# Physical Security Interoperability Alliance
# Service Model
# Version 1.0
# Revision 1.2
# 19 February 2009

| Revision History | Description | Date | By |
|---|---|---|---|
| Version 1.0 Rev 0.1 | Initial Draft | January 15, 2009 | Frank Yeh |
| Version 1.0 Rev 0.9 | Incorporated all Changes from Core Group Review | January 23, 2009 | Frank Yeh |
| Version 1.0 Revision 1.0 | Incorporated final changes from public comment period | February 13, 2009 | Frank Yeh |
| Version 1.0 Revision 1.1 | Incorporated negotiated changes from final review and ratification sessions | February 17, 2009 | Frank Yeh |
| Version 1.0 Revision 1.2 | Final Change for Versioning, Added in .xsd document | February 19, 2009 | Frank Yeh |

## Disclaimer

# Table of Contents

# 1. Introduction

The Service Model is intended to assist the PSIA working groups in creating new protocols or converting contributed protocols to a standard service model that will be common to all endorsed specifications. Adherence to this service model will ensure interoperability between compliant protocols.

This model is similar in nature to Web services but is geared towards lightweight computing requirements on devices. As such, these protocols will not use Simple Object Access Protocol (SOAP) as defined by the W3C-defined Web services but instead will use a simplified XML schema and/or xml schema documents (.xsd's).

Unless otherwise noted, all PSIA specifications should treat all configuration and management aspects as resources utilizing the REpresentational State Transfer (REST) architecture.

The PSIA Service Model is based on a REST architecture. While REST specifies that all interfaces are defined as resources, in the PSIA Model these resources are grouped by service. This architecture provides a convenient way to group related resources within a hierarchical namespace and lends itself to service discovery and future expansion.

The PSIA reserves the right to add services at any time provided said services adhere to the PSIA model as defined herein. Every effort should be taken to maintain full backward compatibility when adding new services. The PSIA Service Model is designed to support expansion with backwards compatibility.

# 2. Design Considerations

Network-attached devices are often equipped with a web server to maintain various web pages. These pages allow the devices to be configured through an internet browser. It is natural to reuse this web server and the HTTP protocol in order for external applications to configure and control the device. Thus, all resources will use a standard HTTP request which will be processed by the device's web server.

When possible, IP devices should implement HTTPS to support privacy of data. It is assumed that the network infrastructure is configured properly with firewall, 802.1x, etc. and other features to provide basic network level security. Additionally, because IP devices are typically endpoint devices, HTTPS is assumed to provide sufficient safeguard in combination with the other features mentioned above.

Some devices are not capable of implementing HTTPS and in certain deployments it may not be necessary (i.e. closed networks). Additionally, SSL/TLS implies certificate management on an endpoint which could pose other problems. Embedded devices may not have a "trusted" certificate without a client explicitly trusting the certificate or uploading a trusted certificate. Furthermore, certificates may need to be regenerated upon configuration changes (IP address, etc.).

As such, the protocols use the HTTP Get and Post methods as described in "Hypertext Transfer Protocol -- HTTP/1.0" (RFC1945) and "Hypertext Transfer Protocol -- HTTP/1.1" (RFC2616).

## 2.1. REST Overview

REST is an approach to creating services that expose all information as resources in a uniform way. This approach is quite different from the traditional Remote Procedure Call (RPC) mechanism which identifies the functions that an application can call. Put simply, a REST Web application is noun-driven

while an RPC Web application is verb-driven. For example, if a Web application were to define an RPC API for user management, it might be written as follows:

```
GET http://webserver/getUserList
GET http://webserver/getUser?userid=100
POST http://webserver/addUser
POST http://webserver/updateUser
GET http://webserver/deleteUser?userid=100
```

On the other hand, a REST API for the same operations would appear as follows:

```
GET http://webserver/users
GET http://webserver/users/user100
POST http://webserver/users
PUT http://webserver/users/user100
DELETE http://webserver/users/user100
```

Part of the simplicity of REST is its uniform interface for operations. Since everything is represented as a resource, create, retrieve, update, and delete (CRUD) operations use the same URI.

## 2.2. Conformance

Every PSIA protocol will define one or more PSIA compliant services. To ensure interoperability, the following conformance requirements are also implied in each PSIA protocol.

### 2.2.1. Minimum API Set

In addition to the service specific mandatory requirements, a system/device must support all of the mandatory PSIA services.

Each specification may define one or more PSIA compliant services. Each service and each contained resource must be assigned a scope of either mandatory or optional. Within each implemented service type, all mandatory resources must be implemented.

### 2.2.2. XML Requirements

A system/device must support the syntax as defined by the W3C XML 1.0 specification.

A system/device must support the UTF-8 character set as described by

http://www.w3.org/International/O-charset

Additionally, XML content must correspond to the following Schemas as defined in Appendix 10:

- “ResourceList XML Schema”
- “ResourceDescription XML Schema”
- “QueryStringParameterList XML Schema”
- “ResponseStatus XML Schema”

Vendors may optionally extend this standard to include proprietary XML content as long as it does not conflict with the minimum set of APIs. In this case, it is recommended to use a vendor-specific XML namespace to avoid conflicting names that may arise with future revisions.

For example, if vendor XYZ123 Inc intends to extend the XML standard to include a <configOption> parameter, it is recommended to use <configOption xmlns="urn:XYZ123-com:configuration:options"> to avoid future namespace conflicts.

### 2.2.3. Protocol Requirements

A system/device must support transport of XML via either the HTTP/1.0 or HTTP/1.1 protocol as specified in RFC1945 and RFC2616, respectively. It is highly recommended that HTTP/1.1 is used in order to support key features (persistent connections, HTTPS, etc.). When HTTP 1.0 is implemented, the client applications must not issue multiple messages to the target systems/devices.

## 2.3. HTTP Methods and REST

The CRUD operations are defined by the HTTP method as shown in the table below.

| HTTP Method | Operation |
|---|---|
| POST | Create the resource |
| GET | Retrieve the resource |
| PUT | Update the resource |
| DELETE | Delete the resource |

Rules of thumb

GET calls should never change the system state. They are meant to only return data to the requestor and not to have any side effects
POST calls should only be used to ADD something that did not already exist.
PUT calls are expected to update an existing resource but if the resource specified does not already exist, it can be created as well. This will be the assumed default behavior of PUT calls. If any resource wishes to deviate from this behavior, it should be considered an exception and this should be noted in the implementation notes of the resource.

## 2.4. HTTP Status Codes and REST

The following table shows how the HTTP status codes map to REST operations along with the general use case for response headers and bodies. For more information, please see the table under each REST API.

| HTTP Status Codes | REST Meaning | POST | GET | PUT | DEL |
|---|---|---|---|---|---|
| 200 | "OK" - The request has succeeded.<br>Header Notes: None<br>Body Notes: The requested resource will be returned in the body. | | X | X | |

| HTTP Status Codes | REST Meaning | POST | GET | PUT | DEL |
|---|---|---|---|---|---|
| 201 | "Created" - The request has created a new resource.<br>Header Notes: The *Location* header contains the URI of the newly created resource.<br>Body Notes: The response returns an entity describing the newly created resource. | X | | | |
| 204 | "No Content" - The request succeeded, but there is no data to return.<br>Header Notes: None<br>Body Notes: No body is allowed. | | | X | X |
| 301 | "Moved Permanently" - The requested resource has moved permanently.<br>Header Notes: The *Location* header contains the URI of the new location.<br>Body Notes: The body may contain the new resource location. | | X | | |
| 302 | "Found" - The requested resource should be accessed through this location, but the resource actually lives at another location. This is typically used to set up an alias.<br>Header Notes: The *Location* header contains the URI of the resource.<br>Body Notes: The body may contain the new resource location. | | X | | |
| 400 | "Bad Request" - The request was badly formed. This is commonly used for creating or updating a resource, but the data was incomplete or incorrect.<br>Header Notes: The Reason-Phrase sent with the HTTP status header may contain information on the error.<br>Body Notes: The response may contain more information of the underlying error that occurred in addition to the Reason-Phrase. | X | | X | |

| HTTP Status Codes | REST Meaning | POST | GET | PUT | DEL |
|---|---|---|---|---|---|
| 401 | "Unauthorized" - The request requires user authentication to access this resource. If the request contains invalid authentication data, this code is sent.<br>Header Notes: At least one authentication mechanism must be specified in the *WWW-Authenticate* header. The Reason-Phrase sent with the HTTP status header may contain information on the error.<br>Body Notes: The response may contain more information of the underlying error that occurred in addition to the Reason-Phrase. | X | X | X | X |
| 403 | "Forbidden" - The request is not allowed because the server is refusing to fill the request. A common reason for this is that the device does not support the requested functionality.<br>Header Notes: The Reason-Phrase sent with the HTTP status header may contain information on the error.<br>Body Notes: The response may contain more information of the underlying error that occurred in addition to the Reason-Phrase. | X | X | X | X |
| 404 | "Not Found" - The requested resource does not exist.<br>Header Notes: None<br>Body Notes: None | X | X | X | X |
| 405 | "Method Not Allowed" – The request used an HTTP method that is not supported for the resource because the {API Protocol} specification does not allow this method. If the device does not support the functionality but it is a valid {API Protocol} operation, then a 403 is returned.<br>Header Notes: The *Allow* header lists the supported HTTP methods for this resource.<br>Body Notes: None | X | X | X | X |
| 500 | "Internal Server Error" - An internal server error has occurred.<br>Header Notes: None<br>Body Notes: None | X | X | X | X |

| HTTP Status Codes | REST Meaning | POST | GET | PUT | DEL |
|---|---|---|---|---|---|
| 503 | "Service Unavailable" – The HTTP server is up, but the REST service is not available. Typically this is caused by too many client requests.<br>Header Notes: The *Retry-After* header suggests to the client when to try resubmitting the request.<br>Body Notes: None | X | X | X | X |

## 2.5. Unique Identifiers

IDs are defined as URL-Valid Strings, as required by REST. The device will create an ID for all resources that add a resource.

IDs within each type should be unique at least on the channel level, but there is no requirement for uniqueness across devices. If globally unique IDs are desired, a globally unique ID should be derived using the method described in RFC4122.

## 2.6. ID Encoding

Because IDs will occur as part of a URI, there are two ways to encode an ID: either following RFC 3986 or, for pure binary IDs, as a hex string

RFC 3986 first converts the URI to UTF and then prints the following unreserved characters in the URI without any encoding:

- A-Z
- a-z
- 0-9
- -
- .
- _
- ~

All non-printable or reserved characters will be encoded as a two digit hex value prefixed by a %. For example, a space (ASCII value of 32) will be encoded as %20.

Because a pure binary ID can contain values that might interfere with the operation of browsers and web servers, PSIA protocols support hex encoding of the ID. The ID must begin with 0x (0X is also acceptable) followed by pairs of hex values. Each hex pair represents a single byte in the ID. For example: 0x3F431245DE67FAC46F9D034CA23AEFD4. The hexadecimal characters A-F can also be represented by a-f. So 0x3f431245de67fac46f9d034ca23aefd4 is equivalent to the previous ID.
If readable IDs are desired, it is recommended that IDs are created with unreserved, printable ASCII characters.

# 3. Architecture and Namespace

In a typical REST-based namespace, every node or object in the tree-structured hierarchy is considered a resource.

The PSIA model adds a resource sub-class called “Service”. Services are simply nodes which can contain other nodes. Nodes that do not contain other nodes (other than the standard node resources of the PSIA model) will continue to be called resources, while the term node will be used to refer to both services and resources.

Viewed as a tree, services are analogous to branches and resources are analogous to leaves.

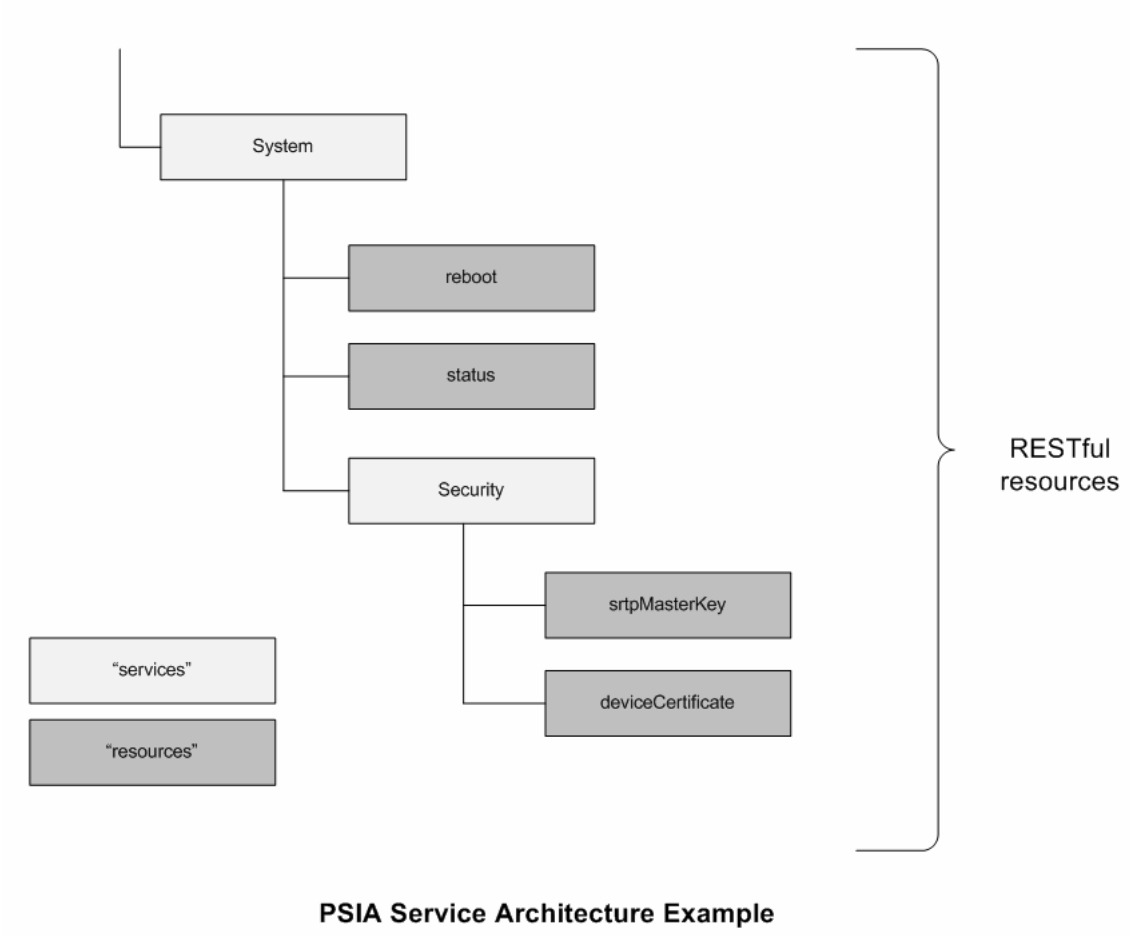
Each node must contain the following standard PSIA resources:

| | |
|---|---|
| description | which will respond to an HTTP GET with a ResourceDescription datablock |
| index | which will respond to an HTP GET with a ResourceList datablock |

Each node may contain the following standard PSIA resources:

| | |
|---|---|
| indexr | which will respond to an HTTP GET with a ResourceList datablock |
| capabilities | which will respond to an HTTP GET with a resource-specific XML Document |

The index resource will return a list of all the immediate “children” of a node. For services, this list could contain other services as well as resources. For resources, this list should only indicate which standard PSIA resources (IE description, index, and optionally indexr and capabilities) are contained. The optional indexr resource will return a recursive listing that descends through the namespace hierarchy.




**PSIA Service Architecture Example**

| Resource Name | Description | Mandatory/Optional |
|---|---|---|
| description | will respond to an HTTP GET with a <ResourceDescription> datablock | Mandatory |
| capabilities | will respond to an HTTP GET with a resource-specific XML Document | Optional |
| index | will respond to an HTTP GET with a <ResourceList> datablock | Mandatory |
| indexr | will respond to an HTTP GET with <ResourceList> datablock | Optional |

For all PSIA protocols, the root namespace of "PSIA" will be implicit, meaning it does not have to be included in the URL. Therefore, the root of any PSIA service's namespace will be a tag particular to that service.

Each service will be mandatory or optional, indicating to implementers which services they must implement at a minimum. Within each service, resources will also be mandatory or optional. This scope will be hierarchical so that any resource of an optional service is, by definition, optional but if an optional service type is deployed, then every mandatory resource within that service then becomes mandatory,

| Service URL | Description | Mandatory/Optional |
|---|---|---|
| /System | Resources related to general system configuration and operation | Mandatory |

| | | |
|---|---|---|
| /System/Storage | Resources related to local storage | Optional |
| /System/Network | Resource related to Network settings | Mandatory |
| /Security | Resources related to security of the device | Mandatory |
| /Security/AAA | Resources related to AAA functions | Mandatory |
| /Streaming | Resources related to streaming media content | Optional |
| /PTZ | Resources related to Pan/Tilt/Zoom | Optional |
| /Archive | Resources related to storage of content | Optional |
| /Diagnostics | Resources related to Diagnostics | Optional |
| /Custom | Resources that are specific to a protocol or vendor specific | Optional |

## 3.1 Multiple Channels and Versions

To provide for multi-channel support, a service must insert the implied “Channels” service as a child-node which should then contain an ID resource for each channel. Each ID resource will then respond to each of the resources applicable to the service.

For Single-Channel Devices, the Channels service must still be included to maintain consistency between single and multi-channel devices and to provide for the case where a multichannel device has only created a single channel.

Note that Channel IDs are arbitrarily assigned by the device.

(EG. For a single channel device:
/Streaming/Channels/0/keyFrame
For a multi-channel device
/Streaming/Channels/0/keyFrame
/Streaming/Channels/1/keyFrame)

Devices may either pre-define this multichannel structure or support dynamic additions and deletions of channels (using HTTP POST and DELETE) as applicable.

In order to differentiate services that essentially provide for multiple instances of something within the hierarchy, it is recommended that services at the root level be referred to as “Root Services” while the term service continue to be used to describe any node that contains other nodes (EG Streaming is a Root Service, Channels is not).

Each node, be it a resource or service, will be able to return a description of itself within the service model. This description will include a version attribute to support versioning within the PSIA Service model. While this practice will allow resources with different versions to exist within the same services, it is mandatory that all resources within a service container are fully backward compatible.

If a new service version is introduced that does not maintain backwards compatibility with previous versions, then a new service must be created for the new, incompatible version (EG /Streaming and /StreamingV2). IE it is acceptable to add resources to a Service but not to replace them with new versions that are not backward compatible. If new resource versions must be added, the Root Service name should be changed to indicate a new Service version.

# 4. System Flow

Before any protocol can be used to manage a device, it must first be discovered. It is required that the Zeroconf (Zero Configuration Networking) technology be supported to discover/locate the device. Once this step is accomplished, transactions can commence.

While devices must support ZeroConf, this does not preclude devices from using DHCP or manual IP Addressing. Devices should check for manually assigned IP addresses and DHCP assigned IP Addresses before attempting to assign an address using IPv4 Link-Local Addressing (which is the method for Ip Address discovery for ZeroConf).

ZeroConf is normally expected to operate in a local area network. Where discovery must be supported in a routed or Wide Area Network, part 2 of ZeroConf (Multicat DNS) becomes superfluous and part 3 (DNS-SD) must be supported by configuration of actual DNS servers.

HTTP requests are made through the device's web server. The HTTP response may contain XML content (for GET actions), XML response information (for PUT or POST actions), or various text/binary content (for retrieval of configuration data, etc.). Edge devices should be able to handle overlapping/simultaneous HTTP requests, as well as persistent connections to handle multiple HTTP transactions.

The XML content should be described by .xsd documents. Relevant XML data structures must be documented in an Appendix section of each PSIA Specification.

## 4.1. Service Discovery

Zeroconf (Zero Configuration Networking) technology specifies DNS-SD (DNS Service Discovery as described in http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt) to discover/locate a device.

All PSIA protocols will require DNS-SD for device discovery. To support this discovery model, the PSIA is registering a DNS SRV (RFC 2782) service type to be used to discover all PSIA protocols via DNS –SD (DNS Service Discovery).

DNS-SD discoveries for the PSIA's public DNS service type should be used to discover the device according to DNS Service Discovery (http://www.dns-sd.org/ServiceTypes.html). Once a device is established as a PSIA-compliant device, its services and resources can be discovered using standard HTTP GETs using the standard, mandatory resources.

The following information should be advertised:
A path of "/index/" – can be obtained from the "path" key in the TXT record
The {host} – can be obtained from the service's SRV record
The {port} – can be obtained from the service's SRV record
The version of the DNS SVR record in "txtvers"
The PSIA protocol version in "protovers"

Once a PSIA-compliant device has been discovered, an HTTP GET of its mandatory index resource will return a list of the services that it supports. At this point, the standard methods can be used to "walk" the namespace tree and discover the supported services and resources.

It should be noted that the index resource returns only the first level resources of a node, but the indexr resource will return a recursive tree structured list with the current resource as root.

## 4.2. Persistent Connections

Devices that implement HTTP/1.1 should support persistent connections in order to support video management systems or client applications that issue multiple HTTP(S) transactions. The PSIA assumes that HTTP/1.1 is implemented and utilized according to RFC2616. For persistent connections with devices that support HTTP/1.0 RFC2068 section 19.7.1 should be referenced.

A video management system or client application should, when using a persistent connection for multiple transactions, implement the “Connection: Keep-Alive” HTTP header. The management system should also use the “Connection: close” HTTP header field for the last transaction made within this persistent connection. This process assumes that the application is aware of the last request in a sequence of multiple requests.

## 4.3. Authentication

When an application sends any request to the device, it must be authenticated by means of Basic Access or Digest authentication according to RFC 2617. This means the user access credentials are sent along with each request. If a user is authenticated, the request will follow the normal execution flow. Basic Access and/or Digest authentication are mandatory for all device implementations. It is up to the client to determine which method to use for different deployment scenarios.

A default user account, “admin”, must be provided by the IP device. This account should have a default privilege level of “administrator”, and must not be deleted. The default password of the “admin” account should be null in factory default configuration.

Example client HTTP request header and body with no authentication credentials:

```
GET /index
…
```

Example unauthorized HTTP response header and body:

```
HTTP/1.1 401 Unauthorized
…
WWW-Authenticate: Digest realm="testrealm@host.com",
                         qop="auth,auth-int",
                         nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                         opaque="5ccc069c403ebaf9f0171e9517f40e41"
Content-Type: application/xml; charset=”UTF-8”
Content-Length: xxx    (note: xxx = size of XML block)

<?xml version="1.0" encoding="UTF-8" ?>
<ResourceList version=“1.0” xmlns=“urn:psialliance-org:resourcelist”>
     …
</ResourceList>
```

Example client HTTP request header and body with authentication credentials (username “Mufasa” and password “Circle of Life”):

```
GET /index
…
Authorization: Digest username="Mufasa",
                      realm="testrealm@host.com",
                      nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                      uri="/dir/index.html",
                      qop=auth,
                      nc=00000001,
                      cnonce="0a4f113b",
                      response="6629fae49393a05397450978507c4ef1",
                      opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Example authorized HTTP response header and body:

```
HTTP/1.1 200 OK
…
Content-Type: application/xml; charset=”UTF-8”
Content-Length: xxx    (note: xxx = size of XML block)

<?xml version="1.0" encoding="UTF-8" ?>
<ResourceList version="1.0" xmlns="urn:psialliance-org:resourcelist">
     <Resource>
          …
     </Resource>
</ResourceList>
```

## 4.4. Access Restrictions

All supported resources on a device must be fully accessible to users with the “Administrator” privilege level. This means that in order to use the full suite of resources a device offers, authentication must be granted with a user account having a privilege level corresponding to “Administrator”.

It is required that at least one account with Administrator privileges be active at all times.

There are no restrictions as to which resources are accessible to users with other privilege levels. A vendor may choose to limit, for example, the allowable resources for user accounts with lower privileges. However, since user-specific authorization is not a function of the protocol, it is often assumed that full administrative rights will be available via the protocol. User-specific authorization functions are expected to be handled by the calling application.

While “Administrator” privilege levels must be provided for, there is no requirement that any specific users be assigned an administrative privilege level. In cases where external requirements preclude a single user having all privileges, more granular authorization can be performed using user and role assignments which do not have full administrative privileges.

## 4.5. Setting Configurations

Resources to set device configurations will use the HTTP PUT method if there is an XML block parameter for the request, and the HTTP GET method if there is no XML block parameter. The inbound XML format is defined according to a resource-specific XML schema For PUT operations, the request status will be indicated by the XML response information returned from the device, and can be used to indicate the status of the set operation. This XML format is defined according to “XML Response Schema” (see section 4.12 for details). After successfully updating the repository, the device returns an XML response with status code “OK”. A separate status code is used for unsuccessful operations. In either case, the device will not return a response until it is ready to continue normal operation – this includes accepting streaming requests, receiving behavioral control commands, etc.

Example HTTP request header and body:

```
POST /System/deviceInfo HTTP/1.1
…
Content-Type: application/xml; charset=”UTF-8”
Content-Length:xxx    (note: xxx = size of XML block)

<?xml version="1.0" encoding="UTF-8" ?>
<DeviceInfo version="1.0" xmlns="urn:psialliance-org:system:deviceinfo">
…
</DeviceInfo>
```

Example HTTP response header and body:

```
HTTP/1.1 200 OK
…
Content-Type: application/xml; charset=”UTF-8”
Content-Length: xxx    (note: xxx = size of XML block)

<?xml version="1.0" encoding="UTF-8" ?>
<ResponseStatus version=“1.0” xmlns=“urn:psialliance-org:response”>
…
</ResponseStatus>
```

## 4.6. Getting Configurations

Resources to get device configurations or status information will use the HTTP GET method. If successful, the result will be returned in XML format according to the resource description. If the request is unsuccessful for any reason (i.e. not authenticated), the result will be returned in XML format according to “ResponseStatus XML Schema”. The Content-Type and Content-Length will be set in the headers of the HTTP response containing the XML data. The Content-Type is: application/xml; charset=“UTF-8”.

Example HTTP request header and body:

```
GET /System/deviceInfo HTTP/1.1
…
```

Example HTTP response header and body:

```
HTTP/1.1 200 OK
…
Content-Type: application/xml; charset=”UTF-8”
Content-Length:  xxx    (note: xxx = size of XML block)

<?xml version="1.0" encoding="UTF-8" ?>
<DeviceInfo version="1.0" xmlns="urn:psialliance-org:system:deviceinfo">
    …
</DeviceInfo>
```

## 4.7. Getting Capabilities

Capabilities can also be retrieved by any resources node that specifies an XML payload for inbound data with an HTTP GET of its “capabilities” resource. In other words, a client application can query a device for its capabilities in order to understand what XML tags are supported, the acceptable data ranges, etc. See Section 5.4 for more detail on the returned capabilities.

Example HTTP request header and body:

```
GET /PTZ/channels/ID/0/absolute/capabilities HTTP/1.1
…
```

Example HTTP response header and body:

```
HTTP/1.1 200 OK
…
Content-Type: application/xml; charset=”UTF-8”Content-Length: xxx (note: xxx = size of XML
block)
<?xml version="1.0" encoding="UTF-8" ?>
<PTZData version=“1.0” xmlns=“urn:psialliance-org”>
    <pan min=”-100” max=”100”/>
    <tilt min=”-100” max=”100”/>
    <zoom min=”-100” max=”100”/>
    <Momentary>
        <duration min=”0”/>
    </Momentary>
    <Relative>
        <positionX min=”0” max=”1024”/>
        <positionY min=”0” max=”1024”/>
        <relativeZoom min=”-100” max=”100”/>
    </Relative>
    <Absolute>
        <elevation min=”-90” max=”90”/>
        <azimuth min=”0” max=”360”/>
        <absoluteZoom min=”0” max=”100”/>
    </Absolute>
    <Digital>
        <positionX min=”0” max=”1024”/>
        <positionY min=”0” max=”1024”/>
        <digitalZoomLevel min=”0” max=”100”/>
    </Digital>
</PTZData>
```

## 4.8. Uploading Data

Resources to upload data (i.e. firmware, configuration file, etc.) to the device will use the HTTP PUT method. The content of the data will be stored in the body of the HTTP request. The Content-Type and Content-Length will be set in the headers of the HTTP request. The Content-Type is “application/octet-stream”. In addition, each resource may optionally specify a different inputXML structure.

Example HTTP upload request header and body:

```
POST /System/configurationData HTTP/1.1
…
Content-Type: application/ xml; charset=”UTF-8”

[proprietary configuration data content]
```

Example HTTP upload response header and body:

```
HTTP/1.1 200 OK
…
Content-Type: application/xml; charset=”UTF-8”
Content-Length: xxx    (note: xxx = size of XML block)

<?xml version="1.0" encoding="UTF-8" ?>
<ResponseStatus version=“1.0” xmlns=“urn:psialliance-org:response”>
     …
</ResponseStatus>
```

## 4.9. Receiving Data

Resources to receive data (i.e. configuration file, etc.) from the device will use the HTTP GET method. The content of the data will be stored in the body of the HTTP response. The Content-Type and Content-Length will be set in the headers of the HTTP response, according to the type of data being returned.
The client may use the Accept: header string to tell the server what formats it accepts. Depending on what the client accepts, the server may transcode, transform or even compress the data to match the client’s expectations.

Example HTTP download request header and body:

```
GET /System/configurationData HTTP/1.1
…
```

Example HTTP download response header and body:

```
HTTP/1.1 200 OK
…
Content-Type: application/octet-stream
Content-Length: xxx    (note: xxx = size of XML block)

[proprietary configuration data content]
```

## 4.10. Operations

For stateless operations (i.e. function calls) the formula is:

PUT /Service/<Operation>

Resources must indicate in their descriptions which XML payload is required or the query string parameters to be used in the operation.

## 4.11. Diagnostics

Diagnostics (and other stateful operations) run in the background on the device, so it must be possible to create them asynchronously and be able to query their status.
The REST model works well here:

| Request | Result |
|---|---|
| POST /Diagnostics/<command> | Returns diagnostic ID |
| GET /Diagnostics/<command>/<ID> | Get information on this ID |
| DELETE /Diagnostics/<command>/<ID> | Delete command in progress |
| GET /Diagnostics/commands | Get information on |

| | |
|---|---|
| | all commands running |

## 4.12. Response Status

Responses to many resource calls contain data in the form of the ResponseStatus XML document. Within each specification, separate services and resources may each have their own data structures. The only provision of the model is that each ResourceDescription must indicate which structures are used and each structure must be defined in an XML schema document within the specification document. If resources do not define their own response structures, they may use the PSIA standard ResponseStatus structure as defined in Appendix 10.

### 4.12.1. Status Code

A ResponseStatus with statusCode=OK will be sent after the command has been completely processed on the device. Even if the request contains some parameters that are not supported, the device will ignore those parameters and return statusCode=OK.

A device will send a Device Busy response to a command which cannot be processed at that time (eg. receiving a reboot command while the flash is being updated)

If the device fails to perform the request - possibly due to a hardware error - it will return a Device Error statusCode and a fault message in the statusString.

An Invalid Operation statusCode is returned in response to a command that has not been implemented. Invalid Operation is also returned if an authentication attempt fails or the logged in user has insufficient privileges to execute the command.

An Invalid XML Format statusCode is returned if the XML is badly formed and causes the parser to fail. The statusString should indicate the fault.

An incomplete message or a message containing an out-of-range parameter will return an Invalid XML Content statusCode and associated statusString.

For settings that require a reboot to take effect, such as changing the network address or a firmware update, the Reboot Required statusCode is returned.

### 4.12.2. Status String

It is recommended that for all responses where the returned statusCode is not OK, a descriptive statusString be returned indicating the reason the command was not completed.

### 4.12.1. ID

In POST operations where the device will return an ID of the resource created, this attribute will be used to pass back the created ID.

## 4.13. Processing Rules

Any field (particularly in the inbound XML parameters) that is not supported by the device should be ignored. For any given resource there may be some special processing rules. These rules are documented in the column associated with the heading “Implementation Note”.

# 5. XML Modeling

## 5.1. File Format

All XML files must use UTF-8 (8-bit UCS/Unicode Transoformation Format) encoding according to RFC3629. A BOM (byte-oder mark) can optionally be used. Thus, a media device should support UTF-8 encoding with or without a BOM.

## 5.2. Data Structures

Any Resource can specify separate input and output XML Documents. If a specific data structure is defined, these must be specified as XML Schema Documents (xsd) within the specification. The xsd's created for PSIA specifications are to be included in the appendix section of the relevant specification. In addition, the PSIA will be posting xsd documents of relevant schemas at http://www.psialliance.org/XMLSchemas to support online reference of the schemas. However, there is no guarantee that the schemas will be posted at the same time the documents are published. For this reason, the schema definitions within the specification documents themselves are the minimal requirement.

## 5.3. Lists

Many of the XML blocks contain lists. The syntax of these lists is <XXXList>, where XXX is a name referring to the XML setting. Inside of the <XXXList> tag is one or more <XXX> nodes. As an example, the <ChannelList> block may contain content as such:

```
<ChannelList>
    <Channel>
            <id>1</id>
            …
    </Channel>
    <Channel>
            <id>2</id>
            …
    </Channel>
    …
</ChannelList>
```

## 5.4. Capabilities

Capabilities for any resource that defines an XML block for input will be returned as an XML document that is essentially an XML instance of the resource-specific input XML block. This XML document must contain the acceptable values for each attribute.

While XML Schema Documents are also required of any XML data defined by any PSIA specification and xsd documents are capable of defining the acceptable range of values for any attribute, using a global xsd to define capacities would imply that all devices support the same options for any parameter. By allowing devices to respond to the capabilities request, each device can support different values for any attribute, within the constraints of the schema.

| Capability Attribute | Description | Syntax | Applicable XML Data Types |
|---|---|---|---|
| Min | The minimum character length for a string, or the minimum numerical value of a number | Examples:<br>min="0"<br>min="64"<br>min="-100" (numerical only)<br>min="1.2" | All except fixed data types [1] |
| max | The maximum character length for a string, or the maximum numerical value of a number | Examples:<br>max="5"<br>max="64"<br>max="4096"<br>max="10.50" | All except fixed data types [1] |
| range | Indicates the possible range of numerical values within the "min" and "max" attributes of an element. This attribute should only be used if the possible values for an XML element does not include the entire numerical range between "min" and "max" attributes | Ranges are listed in numerical order separated by a "," character. A range has the form "x~y" where x is the range floor and y is the range ceiling. Single numbers may also be used.<br><br>*Example*: if an XML element supports values 0, 123, 1024 to 2000, and 2003, the syntax would be:<br>range="0,123,1024~2000,2003" | All numerical data types |
| opt | Lists the supported options for a CodeID data type. Required for XML elements with a CodeID data type. This attribute should *not* be used for any other data type | If all options are supported, the syntax is "all". Otherwise, supported options are listed separated by a "," character.<br><br>Examples:<br>opt="all"<br>opt="1,2,3"<br>opt="1,2,5,8,9,10,11" | CodeID |
| Def | Indicates the default value of the XML element. If the element has no default value, this attribute should *not* be used | Examples:<br>def="1234"<br>def="Device ABC"<br>def="3" | All data types |
| reqReboot | Indicates if configuration of this XML element requires a device reboot before taking effect. If an element doesn't require a reboot, this attribute should *not* be used | reqReboot="true" | All data types |
| dynamic | Indicates if an XML element has dynamic capabilities dependent on other XML configurations. For example, if an element's data | dynamic="true" | All data types |

| | | | |
|---|---|---|---|
| | range changes based on another element's configured value, this attribute must be used. In this case, the element's capability attributes must always reflect the current device configuration | | |
| Size | Indicates the maximum number of entries in an XML list. This attribute is only applicable to XML list elements. This attribute should not be used for any other type of element (see section 5.3 for details) | *Example*: If a device supports 5 users the example would be<br><br><UserSetting><br><UserList size="5"><br>… | Only supported for list elements (see section 5.3) |

[1] Fixed, pre-defined data types do not need certain capability attributes because their formats/data ranges are already defined. Where pre-defined data types are used, each protocol document must include an enumeration of these formats in an Appendix section.

# 6. Custom Services & Resources

In order to support system/device specific resources that are not common to the public service definitions, the CUSTOM service type is provided. An HTTP GET of the index resource of the CUSTOM service returns a list of the custom services and resources supported by the system/device.

For each custom resource, an implicit mandatory resource named “Description” must be supported. An HTTP GET of any custom resource’s Description resource must return a ResourceDescriptionBlock similar to the Resource Description information described in section 7.6.

Custom services and resource can be used to support protocol-specific resources that are thought to be of an interim nature (IE a forthcoming protocol will most probably deprecate these resources) or vendor-specific proprietary resources. As long as all custom services and resources are implemented according to the PSIA Service Model, they can be discovered and called by PSIA-compliant clients and applications.

# 7. Interface Design

The HTTP URL format is of the general form

```
<protocol>://<hostname>:<port>/<URI>? P1=v1&P2=v2....&pn=vn
```

All requests will follow this format. A brief description of these components follows:

## 7.1. Protocol

The protocol field refers to the URL scheme that will be used for the particular request. Note that the current specification allows the following schemes:

- http
- https

## 7.2. Hostname

The hostname field refers to the hostname, IP address, or the FQDN (fully qualified domain name) of an IP device.

## 7.3. Port

The port field indicates the port number to be used for the HTTP request. The default port number for HTTP is 80. For HTTPS, the default port is 443. For RTSP, the default port is 554. If neglected in the URL, these default port numbers will be used for the request (as defined in RFC2616, RFC2818, and RFC2326 respectively).

The HTTP and HTTPS port number is configurable for IP devices. The standard HTTP and HTTPS ports (80 and 443) will be assumed unless otherwise specified.

## 7.4. URI

The URI absolute path is most often of the form “<SERVICE>/<resource>” where <resource> corresponds to one of the resources defined in the specification. For example, <SERVICE> could refer to “System” or “Security”. This is true for resources that update or retrieve device configurations.

## 7.5. Query String

Resources specify required and optional query string parameters. In either case, these query string parameters must be listed in name-value pair syntax (p1=v1&p2=v2…&pn=vn) following the URI.

Example GET HTTP request with query string parameters:

```
GET /Streaming/Channels/1/picture?snapShotImageType=jpeg
…
```

Example POST HTTP request with query string parameters:

```
PUT /System/time?localTime=2009-02-16%2013:30:00
…
```

Each resource may define a set of parameters, in the form of name-value pairs, which exist in the query string. If resources define input dta specific to the resource, this takes precedence over the use of query string parameters for input.

## 7.6. Resource Description

For each resource in this document, the following components are defined:

**Format** – indicates the URL format of the HTTP request

**Type** – indicates whether this is a service or resource

Method specific (GET, PUT, POST, DELETE)

**Query string parameters** – indicates the name/value pairs (P1,P2,P3,…Pn) for the resource.

**Inbound Data**– indicates inbound data for the resource as follows:

- **NONE** – indicates no input data
- **DataBlock** - the name of a Data Block defined within the specification. Datablocks used here must be defined within the specification document. In addition, it is strongly recommended that .xml schema documents be created for each referenced datablock.
- **Mime type** – indicates that the input data is in the HTTP payload with the indicated mime type. NOTE a type of 'application/xml" is not considisered valid.

If a device doesn't support particular XML tags or blocks, they need not be used in the resource operations.

Generally, if fields are not provided in the inbound XML, the current values for these fields should remain unchanged in the device's repository.

If required fields do not already exist in the device's repository, they must be provided in applicable resource operations.

**Function** – describes the general function behavior

**Return Result** – describes the response from the HTTP request

**Implementation Note** – describes the implementation behavior and any special processing rules for the resource.

For example,

| URI | index | Version | 1.0 | Type | Resource |
|---|---|---|---|---|---|
| Function | Enumerate child nodes | | | | |
| Methods | Query String(s) | Inbound Data | | Return Result | |
| GET | None | None | | <ResourceList> | |
| Notes | Returns a flat (non-recursive) listing of all child nodes | | | | |

In order to support discovery of CUSTOM service resources, this resource description data structure is also captured as a data block of type ResourceDescription. Whenever an HTTP GET of a device's CUSTOM/Index resource is executed, a list of the device's custom resources is returned. For each custom resource, an HTTP GET of the mandatory resource "Description" will return a ResourceDescription Block indicating what the resource does and how it should be used.

# 8. PSIA Standard Resource Descriptions

This section describes the standard PSIA resources

## 8.1. index

| URI | index | Version | 1.0 | Type | Resource |
|---|---|---|---|---|---|
| Function | Enumerate child nodes | | | | |
| Methods | Query String(s) | Inbound Data | | Return Result | |
| GET | None | None | | <ResourceList> | |
| Notes | Returns a flat (non-recursive) listing of all child nodes | | | | |

## 8.2. indexr

| URI | indexr | Version | 1.0 | Type | Resource |
|---|---|---|---|---|---|
| Function | Enumerate child nodes | | | | |
| Methods | Query String(s) | Inbound Data | | Return Result | |
| GET | None | None | | <ResourceList> | |
| Notes | Returns a recursive listing of all child nodes | | | | |

## 8.3. description

| URI | Description | Version | 1.0 | Type | Resource |
|---|---|---|---|---|---|
| Function | Describe Current Resource | | | | |
| Methods | Query String(s) | Inbound Data | | Return Result | |
| GET | None | None | | <ResourceDescription> | |
| Notes | Returns a description of the resource | | | | |

## 8.4. capabilities

| URI | capabilities | Version | 1.0 | Type | Resource |
|---|---|---|---|---|---|
| Function | Return capabilities of Current Resource | | | | |
| Methods | Query String(s) | Inbound Data | | Return Result | |
| GET | None | None | | Resource-Specific | |
| Notes | Returns a Capabilities description of the resource | | | | |

# 9. Acknowledgements

This document and the PSIA protocol model would not have been possible without significant contributions by various member companies. While the efforts of all our members are appreciated, the PSIA would like to explicitly acknowledge the contributions of Cisco, Object Video, GE Security, Genetec, MileStone, Texas Instruments, IQInvision, Pelco, IBM, and Honeywell for their contributions of Intellectual Property, Market Requirements and technical activity.

# 10.

# 10. Appendices

## 10.1. Schemas

The following data structures are defined for use with the PSIA Service Model The format used in this section are basic samples intended to quickly demonstrate the structure of the data blocks. Note that the actual PSIA protocols are to include their documented data structures as .xsd files.

### 10.1.1. ResourceDescription

```
<ResourceDescription version="1.0" xmlns="urn:psialliance-org:resourcedescription">
    <name>index</name>
    <version>1.0</version>
    <type>resource</type>
    <get>
            <queryStringParameterList>none</queryStringParameterList>
            <inboundXML>none</inboundXML>
            <function>enumerate 1st level children</function>
            <returnResult>ResourceList</returnResult>
            <notes>non-recursive</notes>
    </get>
    <put></put>
    <post></post>
    <delete></delete>
</resourceDescription>
```

### 10.1.2. ResourceList

```
  <?xml version="1.0" encoding="utf-8" ?>
- <ResourceList version="1.0" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="urn:psialliance-org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:psialliance-org http://www.psialliance.org/XMLSchemas/service.xsd">
-   <Resource version="1.0" xmlns="urn:psialliance-org" xlink:href="/index">
            <name>index</name>
            <type>resource</type>
  </Resource>
- <Resource xlink:href="/System">
    <name>System</name>
    <type>service</type>
-   <ResourceList>
-           <Resource xlink:href="/System/Network">
                  <name>Network</name>
                  <type>service</type>
-                 <ResourceList>
-                       <Resource xlink:href="/System/Network/ipAddress">
                              <name>ipAddress</name>
                              <type>resource</type>
                       </Resource>
                  </ResourceList>
            </Resource>
    </ResourceList>
  </Resource>
  </ResourceList>
```

### 10.1.3. QueryStringParameterList

```
  <?xml version="1.0" encoding="utf-8" ?>
- <QueryStringParameterList version="1.0" xmlns="urn:psialliance-org">
-   <QueryStringParameter>
            <name>positionX</name>
            <type>integer</type>
            <description>X position of scaling window</description>
    </QueryStringParameter>
  </QueryStringParameterList>
```

### 10.1.4. responseStatus

```
  <?xml version="1.0" encoding="utf-8" ?>
- <ResponseStatus version="1.0" xmlns="urn:psialliance-org">
  <requestURL>/Streaming/Channels</requestURL>
  <statusCode>1</statusCode>
    <!-- O=1-OK, 2-Device Busy, 3-Device Error, 4-Invalid Operation, 5-Invalid XML Format, 6-
Invalid XML Content; 7-Reboot Required-->
  <statusString>OK</statusString>
  <ID>1</ID>
  </ResponseStatus>
```

## 10.1.5. service.xsd

The following XML Schema Document contains XML schema definitions for all of the PSIA Service Model data structures. All PSIA specifications are to use this schema document to maintain consistency of the PSIA Service Model data structures.

This document and all subsequent PSIA XML Schema Documents will be posted at http://www.psialliance.org/XMLSchemas.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema version="1.0"
           targetNamespace="urn:psialliance-org"
           xmlns="urn:psialliance-org"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xlink="http://www.w3.org/1999/xlink"
                  elementFormDefault="qualified">
  <xs:import namespace="http://www.w3.org/1999/xlink" schemaLocation="xlink.xsd"/>

  <xs:annotation>
    <xs:documentation xml:lang="en">
      PSIA Core Service Schema
    </xs:documentation>
  </xs:annotation>

  <!-- ID -->
  <xs:simpleType name="Id">
    <xs:restriction base="xs:string">
      <!-- TODO -->
    </xs:restriction>
  </xs:simpleType>

  <!-- StatusCode -->
  <xs:simpleType name="StatusCode">
    <xs:restriction base="xs:int">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="7"/>
    </xs:restriction>
       <!-- O=1-OK, 2-Device Busy, 3-Device Error, 4-Invalid Operation, 5-Invalid XML
Format, 6-Invalid XML Content; 7-Reboot Required-->
  </xs:simpleType>

  <!-- ResourceType -->
  <xs:simpleType name="ResourceType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="service"/>
      <xs:enumeration value="resource"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- QueryStringParameter -->
  <xs:complexType name="QueryStringParameter">
    <xs:sequence>
       <xs:element name="name" type="xs:string" />
       <xs:element name="type" type="xs:string" />
       <xs:element name="description" type="xs:string" minOccurs="0" maxOccurs="1"/>
      <xs:any namespace="##any" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <!-- QueryStringParameterList -->
  <xs:complexType name="QueryStringParameterList">
    <xs:sequence>
      <xs:element name="QueryStringParameter" type="QueryStringParameter" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <!-- URLParameters -->
  <xs:complexType name="URLParameters">
    <xs:sequence>
      <xs:element name="queryStringParameterList" type="QueryStringParameterList" />
         <xs:element name="inboundData" type="xs:string" />
         <xs:element name="returnResult" type="xs:string" />
         <xs:element name="function" type="xs:string" />
         <xs:element name="notes" type="xs:string" />
      <xs:any namespace="##any" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

<!-- ResponseStatus -->
  <xs:complexType name="ResponseStatus">
    <xs:sequence>
      <xs:element name="requestURL" type="xs:anyURI" />
      <xs:element name="statusCode" type="StatusCode" />
      <xs:element name="statusString" type="xs:string" />
      <xs:element name="id" type="Id" minOccurs="0" maxOccurs="1" />
      <xs:any namespace="##any" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
       <xs:attribute name="version" type="xs:string" use="required"/>
  </xs:complexType>

  <!-- ResourceDescription -->
  <xs:complexType name="ResourceDescription">
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
         <xs:element name="version" type="xs:string" />
         <xs:element name="type" type="ResourceType" />
         <xs:element name="description" type="xs:string" minOccurs="0" maxOccurs="1"/>
         <xs:element name="notes" type="xs:string" minOccurs="0" maxOccurs="1"/>
         <xs:element name="get" type="URLParameters" />
         <xs:element name="put" type="URLParameters" />
         <xs:element name="post" type="URLParameters" />
         <xs:element name="delete" type="URLParameters" />
      <xs:any namespace="##any" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="required"/>
  </xs:complexType>

  <!-- Resource -->
  <xs:complexType name="Resource">
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
         <xs:element name="version" type="xs:string" />
         <xs:element name="type" type="ResourceType" />
         <xs:element name="description" type="xs:string" minOccurs="0" maxOccurs="1"/>
         <xs:element name="ResourceList" type="ResourceList" minOccurs="0" maxOccurs="1"/>
      <xs:any namespace="##any" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="required"/>
  </xs:complexType>

  <!-- ResourceList -->
  <xs:complexType name="ResourceList">
    <xs:sequence>
      <xs:element name="Resource" type="Resource" minOccurs="0" maxOccurs="unbounded"/>
```

```
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="required"/>
  </xs:complexType>

</xs:schema>
```