAX        TagId        -- corresponding Tag Reference in
                                    -- eventHandlerEntry
        STATUS        current
        DESCRIPTION
            "Group identifier. All instances with the same group
            identifier belong to one group and can be referenced
            collectively from an eventHandler instance."

        ::= { eventHdlrElementEntry 3}

    eventHdlrElementEventScope OBJECT-TYPE
        SYNTAX        TagReferenceId
        PIB-TAG       { eventHdlrEventScopeGroup }
        STATUS        current
        DESCRIPTION
            "Identifies a group of eventHdlrEventScope entries
            associated with this eventHdlrElement instance."

        ::= { eventHdlrElementEntry 4}

    eventHdlrElementHandleScope OBJECT-TYPE
        SYNTAX        TagReferenceId
        PIB-TAG       { eventHdlrHandleScopeGroup }
        STATUS        current
        DESCRIPTION
            "Identifies a group of eventHdlrHandleScope entries
            associated with this eventHdlrElement instance. This is
            an optional attribute. If it is not present the
            semantics of the Handle processing is interpreted as
            identical to the Event Scope handling specified in the
```



```
            EventScope objects"

        ::= { eventHdlrElementEntry 5}

    eventHdlrElementContext OBJECT-TYPE
        SYNTAX       TagReferenceId
        PIB-TAG       { contextDataGroup }
        STATUS        current
```

```
            DESCRIPTION
                "Identifies a list of ContextDataTable entries
                  associated with this eventHdlrElement instance."

            ::= { eventHdlrElementEntry 6}

        eventHdlrElementMatchNext OBJECT-TYPE
            SYNTAX        Prid
            STATUS        current
            DESCRIPTION
                "The data path for traffic in scope."

            ::= { eventHdlrElementEntry 7}


        --
        -- EventHdlrEventScope Table
        --
        -- This PRC defines the scope of an event handler element using
        -- references to filters defined in the Framework PIB or in some
        -- other PIBs. These filters may describe specific protocol
        -- properties for which events need to be generated. These filter
        -- references are grouped using a TagId, and this group is then
        -- referenced from the eventHdlrElement PRC.

        eventHdlrEventScopeTable OBJECT-TYPE
            SYNTAX          SEQUENCE OF EventHdlrEventScopeEntry
            PIB-ACCESS      install
            STATUS          current
            DESCRIPTION
                "This class defines the criteria to be used for
                partitioning various portions of traffic."

            ::= { eventHdlrClasses 3 }

        eventHdlrEventScopeEntry OBJECT-TYPE
            SYNTAX  EventHdlrEventScopeEntry
            STATUS  current
            DESCRIPTION
                "An instance of this class defines an individual
                criterion to be used towards generating an event."
            PIB-INDEX { eventHdlrEventScopeId }
            UNIQUENESS { eventHdlrEventScopeGroup,
                         eventHdlrEventScopeFilter
                       }
```



```
            ::= { eventHdlrEventScopeTable 1}
```

```
EventHdlrEventScopeEntry::= SEQUENCE {
    eventHdlrEventScopeId         InstanceId,
    eventHdlrEventScopeGroup      TagId,
    eventHdlrEventScopeFilter     Prid,
    eventHdlrEventScopePrecedence INTEGER,
    eventHdlrEventScopeChangeFlag TruthValue
}

eventHdlrEventScopeId    OBJECT-TYPE
    SYNTAX        InstanceId
    STATUS        current
    DESCRIPTION
        "An arbitrary integer index that uniquely identifies an
        instance of the eventHdlrEventScopeTable class."

    ::= { eventHdlrEventScopeEntry 1}

eventHdlrEventScopeGroup  OBJECT-TYPE
    SYNTAX        TagId   -- corresponding TagReference
                          -- defined in eventHdlrElementEntry
    STATUS        current
    DESCRIPTION
        "Represents the binding between the eventHdlrElementEntry
        and the eventHdlrEventScope entries. A group of
        eventHdlrEventScope entries constitutes the criteria for
        partitioning various portions of traffic."

    ::= { eventHdlrEventScopeEntry 2}

eventHdlrEventScopeFilter   OBJECT-TYPE
    SYNTAX        Prid
    STATUS        current
    DESCRIPTION
        "Pointer to a filter to be used as the criteria."
    ::= { eventHdlrEventScopeEntry 3}

eventHdlrEventScopePrecedence OBJECT-TYPE
    SYNTAX        INTEGER
    STATUS        current
    DESCRIPTION
        "Represents the precedence of this criterion with respect
        to other criteria within the same group. When the
        precedence is unique, the instance represents an
        alternative criteria (an ORing function). When the
        precedence for two or more instances of the
        eventHdlrEventScope class is the same, the attributes
        within all the instances are treated collectively as a
        single filter criteria with the following rules:
        1. If the filters are not of the same type, the filters
           are AND'ed as a whole eg (RSVP and IP)
```

```
            2. If the filter types are the same, the attribute values
               are OR'ed and the attributes themselves are AND'ed,
               for example, two IP filters with src protocol values
               56 and 57 respectively and dst protocol values 20 and
               25 , would be treated as the condition (src port (56
               or 57) AND dst port (20 or 25)."

        ::= { eventHdlrEventScopeEntry 4}

    eventHdlrEventScopeChangeFlag OBJECT-TYPE
        SYNTAX        TruthValue
        STATUS        current
        DESCRIPTION
            "Boolean value, if set to 'true' indicates that a new
            event should be generated if any of the assigned fields
            in the associated filter change."

        ::= { eventHdlrEventScopeEntry 5}


    --
    --
    -- ContextData Table
    --
    -- This PRC specifies the context information to send to the PDP
    -- when an event is caught. The context information to send is
    -- described in terms of the PRC data types to include in the
    -- request, the level of encapsulated data and the interface
    -- information for that request.


    contextDataTable OBJECT-TYPE
        SYNTAX          SEQUENCE OF ContextDataEntry
        PIB-ACCESS      install
        STATUS          current
        DESCRIPTION
            "This class points to the context information to be
            included with a request."

        ::= { contextClasses 1 }

    contextDataEntry OBJECT-TYPE
        SYNTAX  ContextDataEntry
        STATUS  current
        DESCRIPTION
            "An instance of this class contains the type description
            (the assigned OID) of the class which needs to be filled
            in by the PEP and included with a PEP request."
        PIB-INDEX { contextDataId }
```

```
        UNIQUENESS { }

        ::= { contextDataTable 1}
```



```
    ContextDataEntry::= SEQUENCE {
        contextDataId            InstanceId,
        contextDataGroup         TagId,
        contextDataIfElement     PrcIdentifierOid,
        contextDataEncapsulation INTEGER
    }

    contextDataId   OBJECT-TYPE
        SYNTAX        InstanceId
        STATUS        current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies an
            instance of the contextDataTable class."

        ::= { contextDataEntry 1}

    contextDataGroup  OBJECT-TYPE
        SYNTAX        TagId   --corresponding TagReference
                              --defined in eventHdlrElement
        STATUS        current
        DESCRIPTION
            "Defines the grouping of contextData instances
            that are applicable to a given eventHdlrElement. When
            instances of this PRC are sent to the PEP without the
            event Handler information, this attribute is unused."

        ::= { contextDataEntry 2}


    contextDataIfElement      OBJECT-TYPE
        SYNTAX        PrcIdentifierOid
        STATUS        current
        DESCRIPTION
            "The OID of a class whose instance is to be included with
            the PEP request or event-specific ContextData Response."

        ::= { contextDataEntry 3}

    contextDataEncapsulation OBJECT-TYPE
        SYNTAX        INTEGER
        STATUS        current
        DESCRIPTION
            "This attribute allows one to distinguish between inner
            and outer headers when there are multiple encapsulated
```

```
            headers of the same type in a packet.

            A value of:
            0 means all headers,
            positive number 'n' means the 'n'th header starting
            from the outermost,
            negative number 'n' means the 'n'th header starting from
            the innermost."
        ::= { contextDataEntry 4}
```



```
    --
    -- Layer 3 Header Data PRC
    --

    ctxtL3HdrTable OBJECT-TYPE
        SYNTAX         SEQUENCE OF ctxtL3HdrEntry
        PIB-ACCESS     notify
        STATUS         current
        DESCRIPTION
            "An instance of this class is created by the PEP and
            sent to the PDP to provide the PDP with information it
            requested in the ContextData PRC. The PDP uses
            this PRC to make Authentication/Provisioning
            decisions."

        ::= { contextClasses 2 }

    ctxtL3HdrEntry OBJECT-TYPE
        SYNTAX         CtxtL3HdrEntry
        STATUS         current
        DESCRIPTION
            "An instance of the ctxtL3HdrTable PRC."

        PIB-INDEX { ctxtL3HdrId }
        UNIQUENESS { }

        ::= { ctxtL3HdrTable 1 }

    CtxtL3HdrEntry::= SEQUENCE {
        ctxtL3HdrId               InstanceId,
        ctxtL3HdrSrcAddrType        InetAddressType,
        ctxtL3HdrSrcAddr            InetAddress,
        ctxtL3HdrDstAddrType        InetAddressType,
        ctxtL3HdrDstAddr            InetAddress,
        ctxtL3HdrProtocol           Unsigned32,
        ctxtL3HdrSrcPort            Unsigned32,
        ctxtL3HdrDstPort            Unsigned32,
```

```
        ctxtL3HdrDscp               Unsigned32,
        ctxtL3HdrEcn                TruthValue,
        ctxtL3HdrIpOpt              OCTET STRING,
        ctxtL3HdrEncap              Integer32
    }

    ctxtL3HdrId  OBJECT-TYPE
        SYNTAX          InstanceId
        STATUS          current
        DESCRIPTION
            "An index to uniquely identify an instance of this
            provisioning class."

        ::= { ctxtL3HdrEntry 1 }
```



```
    ctxtL3HdrSrcAddrType OBJECT-TYPE
        SYNTAX          InetAddressType
        STATUS          current
        DESCRIPTION
            "The address type enumeration value [INETADDR] to specify
             the type of the packet's source L3 address)."

        ::= { ctxtL3HdrEntry 2 }

    ctxtL3HdrSrcAddr OBJECT-TYPE
        SYNTAX          InetAddress
        STATUS          current
        DESCRIPTION
            " The packet's source L3 address."

        ::= { ctxtL3HdrEntry 3 }

    ctxtL3HdrDstAddrType OBJECT-TYPE
        SYNTAX          InetAddressType
        STATUS          current
        DESCRIPTION
            "The address type enumeration value [INETADDR] to specify
             the type of the packet's destination L3 address."

        ::= { ctxtL3HdrEntry 4 }


    ctxtL3HdrDstAddr OBJECT-TYPE
        SYNTAX          InetAddress
        STATUS          current
        DESCRIPTION
            "The packet's destination L3 address."
```

```
        ::= { ctxtL3HdrEntry 5 }


    ctxtL3HdrProtocol OBJECT-TYPE
        SYNTAX         Unsigned32
        STATUS         current
        DESCRIPTION
            "The packet's protocol field."

        ::= { ctxtL3HdrEntry 6 }

    ctxtL3HdrSrcPort  OBJECT-TYPE
        SYNTAX         Unsigned32
        STATUS         current
        DESCRIPTION
            "This attribute binds an existing upstream session to
            this session instance."

        ::= { ctxtL3HdrEntry 7 }
```



```
    ctxtL3HdrDstPort  OBJECT-TYPE
        SYNTAX         Unsigned32
        STATUS         current
        DESCRIPTION
            "This attribute binds an existing upstream session to
            this session instance."

        ::= { ctxtL3HdrEntry 8 }

    ctxtL3HdrDscp  OBJECT-TYPE
        SYNTAX         Unsigned32
        STATUS         current
        DESCRIPTION
            "DiffServ CodePoint."

        ::= { ctxtL3HdrEntry 9 }

    ctxtL3HdrEcn  OBJECT-TYPE
        SYNTAX         TruthValue
        STATUS         current
        DESCRIPTION
            "PEP sets this attribute to true(1) if ECN capable."

        ::= { ctxtL3HdrEntry 10 }

    ctxtL3HdrIpOpt  OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
```

```
            DESCRIPTION
                "IP Options field in the packet."

            ::= { ctxtL3HdrEntry 11 }

        ctxtL3HdrEncap  OBJECT-TYPE
            SYNTAX         Integer32
            STATUS         current
            DESCRIPTION
                "This attribute specifies which encapsulated header is
                being described. The sign on this value will be the same
                as the value specified in the ContextData
                instance that requested this header. If the original
                ContextData instance specified a
                ContextDataEncapsulation value of zero (meaning
                return all headers), then all instances of this attribute
                MUST be expressed as positive numbers.

                A value of:

                positive number 'n' means the 'n'th header starting
                from the outermost,
                negative number 'n' means the 'n'th header starting from
                the innermost."
```



```
            ::= { ctxtL3HdrEntry 12 }


        --
        -- 802.1 Header Data PRC
        --

        ctxt802HdrTable OBJECT-TYPE
            SYNTAX         SEQUENCE OF Ctxt802HdrEntry
            PIB-ACCESS     notify
            STATUS         current
            DESCRIPTION
                "An instance of this class is created by the PEP and sent
                to the PDP to provide the PDP with information it
                requested in the ContextData PRC. The PDP uses this PRC
                to make Authorization/Provisioning decisions."

            ::= { contextClasses 3 }

        ctxt802HdrEntry OBJECT-TYPE
            SYNTAX         Ctxt802HdrEntry
            STATUS         current
            DESCRIPTION
```

```
            "An instance of the ctxt802HdrTable PRC."

        PIB-INDEX { ctxt802HdrId }
        UNIQUENESS { }

        ::= { ctxt802HdrTable 1 }

    Ctxt802HdrEntry::= SEQUENCE {
        ctxt802HdrId               InstanceId,
        ctxt802HdrSrcAddr          PhysAddress,
        ctxt802HdrDstAddr          PhysAddress,
        ctxt802HdrProtocol         Unsigned32,
        ctxt802HdrPriority         Unsigned32,
        ctxt802HdrVlan             Unsigned32,
        ctxt802HdrEncap            Integer32
    }

    ctxt802HdrId  OBJECT-TYPE
        SYNTAX         InstanceId
        STATUS         current
        DESCRIPTION
            "An index to uniquely identify an instance of this
            provisioning class."

        ::= { ctxt802HdrEntry 1 }


    ctxt802HdrSrcAddr OBJECT-TYPE
        SYNTAX         PhysAddress
```



```
        STATUS         current
        DESCRIPTION
            " The packet's source MAC address."

        ::= { ctxt802HdrEntry 2 }

    ctxt802HdrDstAddr OBJECT-TYPE
        SYNTAX         PhysAddress
        STATUS         current
        DESCRIPTION
            "The packet's destination MAC address."

        ::= { ctxt802HdrEntry 3 }


    ctxt802HdrProtocol OBJECT-TYPE
        SYNTAX         Unsigned32 (0..'ffff'h)
        STATUS         current
        DESCRIPTION
```

```
            "The L2 packet's protocol field."

        ::= { ctxt802HdrEntry 4 }


    ctxt802HdrPriority OBJECT-TYPE
        SYNTAX         Unsigned32 (0..7)
        STATUS         current
        DESCRIPTION
            "The L2 packet's priority field. This attribute is only
            valid for packets using the 802.1q header extension."

        ::= { ctxt802HdrEntry 5 }

    ctxt802HdrVlan OBJECT-TYPE
        SYNTAX         Unsigned32 (1..4094)
        STATUS         current
        DESCRIPTION
            "The L2 packet's VLAN field. This attribute is only valid
            for packets using the 802.1q header extension."

        ::= { ctxt802HdrEntry 6 }

    ctxt802HdrEncap OBJECT-TYPE
        SYNTAX         Integer32
        STATUS         current
        DESCRIPTION
            "This attribute specifies which encapsulated header is
            being described. The sign on this value will be the same
            as the value specified in the ContextData
            instance that requested this header. If the original
            ContextData instance specified an
            ContextDataEncapsulation value of zero (meaning
            return all headers), then all instances of this attribute
```



```
            MUST be expressed as positive numbers.

            A value of:
            positive number 'n' means the 'n'th header starting
            from the outermost,
            negative number 'n' means the 'n'th header starting from
            the innermost."

           ::= { ctxt802HdrEntry 7 }


    --
    -- conformance section tbd
    --
```

```
END
```


10. Identity Extensions PIB

```
--
-- The AccessBind Identity Extensions PIB Module
--

ACCESSBIND-IDENTEXT-PIB PIB-DEFINITIONS ::= BEGIN

   IMPORTS
       MODULE-IDENTITY, MODULE-COMPLIANCE,
       OBJECT-TYPE, OBJECT-GROUP, pib
               FROM COPS-PR-SPPI
```

```
        InstanceId, Prid
                FROM COPS-PR-SPPI-TC
        TruthValue
                FROM SNMPv2-TC;

    accessBindIdentityExtPib  MODULE-IDENTITY
        SUBJECT-CATEGORIES { all }
        LAST-UPDATED "200211032002Z"
        ORGANIZATION "IETF RAP WG"
        CONTACT-INFO "
                      Walter Weiss
                      Ellacoya Networks
                      7 Henry Clay Drive
                      Merrimack, NH 03054
                      Phone: 603-879-7364
                      E-mail: wweiss@ellacoya.com
                     "
        DESCRIPTION
            "A PIB module containing the set of classes to
            associate authentication protocols with configured
            event handlers, and outsource authentication events
            as they occur."

        ::= { pib 7 }    -- xxx to be assigned by IANA


    --
    -- The branch OIDs in the AccessBind Signalling PIB
    --

    identEventHdlrClasses OBJECT IDENTIFIER ::= {
        accessBindIdentityExtPib  1 }
    identAuthClasses OBJECT IDENTIFIER ::= {
        accessBindIdentityExtPib  2 }

    --
    -- Identity Event Handler Table
    --
    -- This PRC is an extension of the EventHandler PRC. This
    -- extension illustrates the use of the EventHandler PRC
    -- concept for authentication usage. Instances of this PRC are
```



```
    -- provisioned by the PDP on the PEP to catch specific access
    -- events. This PRC references a group of
    -- eventHdlrAuthProtocol instances which define a set of
    -- Authentication mechanisms to use if an access event is
    -- caught by this event Handler. From its base class (Event
    -- Handler) this PRC also references a group of
```

```
-- eventHdlrElement PRIs that contain the scope of the
-- access event and specify the context data to send to the
-- PDP when an access event is caught.

identityEventHdlrTable OBJECT-TYPE
    SYNTAX          SEQUENCE OF IdentityEventHdlrEntry
    PIB-ACCESS      install
    STATUS          current
    DESCRIPTION
        "The identityEventHdlrTable specifies for what access
        events the PEP should send an access request to the PDP.
        As a result of this access request, the PEP may send
        configuration changes to the PEP or specific policies for
        specific users. An instance of this class defines the
        circumstances for generating an access request, and
        provides the means for specifying the authentication
        mechanisms and contents of the PEP Request. Hence, the
        identityEventHdlrTable can be said to create eventTable
        entries for user access. "

    ::= {  identEventHdlrClasses 1 }

identityEventHdlrEntry OBJECT-TYPE
    SYNTAX  IdentityEventHdlrEntry
    STATUS  current
    DESCRIPTION
        "identityEventHdlrTable entry."
    EXTENDS { eventHandlerEntry }
    UNIQUENESS {    eventHandlerElements,
                    eventHandlerNonMatchNext,
                    identityEventHdlrRequestAuth
               }

    ::= { identityEventHdlrTable 1}

IdentityEventHdlrEntry ::= SEQUENCE {
    identityEventHdlrRequestAuth    TruthValue,
    identityEventHdlrAuthProtocol   TagReferenceId
}

identityEventHdlrRequestAuth    OBJECT-TYPE
    SYNTAX        TruthValue
    STATUS        current
    DESCRIPTION
        "Boolean flag, if set to 'true' requires authentication
        data to be sent in the request sent to the PDP with the
        access event."
```

```
    ::= { identityEventHdlrEntry 1}


identityEventHdlrAuthProtocol    OBJECT-TYPE
    SYNTAX        TagReferenceId
    PIB-TAG       { eventHdlrAuthProtocolGroup }
    STATUS        current
    DESCRIPTION
        "References a group of eventHdlrAuthProtocol instances,
        each of which specifies an authentication mechanism."

    ::= { identityEventHdlrEntry 2}



--
-- EventHdlrAuthProtocol Table
--
-- This PRC specifies the Auth Mechanism to use in the Access
-- request when a identity Event Handler is configured to
-- catch access events.
--

eventHdlrAuthProtocolTable OBJECT-TYPE
    SYNTAX          SEQUENCE OF EventHdlrAuthProtocolEntry
    PIB-ACCESS      install
    STATUS          current
    DESCRIPTION
        "This class lists the authentication protocols that can
        be used for an access request."

    ::= { identEventHdlrClasses 2 }

eventHdlrAuthProtocolEntry OBJECT-TYPE
    SYNTAX  EventHdlrAuthProtocolEntry
    STATUS  current
    DESCRIPTION
        "An instance of this class describes an authentication
        protocol that may be used for an access request.
        Instances of this class that share the same TagId value
        collectively constitute a list of authentication
        protocols that may be used for a given access request"
    PIB-INDEX { eventHdlrAuthProtocolId }
    UNIQUENESS { eventHdlrAuthProtocolGroup,
                 eventHdlrAuthProtocolAuthMechanism
               }

    ::= { eventHdlrAuthProtocolTable 1}

EventHdlrAuthProtocolEntry::= SEQUENCE {
    eventHdlrAuthProtocolId             InstanceId,
    eventHdlrAuthProtocolGroup          TagId,
```

```
        eventHdlrAuthProtocolAuthMechanism  INTEGER
    }

    eventHdlrAuthProtocolId    OBJECT-TYPE
        SYNTAX        InstanceId
        STATUS        current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies an
            instance of the ContextDataTable class."

        ::= { eventHdlrAuthProtocolEntry 1}

    eventHdlrAuthProtocolGroup OBJECT-TYPE
        SYNTAX        TagId  -- corresponding TagReference
                             -- in identityEventHdlrEntry
        STATUS        current
        DESCRIPTION
            "Represents a binding between an identityEventHdlrTable
            instance and a list of eventHdlrAuthProtocolTable
            instances."

        ::= { eventHdlrAuthProtocolEntry 2}

    eventHdlrAuthProtocolAuthMechanism OBJECT-TYPE
        SYNTAX        INTEGER {
                                  PAP (0),
                                  CHAP_MD5 (1),
                                  CHAP_MS (2),
                                  EAP_MD5(3),
                                  EAP_TLS(4)
                                }
        STATUS        current
        DESCRIPTION
            "The authentication protocol that may be used for an
            access request. Based on this attribute the
            corresponding Auth Extensions PRC must be used as
            defined under the identAuthClasses branch. For
            CHAP_MD5, and CHAP_MS, the same authChapExtTable must
            be used."
        ::= { eventHdlrAuthProtocolEntry 3}


    --
    -- Authentication Extension Tables
    --

    --
    -- AuthExtensions Base Table
```

```
      --

      authExtTable OBJECT-TYPE
          SYNTAX         SEQUENCE OF AuthExtEntry
          PIB-ACCESS     install-notify
```



```
          STATUS         current
          DESCRIPTION
              "This is an abstract PRC. This PRC can be extended by
              authentication PRCs that contain attributes specific to
              that authentication protocol. An instance of the extended
              class is created by the PEP and sent to the PDP. The PDP
              may send information back to the PEP or may uses the
              information to authenticate the PEP's access request.
              This  PRC itself should not be instantiated.

              This is a 'transient' class. Its instances are temporary
              and are deleted by the PEP after a certain time/event.
              Thus it must not be referred to by the server."

          ::= { identAuthClasses 1 }

      authExtEntry OBJECT-TYPE
          SYNTAX         AuthExtEntry
          STATUS         current
          DESCRIPTION
              "Entry oid for the AuthExtTable PRC."

          PIB-INDEX { authExtId }
          UNIQUENESS { }

          ::= { authExtTable 1 }

      AuthExtEntry ::= SEQUENCE {
          authExtId                InstanceId
      }

      authExtId  OBJECT-TYPE
          SYNTAX         InstanceId
          STATUS         current
          DESCRIPTION
              "An index to uniquely identify an instance of the
              entended provisioning class."

          ::= { authExtEntry 1 }

      --
      -- UserAuthExt Table
      --
```

```
      userAuthExtTable OBJECT-TYPE
          SYNTAX         SEQUENCE OF UserAuthExtEntry
          PIB-ACCESS     notify
          STATUS         current
          DESCRIPTION
              "This is a concrete PRC used to contain user
              authentication fields. This PRC extends the base PRC
              authExtEntry."
```



```
          ::= { identAuthClasses 2 }

      userAuthExtEntry OBJECT-TYPE
          SYNTAX         UserAuthExtEntry
          STATUS         current
          DESCRIPTION
              "Entry for the UserAuthExtTable PRC. InstanceId's for
              this extended PRC are assigned by the base PRC AuthExt
              [SPPI]."

          EXTENDS { authExtEntry }
          UNIQUENESS { }

          ::= { userAuthExtTable 1 }

      UserAuthExtEntry ::= SEQUENCE {
          userAuthExtRealm      OCTET STRING,
          userAuthExtUsername   OCTET STRING
      }


      userAuthExtRealm OBJECT-TYPE
          SYNTAX         OCTET STRING
          STATUS         current
          DESCRIPTION
              "user realm octet string."

          ::= { userAuthExtEntry 1 }

      userAuthExtUsername OBJECT-TYPE
          SYNTAX         OCTET STRING
          STATUS         current
          DESCRIPTION
              "Username octet string."

          ::= { userAuthExtEntry 2 }
```

```
      --
      -- AuthChapExt Table
      --

      authChapExtTable OBJECT-TYPE
          SYNTAX         SEQUENCE OF AuthChapExtEntry
          PIB-ACCESS     notify
          STATUS         current
          DESCRIPTION
              "This is a concrete PRC used to contain CHAP
              authentication fields. This PRC extends the PRC
              userAuthExtEntry."

          ::= { identAuthClasses 3 }
```



```
      authChapExtEntry OBJECT-TYPE
          SYNTAX         AuthChapExtEntry
          STATUS         current
          DESCRIPTION
              "Entry oid for the AuthChapExtTable PRC. InstanceId's for
              this extended PRC are assigned by the base PRC [SPPI]."

          EXTENDS { userAuthExtEntry }
          UNIQUENESS { }

          ::= { authChapExtTable 1 }

      AuthChapExtEntry::= SEQUENCE {
          authChapExtId        Unsigned32,
          authChapExtChal      OCTET STRING,
          authChapExtResp      OCTET STRING
      }

      authChapExtId OBJECT-TYPE
          SYNTAX         Unsigned32
          STATUS         current
          DESCRIPTION
              "CHAP Id field."

          ::= { authChapExtEntry 1 }

      authChapExtChal OBJECT-TYPE
          SYNTAX         OCTET STRING
          STATUS         current
          DESCRIPTION
              "CHAP Challenge octet string. The challenge is generated
              by the PEP."
```

```
        ::= { authChapExtEntry 2 }

    authChapExtResp OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
        DESCRIPTION
            "CHAP Challenge Response octet string. The challenge
            response is sent to the PDP along with the challenge."

        ::= { authChapExtEntry 3 }


    --
    -- AuthPapExt Table
    --

    authPapExtTable OBJECT-TYPE
        SYNTAX         SEQUENCE OF AuthPapExtEntry
        PIB-ACCESS     notify
```



```
        STATUS         current
        DESCRIPTION
            "This is a concrete PRC used to contain PAP
            authentication fields. This PRC extends the PRC
            userAuthExtEntry."

        ::= { identAuthClasses 4 }

    authPapExtEntry OBJECT-TYPE
        SYNTAX         AuthPapExtEntry
        STATUS         current
        DESCRIPTION
            "Entry oid for the AuthPapExtTable PRC. InstanceId's for
            this extended PRC are assigned by the base PRC [SPPI]."

        EXTENDS { userAuthExtEntry }
        UNIQUENESS { }

        ::= { authPapExtTable 1 }

    AuthPapExtEntry::= SEQUENCE {
        authPapExtPwd      OCTET STRING
    }

    authPapExtPwd OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
        DESCRIPTION
```

```
            "PAP password octet string."

        ::= { authPapExtEntry 1 }




    --
    -- AuthExtResult Table
    --

    authExtResultTable OBJECT-TYPE
        SYNTAX         SEQUENCE OF AuthExtResultEntry
        PIB-ACCESS     install
        STATUS         current
        DESCRIPTION
            "This is a concrete PRC used to contain authentication
            results. This PRC extends the base PRC authExtEntry."

        ::= { identAuthClasses 5 }

    authExtResultEntry OBJECT-TYPE
        SYNTAX         AuthExtResultEntry
        STATUS         current
```



```
        DESCRIPTION
            "Entry for the authExtResultTable PRC. InstanceId's for
            this extended PRC are assigned by the base PRC AuthExt
            [SPPI]."

        EXTENDS { authExtEntry }
        UNIQUENESS { }

        ::= { authExtResultTable 1 }

    AuthExtResultEntry ::= SEQUENCE {
        authExtResultSuccess          TruthValue
    }


    authExtResultSuccess OBJECT-TYPE
        SYNTAX         TruthValue
        STATUS         current
        DESCRIPTION
            "Set to 'true' if authentication was successful, else
            false."

        ::= { authExtResultEntry 1 }
```

```
      --
      -- AuthEapReqExt Table
      --

      authEapReqExtTable OBJECT-TYPE
          SYNTAX         SEQUENCE OF AuthEapReqExtEntry
          PIB-ACCESS     notify
          STATUS         current
          DESCRIPTION
              "This is a concrete PRC used to contain EAP
              authentication fields. This PRC extends the base PRC
              authExtEntry. The PEP uses this PRC to send EAP messages
              to the PDP."

          ::= { identAuthClasses 6 }

      authEapReqExtEntry OBJECT-TYPE
          SYNTAX         AuthEapReqExtEntry
          STATUS         current
          DESCRIPTION
              "Entry oid for the authEapReqExtTable PRC. InstanceId's
              for this extended PRC are assigned by the base PRC
              [SPPI]."

          EXTENDS { authExtEntry }
          UNIQUENESS { }

          ::= { authEapReqExtTable 1 }
```



```
      AuthEapReqExtEntry::= SEQUENCE {
          authEapReqExtSpecific    OCTET STRING
      }

      authEapReqExtSpecific OBJECT-TYPE
          SYNTAX         OCTET STRING
          STATUS         current
          DESCRIPTION
              "Opaque EAP Request octet string."

          ::= { authEapReqExtEntry 1 }


      --
      -- AuthEapRespExt Table
      --
```

```
    authEapRespExtTable OBJECT-TYPE
        SYNTAX         SEQUENCE OF AuthEapRespExtEntry
        PIB-ACCESS     install
        STATUS         current
        DESCRIPTION
            "This is a concrete PRC used to contain EAP
            authentication fields. This PRC extends the base PRC
            authExtEntry. The PDP responds using this PRC for EAP
            exchanges."

        ::= { identAuthClasses 7 }

    authEapRespExtEntry OBJECT-TYPE
        SYNTAX         AuthEapRespExtEntry
        STATUS         current
        DESCRIPTION
            "Entry oid for the authEapRespExtTable PRC. InstanceId's
            for this extended PRC are assigned by the base PRC
            [SPPI]."

        EXTENDS { authExtEntry }
        UNIQUENESS { }

        ::= { authEapRespExtTable 1 }

    AuthEapRespExtEntry::= SEQUENCE {
        authEapRespExtSpecific    OCTET STRING
    }

    authEapRespExtSpecific OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
        DESCRIPTION
            "Opaque EAP Response octet string."
```



```
        ::= { authEapRespExtEntry 1 }

    --
    -- conformance section tbd
    --

END
```

11. Application Specific RSVP Handling PIB Module

```
   --
   -- The AccessBind RSVP Handling PIB Module
   --

   ACCESSBIND-APP-RSVP-PIB PIB-DEFINITIONS ::= BEGIN

      IMPORTS
          Unsigned32, Integer32, MODULE-IDENTITY,
          MODULE-COMPLIANCE, OBJECT-TYPE, OBJECT-GROUP, pib
```

```
                FROM COPS-PR-SPPI
        InstanceId, Prid
                FROM COPS-PR-SPPI-TC
        InetAddress, InetAddressType
                FROM INET-ADDRESS-MIB;

    accessBindAppRsvpPib  MODULE-IDENTITY
        SUBJECT-CATEGORIES { all }
        LAST-UPDATED "200211032002Z"
        ORGANIZATION "IETF RAP WG"
        CONTACT-INFO "
                      Walter Weiss
                      Ellacoya Networks
                      7 Henry Clay Drive
                      Merrimack, NH 03054
                      Phone: 603-879-7364
                      E-mail: wweiss@ellacoya.com
                     "
        DESCRIPTION
            "A PIB module containing the set of classes to
            be used with the access bind PIB framework classes
            to configure RSVP specific event handlers, and
            outsource RSVP events as they occur."

        ::= { pib 5 }    -- xxx to be assigned by IANA


    --
    -- The branch OIDs in the AccessBind PIB
    --

    contextClasses OBJECT IDENTIFIER ::= {
        accessBindAppRsvpPib  1
    }
    filterClasses OBJECT IDENTIFIER ::= {
        accessBindAppRsvpPib  2
    }



    --
    -- The RSVP Filter table
```



```
    --
    rsvpFilterTable OBJECT-TYPE
        SYNTAX  SEQUENCE OF RsvpFilterEntry
        PIB-ACCESS      install
        STATUS                current
        DESCRIPTION
```

```
            "RSVP specific filter table."

        ::= { filterClasses 1 }

    rsvpFilterEntry OBJECT-TYPE
        SYNTAX  RsvpFilterEntry
        STATUS  current
        DESCRIPTION
            " RSVP specific filter table entry."

        PIB-INDEX { rsvpFilterId }
        UNIQUENESS { }

        ::= { rsvpFilterTable 1 }


    RsvpFilterEntry ::= SEQUENCE {
        rsvpFilterId                          InstanceId,
        rsvpFilterFlags                       OCTET STRING,
        rsvpFilterSendTTL                     Unsigned32,
        rsvpFilterDClassDscp          Integer32,
        rsvpFilterSessionDestAddrType InetAddressType,
        rsvpFilterSessionDestAddr     InetAddress,
        rsvpFilterSessionDestAddrMask Unsigned32,
        rsvpFilterSessionProtocol     Integer32,
        rsvpFilterSessionDestPort     Unsigned32,
        rsvpFilterSessionSrcAddrType  InetAddressType,
        rsvpFilterSessionSrcAddr              InetAddress,
        rsvpFilterSessionSrcAddrMask  Unsigned32,
        rsvpFilterSessionSrcPort              Unsigned32,
        rsvpFilterStyleValue          OCTET STRING
    }

    rsvpFilterId OBJECT-TYPE
        SYNTAX       InstanceId
        STATUS       current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies
            an instance of the class."

        ::= { rsvpFilterEntry 1 }

    rsvpFilterFlags OBJECT-TYPE
        SYNTAX       OCTET STRING
        STATUS       current
        DESCRIPTION
            "The Flags carried in the RSVP header. Currently all
```



```
            these flags should be set to zero."
```

```
    ::= { rsvpFilterEntry 2 }

rsvpFilterSendTTL OBJECT-TYPE
    SYNTAX       Unsigned32 (0..255)
    STATUS       current
    DESCRIPTION
        "The IP TTL value with which the message was sent."

    ::= { rsvpFilterEntry 3 }

rsvpFilterDClassDscp OBJECT-TYPE
    SYNTAX       Integer32 (-1| 0..63)
    STATUS       current
    DESCRIPTION
        "The DClass dscp value."

    ::= { rsvpFilterEntry 4 }

rsvpFilterSessionDestAddrType OBJECT-TYPE
    SYNTAX       InetAddressType
    STATUS       current
    DESCRIPTION
        "The address type enumeration value [INETADDR] to
        specify the type of the destination IP address."

    ::= { rsvpFilterEntry 5 }

rsvpFilterSessionDestAddr OBJECT-TYPE
    SYNTAX       InetAddress
    STATUS       current
    DESCRIPTION
        "The destination IP address."

    ::= { rsvpFilterEntry 6 }

rsvpFilterSessionDestAddrMask OBJECT-TYPE
    SYNTAX       Unsigned32
    STATUS       current
    DESCRIPTION
        "The length of a mask for the matching of the
        destination IP address.."

    ::= { rsvpFilterEntry 7 }

rsvpFilterSessionProtocol OBJECT-TYPE
    SYNTAX       Integer32 (-1 | 0..255)
    STATUS       current
    DESCRIPTION
        "The IP protocol to match against the packet's
        protocol. A value of -1 means match all."
```

```
        ::= { rsvpFilterEntry 8 }

    rsvpFilterSessionDestPort OBJECT-TYPE
        SYNTAX       Unsigned32 (0..65535)
        STATUS       current
        DESCRIPTION
            "The packet's Layer 4 destination port."

        ::= { rsvpFilterEntry 9 }

    rsvpFilterSessionSrcAddrType OBJECT-TYPE
        SYNTAX       InetAddressType
        STATUS       current
        DESCRIPTION
            "The address type enumeration value [INETADDR] to
            specify the type of the source IP address."

        ::= { rsvpFilterEntry 10 }

    rsvpFilterSessionSrcAddr OBJECT-TYPE
        SYNTAX       InetAddress
        STATUS       current
        DESCRIPTION
            "The source IP address."

        ::= { rsvpFilterEntry 11 }

    rsvpFilterSessionSrcAddrMask OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
            "The length of a mask for the matching of the source
            IP address."

        ::= { rsvpFilterEntry 12 }

    rsvpFilterSessionSrcPort OBJECT-TYPE
        SYNTAX       Unsigned32 (0..65535)
        STATUS       current
        DESCRIPTION
            "The packet's Layer 4 source port."

        ::= { rsvpFilterEntry 13 }

    rsvpFilterStyleValue OBJECT-TYPE
        SYNTAX       OCTET STRING
        STATUS       current
        DESCRIPTION
```

```
            "The RSVP packet's Style value."

        ::= { rsvpFilterEntry 14 }
```



```
    --
    -- RSVP Common Context Data
    --

    ctxtRsvpTable OBJECT-TYPE
        SYNTAX  SEQUENCE OF CtxtRsvpEntry
        PIB-ACCESS      notify
        STATUS                current
        DESCRIPTION
            ""

        ::= { contextClasses 1 }

    ctxtRsvpEntry OBJECT-TYPE
        SYNTAX  CtxtRsvpEntry
        STATUS  current
        DESCRIPTION
            ""

        PIB-INDEX { ctxtRsvpId }
        UNIQUENESS { }

        ::= { ctxtRsvpTable 1 }

    CtxtRsvpEntry ::= SEQUENCE {
        ctxtRsvpId                    InstanceId,
              ctxtRsvpMsgType         INTEGER,
        ctxtRsvpFlags                 OCTET STRING,
        ctxtRsvpSendTTL               Unsigned32,
        ctxtRsvpInIntfId              Unsigned32,
        ctxtRsvpInIntfAddrType        InetAddressType,
        ctxtRsvpInIntfAddr            InetAddress,
        ctxtRsvpOutIntfId             Unsigned32,
        ctxtRsvpOutIntfAddrType       InetAddressType,
        ctxtRsvpOutIntfAddr           InetAddress
    }

    ctxtRsvpId OBJECT-TYPE
        SYNTAX       InstanceId
        STATUS       current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies
            an instance of the class."
```

```
        ::= { ctxtRsvpEntry 1 }

    ctxtRsvpMsgType OBJECT-TYPE
        SYNTAX       INTEGER {
                                Path (1),
                                PathErr (2),
                                Resv (3),
                                ResvErr (4)
        }
```



```
        STATUS       current
        DESCRIPTION
            "The RSVP message type."

        ::= { ctxtRsvpEntry 2 }

    ctxtRsvpFlags OBJECT-TYPE
        SYNTAX       OCTET STRING
        STATUS       current
        DESCRIPTION
            "The RSVP flags contained in the message header.
            They are currently undefined and should be set to
            zero."

        ::= { ctxtRsvpEntry 3 }

    ctxtRsvpSendTTL OBJECT-TYPE
        SYNTAX       Unsigned32 (0..255)
        STATUS       current
        DESCRIPTION
            "The IP TTL value."

        ::= { ctxtRsvpEntry 4 }

    ctxtRsvpInIntfId OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
            "The Interface Id."

        ::= { ctxtRsvpEntry 5 }

    ctxtRsvpInIntfAddrType OBJECT-TYPE
        SYNTAX       InetAddressType
        STATUS       current
        DESCRIPTION
            "The address type enumeration value [INETADDR] to
            specify the type of the In Interface IP address."
```

```
        ::= { ctxtRsvpEntry 6 }

    ctxtRsvpInIntfAddr OBJECT-TYPE
        SYNTAX       InetAddress
        STATUS       current
        DESCRIPTION
            "The In Interface IP address."

        ::= { ctxtRsvpEntry 7 }

    ctxtRsvpOutIntfId OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
```



```
            "The Out Interface Id."

        ::= { ctxtRsvpEntry 8 }

    ctxtRsvpOutIntfAddrType OBJECT-TYPE
        SYNTAX       InetAddressType
        STATUS       current
        DESCRIPTION
            "The address type enumeration value [INETADDR] to
            specify the type of the Out Interface IP address."

        ::= { ctxtRsvpEntry 9 }

    ctxtRsvpOutIntfAddr OBJECT-TYPE
        SYNTAX       InetAddress
        STATUS       current
        DESCRIPTION
            "The Out Interface IP address."

        ::= { ctxtRsvpEntry 10 }


    --
    -- RSVP Path Context Data
    --

    ctxtRsvpPathTable OBJECT-TYPE
        SYNTAX  SEQUENCE OF CtxtRsvpPathEntry
        PIB-ACCESS      notify
        STATUS               current
        DESCRIPTION
            ""
```

```
        ::= { contextClasses 2 }

    ctxtRsvpPathEntry OBJECT-TYPE
        SYNTAX  CtxtRsvpPathEntry
        STATUS  current
        DESCRIPTION
            ""

        PIB-INDEX { ctxtRsvpPathId }
        UNIQUENESS { }

        ::= { ctxtRsvpPathTable 1 }

    CtxtRsvpPathEntry ::= SEQUENCE {
           ctxtRsvpPathId             InstanceId,
           ctxtRsvpPathTokenRate      Unsigned32
    }

    ctxtRsvpPathId OBJECT-TYPE
        SYNTAX       InstanceId
```



```
        STATUS       current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies
            an instance of the class."

        ::= { ctxtRsvpPathEntry 1 }

    ctxtRsvpPathTokenRate OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
            "The token bucket rate for the TSPEC."

        ::= { ctxtRsvpPathEntry 2 }

    --
    -- RSVP PathErr Context Data
    --

    ctxtRsvpPathErrTable OBJECT-TYPE
        SYNTAX  SEQUENCE OF CtxtRsvpPathErrEntry
        PIB-ACCESS      notify
        STATUS               current
        DESCRIPTION
            ""

        ::= { contextClasses 3 }
```

```
      ctxtRsvpPathErrEntry OBJECT-TYPE
          SYNTAX  CtxtRsvpPathErrEntry
          STATUS  current
          DESCRIPTION
              ""

          PIB-INDEX { ctxtRsvpPathErrId }
          UNIQUENESS { }

          ::= { ctxtRsvpPathErrTable 1 }

      CtxtRsvpPathErrEntry ::= SEQUENCE {
             ctxtRsvpPathErrId          InstanceId,
          ctxtRsvpPathErrTokenRate      Unsigned32,
          ctxtRsvpPathErrErrorAddrType  InetAddressType,
          ctxtRsvpPathErrErrorAddr      InetAddress,
          ctxtRsvpPathErrErrorCode      Unsigned32,
          ctxtRsvpPathErrErrorValue     Unsigned32
      }

      ctxtRsvpPathErrId OBJECT-TYPE
          SYNTAX       InstanceId
          STATUS       current
          DESCRIPTION
              "An arbitrary integer index that uniquely identifies
```



```
              an instance of the class."

          ::= { ctxtRsvpPathErrEntry 1 }

      ctxtRsvpPathErrTokenRate OBJECT-TYPE
          SYNTAX       Unsigned32
          STATUS       current
          DESCRIPTION
              "The token bucket rate for the TSPEC."

          ::= { ctxtRsvpPathErrEntry 2 }

      ctxtRsvpPathErrErrorAddrType OBJECT-TYPE
          SYNTAX       InetAddressType
          STATUS       current
          DESCRIPTION
              "The address type IP address in error."

          ::= { ctxtRsvpPathErrEntry 3 }

      ctxtRsvpPathErrErrorAddr OBJECT-TYPE
          SYNTAX       InetAddress
          STATUS       current
```

```
            DESCRIPTION
                "The Error IP address."

            ::= { ctxtRsvpPathErrEntry 4 }

        ctxtRsvpPathErrErrorCode OBJECT-TYPE
            SYNTAX       Unsigned32
            STATUS       current
            DESCRIPTION
                "The RSVP error code."

            ::= { ctxtRsvpPathErrEntry 5 }

        ctxtRsvpPathErrErrorValue OBJECT-TYPE
            SYNTAX       Unsigned32
            STATUS       current
            DESCRIPTION
                "The RSVP error value."

            ::= { ctxtRsvpPathErrEntry 6 }

        --
        -- RSVP Resv Context Data
        --

        ctxtRsvpResvTable OBJECT-TYPE
            SYNTAX  SEQUENCE OF CtxtRsvpResvEntry
            PIB-ACCESS      notify
            STATUS                current
            DESCRIPTION
```



```
                ""

            ::= { contextClasses 4 }

        ctxtRsvpResvEntry OBJECT-TYPE
            SYNTAX  CtxtRsvpResvEntry
            STATUS  current
            DESCRIPTION
                ""

            PIB-INDEX { ctxtRsvpResvId }
            UNIQUENESS { }

            ::= { ctxtRsvpResvTable 1 }

        CtxtRsvpResvEntry ::= SEQUENCE {
               ctxtRsvpResvId     InstanceId,
               ctxtRsvpResvFSpecGrp       TagReferenceId,
```

```
        ctxtRsvpResvSvcType        INTEGER,
        ctxtRsvpResvTokenRate      Unsigned32
    }

    ctxtRsvpResvId OBJECT-TYPE
        SYNTAX       InstanceId
        STATUS       current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies
            an instance of the class."

        ::= { ctxtRsvpResvEntry 1 }

    ctxtRsvpResvFSpecGrp OBJECT-TYPE
        SYNTAX       TagReferenceId
        PIB-TAG { ctxtRsvpFilterSpecTagId }
        STATUS       current
        DESCRIPTION
            "Identifies a group of Filter Spec entries."

        ::= { ctxtRsvpResvEntry 2 }

    ctxtRsvpResvSvcType OBJECT-TYPE
        SYNTAX       INTEGER {
                                Controlled_Load(1),
                                Guaranteed(2)
                             }
        STATUS       current
        DESCRIPTION
            "An enum describing the type of service."

        ::= { ctxtRsvpResvEntry 3 }

    ctxtRsvpResvTokenRate OBJECT-TYPE
        SYNTAX       Unsigned32
```



```
        STATUS       current
        DESCRIPTION
            "The token bucket rate for the TSPEC."

        ::= { ctxtRsvpResvEntry 4 }

    --
    -- RSVP ResvErr Context Data
    --

    ctxtRsvpResvErrTable OBJECT-TYPE
        SYNTAX  SEQUENCE OF CtxtRsvpResvErrEntry
        PIB-ACCESS      notify
```

```
        STATUS               current
        DESCRIPTION
            ""

        ::= { contextClasses 5 }

    ctxtRsvpResvErrEntry OBJECT-TYPE
        SYNTAX  CtxtRsvpResvErrEntry
        STATUS  current
        DESCRIPTION
            ""

        PIB-INDEX { ctxtRsvpResvErrId }
        UNIQUENESS { }

        ::= { ctxtRsvpResvErrTable 1 }

    CtxtRsvpResvErrEntry ::= SEQUENCE {
        ctxtRsvpResvErrId                    InstanceId,
        ctxtRsvpResvErrFSpecGrp       TagReferenceId,
        ctxtRsvpResvErrSvcType               INTEGER,
        ctxtRsvpResvErrTokenRate             Unsigned32,
        ctxtRsvpResvErrErrorAddrType  InetAddressType,
        ctxtRsvpResvErrErrorAddr             InetAddress,
        ctxtRsvpResvErrErrorCode      Unsigned32,
        ctxtRsvpResvErrErrorValue     Unsigned32
    }

    ctxtRsvpResvErrId OBJECT-TYPE
        SYNTAX       InstanceId
        STATUS       current
        DESCRIPTION
            "An arbitrary integer index that uniquely identifies
            an instance of the class."

        ::= { ctxtRsvpResvErrEntry 1 }

    ctxtRsvpResvErrFSpecGrp OBJECT-TYPE
        SYNTAX       TagReferenceId
        PIB-TAG { ctxtRsvpFilterSpecTagId }
```



```
        STATUS       current
        DESCRIPTION
            "Identifies a group of Filter Spec entries."

        ::= { ctxtRsvpResvErrEntry 2 }

    ctxtRsvpResvErrSvcType OBJECT-TYPE
        SYNTAX       INTEGER {
```

```
                            Controlled_Load(1),
                            Guaranteed(2)
                        }
        STATUS       current
        DESCRIPTION
            "An enum describing the type of service."

        ::= { ctxtRsvpResvErrEntry 3 }

    ctxtRsvpResvErrTokenRate OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
            "The token bucket rate for the TSPEC."

        ::= { ctxtRsvpResvErrEntry 4 }

    ctxtRsvpResvErrErrorAddrType OBJECT-TYPE
        SYNTAX       InetAddressType
        STATUS       current
        DESCRIPTION
            "The address type IP address in error."

        ::= { ctxtRsvpResvErrEntry 5 }

    ctxtRsvpResvErrErrorAddr OBJECT-TYPE
        SYNTAX       InetAddress
        STATUS       current
        DESCRIPTION
            "The Error IP address."

        ::= { ctxtRsvpResvErrEntry 6 }

    ctxtRsvpResvErrErrorCode OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
            "The RSVP error code."

        ::= { ctxtRsvpResvErrEntry 7 }

    ctxtRsvpResvErrErrorValue OBJECT-TYPE
        SYNTAX       Unsigned32
        STATUS       current
        DESCRIPTION
```



```
            "The RSVP error value."

        ::= { ctxtRsvpResvErrEntry 8 }
```

```
--
-- RSVP Filter Spec Context Data
--

ctxtRsvpFilterSpecTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF CtxtRsvpFilterSpecEntry
    PIB-ACCESS      notify
    STATUS               current
    DESCRIPTION
        ""

    ::= { contextClasses 6 }

ctxtRsvpFilterSpecEntry OBJECT-TYPE
    SYNTAX  CtxtRsvpFilterSpecEntry
    STATUS  current
    DESCRIPTION
        ""

    PIB-INDEX { ctxtRsvpFilterSpecId }
    UNIQUENESS { }

    ::= { ctxtRsvpFilterSpecTable 1 }

CtxtRsvpFilterSpecEntry::= SEQUENCE {
    ctxtRsvpFilterSpecId          InstanceId,
    ctxtRsvpFilterSpecTagId       TagId,
    ctxtRsvpFilterSpecAddrType    InetAddressType,
    ctxtRsvpFilterSpecAddr                InetAddress,
    ctxtRsvpFilterSpecPort                Unsigned32
}

ctxtRsvpFilterSpecId OBJECT-TYPE
    SYNTAX       InstanceId
    STATUS       current
    DESCRIPTION
        "An arbitrary integer index that uniquely identifies
        an instance of the class."

    ::= { ctxtRsvpFilterSpecEntry 1 }

ctxtRsvpFilterSpecTagId OBJECT-TYPE
    SYNTAX       TagId
    STATUS       current
    DESCRIPTION
        "Identifies the group of Filter Spec PRIs that this
        PRI belongs to."

    ::= { ctxtRsvpFilterSpecEntry 2 }
```

```
   ctxtRsvpFilterSpecAddrType OBJECT-TYPE
       SYNTAX       InetAddressType
       STATUS       current
       DESCRIPTION
           "The address type enumeration value [INETADDR] to
           specify the type of the IP address."

       ::= { ctxtRsvpFilterSpecEntry 3 }

   ctxtRsvpFilterSpecAddr OBJECT-TYPE
       SYNTAX       InetAddress
       STATUS       current
       DESCRIPTION
           "The Filter Spec IP address."

       ::= { ctxtRsvpFilterSpecEntry 4 }

   ctxtRsvpFilterSpecPort OBJECT-TYPE
       SYNTAX       Unsigned32 (0..65535)
       STATUS       current
       DESCRIPTION
           "The packet's Layer 4 destination port."

       ::= { ctxtRsvpFilterSpecEntry 5 }


END
```

12. Application Specific Dialup Handling PIB Module

```
   --
   -- The AccessBind Dialup Application PIB Module
   --

   ACCESSBIND-APP-DIALUP-PIB PIB-DEFINITIONS ::= BEGIN

      IMPORTS
          Unsigned32, Integer32, MODULE-IDENTITY,
          MODULE-COMPLIANCE, OBJECT-TYPE, OBJECT-GROUP, pib
                  FROM COPS-PR-SPPI
          InstanceId
                  FROM COPS-PR-SPPI-TC
          InetAddress, InetAddressType
                  FROM INET-ADDRESS-MIB;

      accessBindAppDialupPib  MODULE-IDENTITY
          SUBJECT-CATEGORIES { all }
          LAST-UPDATED "200211032002Z"
          ORGANIZATION "IETF RAP WG"
          CONTACT-INFO "
                        Walter Weiss
                        Ellacoya Networks
                        7 Henry Clay Drive
                        Merrimack, NH 03054
                        Phone: 603-879-7364
                        E-mail: wweiss@ellacoya.com
                       "
          DESCRIPTION
              "A PIB module containing the set of classes to
              be used with the access bind PIB framework classes
              to configure dialup event handlers, and outsource
              dialup events as they occur."

          ::= { pib 5 }    -- xxx to be assigned by IANA


      --
      -- The branch OIDs in the AccessBind PIB
      --

      contextClasses OBJECT IDENTIFIER ::= {
          accessBindAppDialupPib  1
      }

      --
```

```
      -- CtxtDialupInterface Table
      --

      ctxtDialupInterfaceTable OBJECT-TYPE
          SYNTAX         SEQUENCE OF CtxtDialupInterfaceEntry
          PIB-ACCESS     notify
```



```
          STATUS         current
          DESCRIPTION
              "Dialup Interface context data."

          ::= { contextClasses 1 }

      ctxtDialupInterfaceEntry OBJECT-TYPE
          SYNTAX         CtxtDialupInterfaceEntry
          STATUS         current
          DESCRIPTION
              "Entry oid of the ctxtDialupInterfaceTable PRC."

          PIB-INDEX { ctxtDialupInterfaceId }
          UNIQUENESS { }

          ::= { ctxtDialupInterfaceTable 1 }

      CtxtDialupInterfaceEntry::= SEQUENCE {
          ctxtDialupInterfaceId                InstanceId,
          ctxtDialupInterfaceNASPort           Integer32,
          ctxtDialupInterfaceNASPortId         OCTET STRING,
          ctxtDialupInterfaceNASPortType       INTEGER,
          ctxtDialupInterfaceCalledStationId   OCTET STRING,
          ctxtDialupInterfaceCallingStationId  OCTET STRING,
          ctxtDialupInterfaceConnectInfo       OCTET STRING
      }

      ctxtDialupInterfaceId  OBJECT-TYPE
          SYNTAX         InstanceId
          STATUS         current
          DESCRIPTION
              "An index to uniquely identify an instance of this
              provisioning class."

          ::= { ctxtDialupInterfaceEntry 1 }


      ctxtDialupInterfaceNASPort  OBJECT-TYPE
          SYNTAX         Integer32
          STATUS         current
          DESCRIPTION
              "This Attribute indicates the physical port number
```

```
            of the NAS which is authenticating the user.  It is
            only used in Access-Request packets.  Note that this
            is using 'port' in its sense of a physical
            connection on the NAS, not in the sense of a TCP or
            UDP port number."

        ::= { ctxtDialupInterfaceEntry 2 }


    ctxtDialupInterfaceNASPortId  OBJECT-TYPE
        SYNTAX         OCTET STRING
```



```
        STATUS         current
        DESCRIPTION
            "This Attribute contains a text string which
            identifies the port of the NAS which is
            authenticating the user. It is only used in
            Access-Request and Accounting-Request packets.  Note
            that this is using 'port' in its sense of a physical
            connection on the NAS, not in the sense of a TCP or
            UDP port number. "

        ::= { ctxtDialupInterfaceEntry 3 }

    ctxtDialupInterfaceNASPortType  OBJECT-TYPE
        SYNTAX  INTEGER {
                    radAsync(0),
                    radSync(1),
                    radIsdnSync(2),
                    radIsdnAsyncV120(3),
                    radIsdnAsyncV110(4),
                    radVirtual(5),
                    radPIAFS(6),
                    radHdlcClearChannel(7),
                    radX25(8),
                    radX75(9),
                    radG3Fax(10),
                    radSDSL(11),
                    radAdslCAP(12),
                    radAdslDMT(13),
                    radIdsl(14),
                    radEthernet(15),
                    radXdsl(16),
                    radCable(17),
                    radWirelessOther(18),
                    radWirelessIEEE80211(19)
        }
        STATUS         current
```

DESCRIPTION
"This Attribute indicates the type of the physical port of the NAS which is authenticating the user. It can be used instead of or in addition to the radNasPort (5) attribute. It is only used in Access-Request packets. Either radNasPort (5) or radNasPortType or both SHOULD be present in an Access-Request packet, if the NAS differentiates among its ports.

A value of 'radAsync(0)' indicates Async.

A value of 'radSync(1)' indicates Sync.

A value of 'radIsdnSync(2)' indicates ISDN Sync.

A value of 'radIsdnAsyncV120(3)' indicates ISDN

Async V.120.

A value of 'radIsdnAsyncV110(4)' indicates ISDN Async V.110.

A value of 'radVirtual(5)' indicates Virtual. Virtual refers to a connection to the NAS via some transport protocol, instead of through a physical port. For example, if a user telnetted into a NAS to authenticate himself as an Outbound-User, the Access-Request might include radNasPortType = Virtual as a hint to the RADIUS server that the user was not on a physical port.

A value of 'radPIAFS(6)' indicates PIAFS. PIAFS is a form of wireless ISDN commonly used in Japan, and stands for PHS (Personal Handyphone System) Internet Access Forum Standard (PIAFS).

A value of 'radHdlcClearChannel(7)' indicates HDLC Clear Channel.

A value of 'radX25(8)' indicates X.25.

A value of 'radX75(9)' indicates X.75.

A value of 'radG3Fax(10)' indicates G.3 Fax.

A value of 'radSDSL(11)' indicates SDSL _ Symmetric DSL.

```
            A value of 'radAdslCAP(12)' indicates ADSL-CAP -
            Asymmetric DSL, Carrierless Amplitude Phase
            Modulation.

            A value of 'radAdslDMT(13)' indicates ADSL-DMT -
            Asymmetric DSL, Discrete Multi-Tone.

            A value of 'radIdsl(14)' indicates IDSL _ ISDN
            Digital Subscriber Line.

            A value of 'radEthernet(15)' indicates Ethernet.

            A value of 'radXdsl(16)' indicates xDSL - Digital
            Subscriber Line of unknown type.

            A value of 'radCable(17)' indicates Cable.

            A value of 'radWirelessOther(18)' indicates
            Wireless - Other.

            A value of 'radWirelessIEEE80211(19)' indicates
            Wireless - IEEE 802.11."
```



```
        ::= { ctxtDialupInterfaceEntry 4 }


    ctxtDialupInterfaceCalledStationId  OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
        DESCRIPTION
            "This Attribute allows the NAS to send in the
            Access-Request packet the phone number that the user
            called, using Dialed Number Identification (DNIS) or
            similar technology.  Note that this may be different
            from the phone number the call comes in on.  It is
            only used in Access-Request packets."

        ::= { ctxtDialupInterfaceEntry 5 }

    ctxtDialupInterfaceCallingStationId  OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
        DESCRIPTION
            "This Attribute allows the NAS to send in the
            Access-Request packet the phone number that the user
            is calling from, using Dialed Number Identification
            (DNIS) or similar technology.  Note that this may be
            different from the phone number called.  It is only
```

```
            used in Access-Request packets."

        ::= { ctxtDialupInterfaceEntry 6 }

    ctxtDialupInterfaceConnectInfo  OBJECT-TYPE
        SYNTAX         OCTET STRING
        STATUS         current
        DESCRIPTION
            "This Attribute allows the NAS to send in the
            Access-Request packet the phone number that the call
            came from, using Automatic Number Identification
            (ANI) or similar technology.  It is only used in
            Access-Request packets."

        ::= { ctxtDialupInterfaceEntry 7 }




    ---
    --- CtxtDialupInterfaceFramedProtocol Table
    ---

    ctxtDialupIfFramedProtocolTable OBJECT-TYPE
        SYNTAX         SEQUENCE OF
                           CtxtDialupIfFramedProtocolEntry
        PIB-ACCESS     notify
```



```
        STATUS         current
        DESCRIPTION
            "."

        ::= { contextClasses 2 }

    ctxtDialupIfFramedProtocolEntry OBJECT-TYPE
        SYNTAX         CtxtDialupIfFramedProtocolEntry
        STATUS         current
        DESCRIPTION
            "Entry oid of the ctxtDialupIfFramedProtocolTable
            PRC."

        PIB-INDEX { ctxtDialupIfFramedProtocolId }
        UNIQUENESS { }

        ::= { ctxtDialupIfFramedProtocolTable 1 }

    CtxtDialupIfFramedProtocolEntry ::= SEQUENCE {
        ctxtDialupIfFramedProtocolId           InstanceId,
        ctxtDialupIfFramedProtocolProt         INTEGER,
```

```
        ctxtDialupIfFramedProtocolMTU          Integer32,
        ctxtDialupIfFramedProtocolCompression  INTEGER,
        ctxtDialupIfFramedProtocolPortLimit    Unsigned32,
        ctxtDialupIfFramedProtocolIpAddress    InetAddress,
        ctxtDialupIfFramedProtocolIpNetmask    InetAddress
    }

    ctxtDialupIfFramedProtocolId OBJECT-TYPE
        SYNTAX         InstanceId
        STATUS         current
        DESCRIPTION
            "An index to uniquely identify an instance of this
            provisioning class."

        ::= { ctxtDialupIfFramedProtocolEntry 1 }


    ctxtDialupIfFramedProtocolProt  OBJECT-TYPE
        SYNTAX  INTEGER {
                    radPPP(1),
                    radSLIP(2),
                    radARAP(3),
                    radGandalf(4),
                    radXylogics(5),
                    radX75Synchronous(6)
        }
        STATUS         current
        DESCRIPTION
            "This Attribute indicates the framing to be used for
             framed access. It MAY be used in both Access-
             Request and Access-Accept packets.
```



```
             A value of 'radPPP(1)' represents PPP.

             A value of 'radSLIP(2)' represents SLIP.

             A value of 'radARAP(3)' represents AppleTalk Remote
             Access Protocol (ARAP).

             A value of 'radGandalf(4)' represents Gandalf
             proprietary SingleLink/MultiLink protocol.

             A value of 'radXylogics(5)' represents Xylogics
             proprietary IPX/SLIP.

             A value of 'radX75Synchronous(6)' represents X.75
             Synchronous."
```

```
        ::= { ctxtDialupIfFramedProtocolEntry 2 }


    ctxtDialupIfFramedProtocolMTU  OBJECT-TYPE
        SYNTAX         Integer32
        STATUS         current
        DESCRIPTION
            "This Attribute indicates the Maximum Transmission
            Unit to be configured for the user, when it is not
            negotiated by some other means (such as PPP).  It
            MAY be used in Access-Accept packets.  It MAY be
            used in an Access-Request packet as a hint by the
            NAS to the server that it would prefer that value,
            but the server is not required to honor the hint."

        ::= { ctxtDialupIfFramedProtocolEntry 3 }


    ctxtDialupIfFramedProtocolCompression  OBJECT-TYPE
        SYNTAX         INTEGER {
                           radNone(0),
                           radVJ(1),
                           radIPXheader(2),
                           radStacLZS(3)
        }
        STATUS         current
        DESCRIPTION
            "This Attribute indicates a compression protocol to
            be used for the link.  It MAY be used in Access-
            Accept packets.  It MAY be used in an Access-Request
            packet as a hint to the server that the NAS would
            prefer to use that compression, but the server is
            not required to honor the hint.

            More than one compression protocol Attribute MAY be
            sent. It is the responsibility of the NAS to apply
            the proper compression protocol to appropriate link
```



```
            traffic.

            A value of 'radNone(0)' indicates None.

            A value of 'radVJ(1)' indicates VJ TCP/IP header
            compression.

            A value of 'radIPXheader(2)' indicates IPX header
            compression.

            A value of 'radStacLZS(3)' indicates Stac-LZS
```

```
            compression."

        ::= { ctxtDialupIfFramedProtocolEntry 4 }


    ctxtDialupIfFramedProtocolPortLimit  OBJECT-TYPE
        SYNTAX         Unsigned32
        STATUS         current
        DESCRIPTION
            "This Attribute sets the maximum number of ports to
            be provided to the user by the NAS.  This Attribute
            MAY be sent by the server to the client in an
            Access-Accept packet.  It is intended for use in
            conjunction with Multilink PPP [10] or similar uses.
            It MAY also be sent by the NAS to the server as a
            hint that that many ports are desired for use, but
            the server is not required to honor the hint."

        ::= { ctxtDialupIfFramedProtocolEntry 5 }


    ctxtDialupIfFramedProtocolIpAddress  OBJECT-TYPE
        SYNTAX         InetAddress
        STATUS         current
        DESCRIPTION
            "This Attribute indicates the address to be
            configured for the user.  It MAY be used in Access-
            Accept packets. It MAY be used in an Access-Request
            packet as a hint by the NAS to the server that it
            would prefer that address, but the server is not
            required to honor the hint."

        ::= { ctxtDialupIfFramedProtocolEntry 6 }


    ctxtDialupIfFramedProtocolIpNetmask  OBJECT-TYPE
        SYNTAX         InetAddress
        STATUS         current
        DESCRIPTION
            "This Attribute indicates the IP netmask to be
            configured for the user when the user is a router to
            a network.  It MAY be used in Access-Accept packets.
```



```
            It MAY be used in an Access-Request packet as a hint
            by the NAS to the server that it would prefer that
            netmask, but the server is not required to honor the
            hint."

        ::= { ctxtDialupIfFramedProtocolEntry 7 }
```

```
---
--- CtxtDialupIfLoginService Table
---

ctxtDialupIfLoginServiceTable OBJECT-TYPE
    SYNTAX         SEQUENCE OF CtxtDialupIfLoginServiceEntry
    PIB-ACCESS     notify
    STATUS         current
    DESCRIPTION
        "Base class."

    ::= { contextClasses 3 }


ctxtDialupIfLoginServiceEntry OBJECT-TYPE
    SYNTAX         CtxtDialupIfLoginServiceEntry
    STATUS         current
    DESCRIPTION
        "Entry oid of the ctxtDialupIfLoginServiceTable
        PRC."

    PIB-INDEX { ctxtDialupIfLoginServiceId }
    UNIQUENESS { }

    ::= { ctxtDialupIfLoginServiceTable 1 }


CtxtDialupIfLoginServiceEntry::= SEQUENCE {
    ctxtDialupIfLoginServiceId       InstanceId,
    ctxtDialupIfLoginServiceIpHost   InetAddress
}

ctxtDialupIfLoginServiceId OBJECT-TYPE
    SYNTAX         InstanceId
    STATUS         current
    DESCRIPTION
        "An index to uniquely identify an instance of this
        provisioning class."

    ::= { ctxtDialupIfLoginServiceEntry 1 }


ctxtDialupIfLoginServiceIpHost OBJECT-TYPE
    SYNTAX         InetAddress
```



```
    STATUS         current
```

```
    DESCRIPTION
        "."

    ::= { ctxtDialupIfLoginServiceEntry 2 }



---
--- CtxtDialupIfLoginLat Table (Extends
--- CtxtDialupIfLoginService)
---

ctxtDialupIfLoginLatTable OBJECT-TYPE
    SYNTAX         SEQUENCE OF CtxtDialupIfLoginLatEntry
    PIB-ACCESS     notify
    STATUS         current
    DESCRIPTION
        "Extended class."

    ::= { contextClasses 4 }


ctxtDialupIfLoginLatEntry OBJECT-TYPE
    SYNTAX         CtxtDialupIfLoginLatEntry
    STATUS         current
    DESCRIPTION
        "Entry oid of the ctxtDialupIfLoginLatTable PRC."
    EXTENDS { ctxtDialupIfLoginServiceEntry }
    UNIQUENESS { }

    ::= { ctxtDialupIfLoginLatTable 1 }


CtxtDialupIfLoginLatEntry::= SEQUENCE {
    ctxtDialupIfLoginLatService   OCTET STRING,
    ctxtDialupIfLoginLatNode      OCTET STRING,
    ctxtDialupIfLoginLatGroup     OCTET STRING,
    ctxtDialupIfLoginLatPort      OCTET STRING
}


ctxtDialupIfLoginLatService  OBJECT-TYPE
    SYNTAX         OCTET STRING
    STATUS         current
    DESCRIPTION
        "."

    ::= { ctxtDialupIfLoginLatEntry 1 }


ctxtDialupIfLoginLatNode  OBJECT-TYPE
    SYNTAX         OCTET STRING
```

```
          STATUS         current
          DESCRIPTION
              "."

          ::= { ctxtDialupIfLoginLatEntry 2 }


      ctxtDialupIfLoginLatGroup  OBJECT-TYPE
          SYNTAX         OCTET STRING
          STATUS         current
          DESCRIPTION
              "."

          ::= { ctxtDialupIfLoginLatEntry 3 }


      ctxtDialupIfLoginLatPort  OBJECT-TYPE
          SYNTAX         OCTET STRING
          STATUS         current
          DESCRIPTION
              "."

          ::= { ctxtDialupIfLoginLatEntry 4 }


   END
```

13. Security Considerations

A COPS client-type implemented within the framework outlined in this document necessarily transmits sensitive authentication credentials between the PEP and the PDP. The COPS protocol provides optional message level security for authentication, replay protection, and message integrity for communications occurring between the PEP and the PDP by the use of the COPS Message Integrity Object [COPS]. Additionally, COPS optionally reuses existing protocols for security such as IPSEC [IPSEC] or TLS to authenticate and secure COPS communications. Careful consideration should be given to using these mechanisms to reduce the probability of compromising authentication credentials. Furthermore, using these mechanisms cannot protect communication between an external authentication server and the PDP. So, when the PDP acts a proxy for an authentication server, consideration must be given to securing communications between the PDP and the authentication server as well. A discussion of applicable security techniques would be specific to any given authentication protocol and is outside the scope of this document.

14. References

[MODEL] Y. Bernet, S. Blake, A. Smith, D. Grossman, "An Informal Management Model for Diffserv Routers," draft-ietf-diffserv-model-06.txt, February 2002.

[DSPIB] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, C. Bell, A. Smith, F. Reichmeyer, "Differentiated Services Quality of Service Policy Information Base," draft-ietf-diffserv-pib-03.txt, March 2, 2001.

[FWPIB] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, R. Sahita, A. Smith, F. Reichmeyer, _Framework Policy Information Base," draft-ietf-rap-frameworkpib-04.txt, March 1, 2001.

[AUTH] B. Lloyd, W. Simpson, _PPP Authentication Protocols,_ RFC 1334, October 1992.

[CHAP] W. Simpson, "PPP Challenge Handshake Authentication Protocol (CHAP)", RFC 1994, August 1996.

[EAP] L. Blunk, J. Vollbrecht, _PPP Extensible Authentication Protocol (EAP)_, RFC 2284, March 1998.

[NAI] B. Aboba, M. Beadles, _The Network Access Identifier,_ RFC 2486, January 1999.

[IPSEC] R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, August 1995.

[COPS] D. Durham, et al., "The COPS (Common Open Policy Service) Protocol", RFC 2748, January 2000.

[COPSPR] K. Chan, et al., "COPS Usage for Policy Provisioning (COPS-PR)", RFC 3084, March 2001.

[RSVP] R. Braden, et al., " Resource ReSerVation Protocol (RSVP) _ Version 1 Functional Specification ", September 1997.

15. Author Information and Acknowledgments

Walter Weiss
Ellacoya Networks
7 Henry Clay Drive
Merrimack, NH 03054
Phone: +1 603 879 7364
E-mail: wweiss@ellacoya.com

John Vollbrecht
Interlink Networks, Inc.
775 Technology Drive, Suite 200
Ann Arbor, MI 48108
Phone: +1 734 821 1205
E-Mail: jrv@interlinknetworks.com

David Spence
Interlink Networks, Inc.
775 Technology Drive, Suite 200
Ann Arbor, MI 48108
Phone: +1 734 821 1203
E-Mail: dspence@interlinknetworks.com

David Rago
Interlink Networks, Inc.
775 Technology Drive, Suite 200
Ann Arbor, MI 48108
Phone: +1 734 821 1241
E-Mail: drago@interlinknetworks.com

Freek Dijkstra
Physics and Astronomy Department
Utrecht University
Princetonplein 5
3584 CC Utrecht
The Netherlands
Phone: +31 30 2537724
Email: F.Dijkstra@phys.uu.nl

Cees de Laat
Faculty of Science, Informatics Institute,
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands
Phone: +31 20 5257590
E-Mail: delaat@science.uva.nl

Leon Gommans
Enterasys Networks EMEA,
Kerkplein 24
2841 XM Moordrecht
The Netherlands

Phone: +31 182 379278
E-Mail: leon.gommans@enterasys.com

Amol Kulkarni
Intel
2111 NE 25th Avenue
Hillsboro, OR 97124
Phone: 503.712.1168
E-Mail: Amol.Kulkarni@intel.com

Ravi Sahita
Intel
2111 NE 25th Avenue
Hillsboro, OR 97124
Phone: 503.712.1554
E-Mail: Ravi.Sahita@intel.com

Kwok Ho Chan
Nortel Networks
600 Technology Park Drive
Billerica, MA 01821 USA
Phone: +1 978 288 8175
E-Mail: khchan@nortelnetworks.com