RDK-B Architecture

RDK- B Architecture

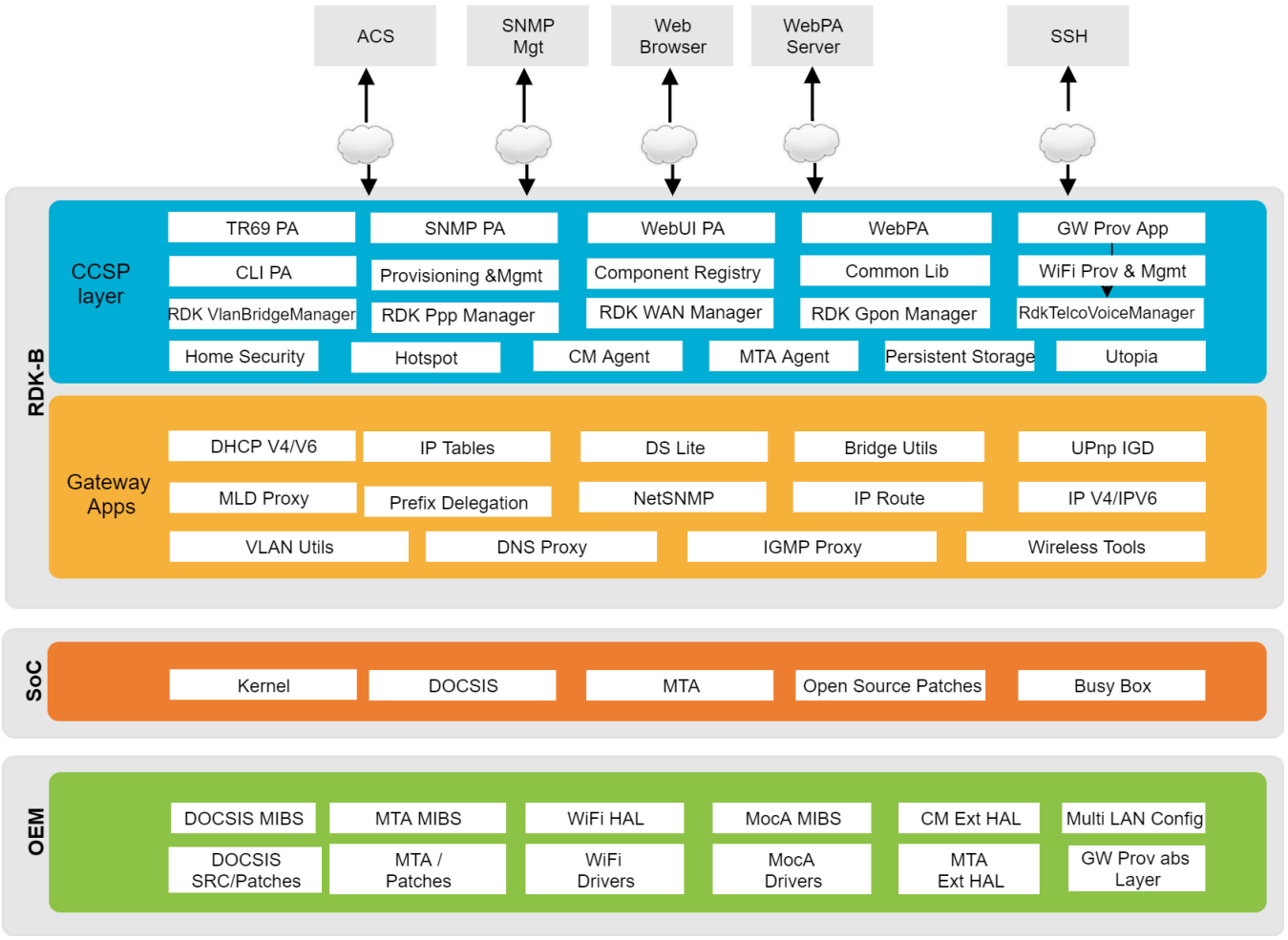
RDK-B is developed as a modular software stack built from a collection of individually reusable software components and is based on the following design considerations:

- Software modularity
- Abstraction of external management protocols
- Independence from wide area network type
- Silicon independence
- Linux kernel independence
- Software structure that allows multiple organizations and teams to work in parallel

On this page:

The architecture supports pluggable component modules which communicate over the CCSP message bus. RDK-B uses a collection of protocol agent components and supports multiple device management protocols(TR-069, SNMP etc). Protocol agents process the protocol specific details and provide abstraction to the common internal data model.

TR-181 data model is the common internal data model used by all RDK-B components to communicate over the message bus. RDK-B also supports multiple SoC vendors through component level hardware abstraction layers.



- RDKB is architected with “Software Components” . A software component is a software package that delivers a set of features or services.

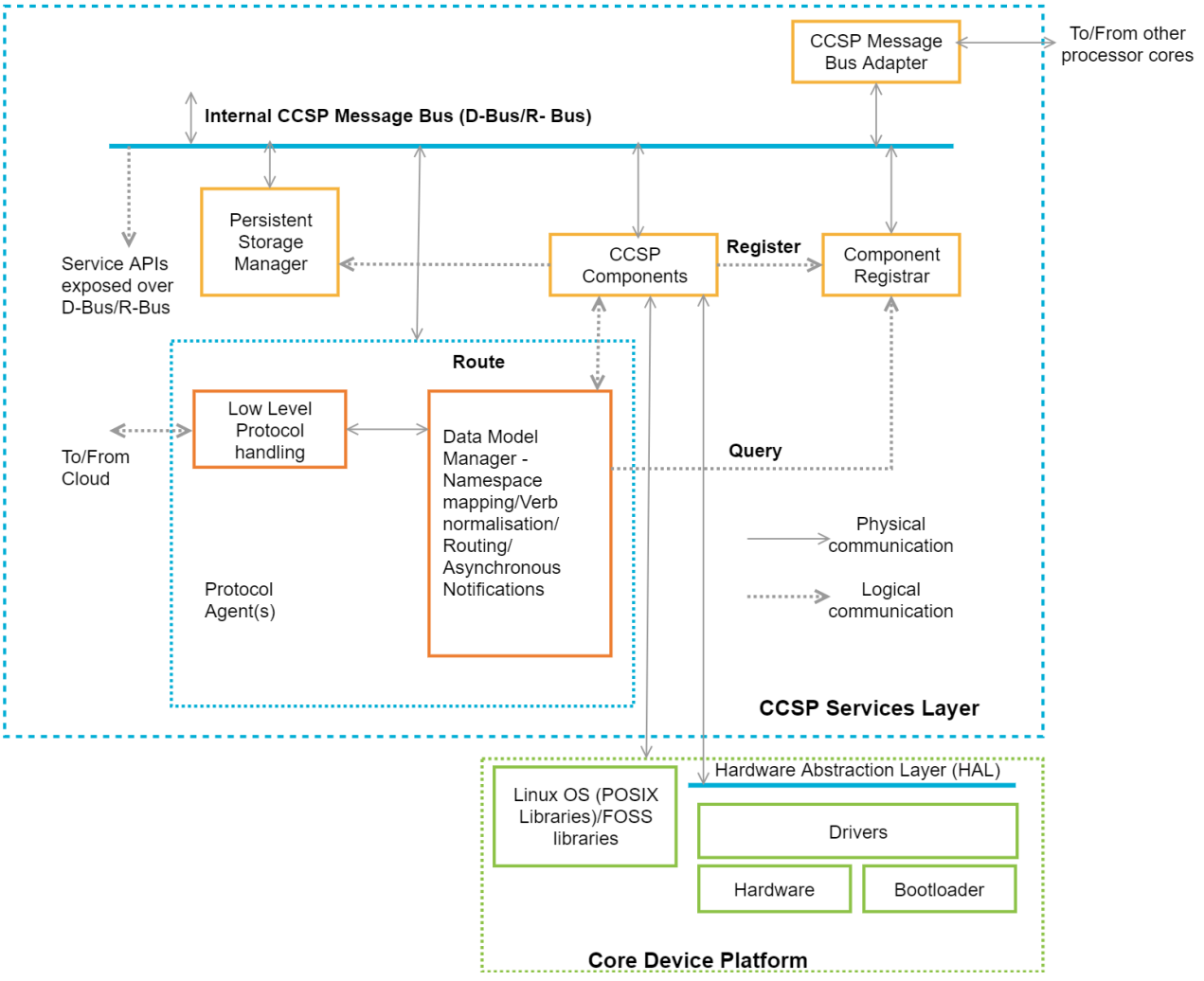
Examples: Cable Modem (CM) Agent, EPON Agent, DSL Agent, WiFi AP Manager, TR-069 Protocol Adapter, WebPA, DMCLI.

- RDK-B is Yocto based and it can run on any modern Linux kernel and can easily be ported/customised by developers.
- Also, RDK-B is not dependent on WAN type and supports DOCSIS and EPON.

CCSP High Level Architecture

A high level view of how CCSP components communicate with each other and fit together to form larger device profiles is explained here. An overview information on the CCSP Message Bus and CCSP Component Registry and other core infrastructural framework components also is provided.

CCSP High Level Architecture Overview



CCSP High Level Architecture

Above figure illustrates the core building blocks of the CCSP Platform at a high level along with the IPC mechanism used to communicate with each other. These core components are explained in later sections.

The architecture is structured into two layers:

1. CCSP Services Layer
2. Core Device Platform

The CCSP components in the CCSP Services Layer communicate among each other via the CCSP internal message bus. The CCSP message bus is based on D-Bus/R-Bus.

The Core Device Platform includes the Linux OS libraries, Free and Open Source Software (FOSS) and drivers accessible via Hardware Abstraction Layers (HAL).

CCSP components in the CCSP Services Layer call into OS primitives via POSIX interfaces. Since the CCSP platform is implemented and optimized for Linux OS, there is no need for an additional RTOS abstraction Layer. The necessary abstraction is provided by POSIX.

CCSP Component Model

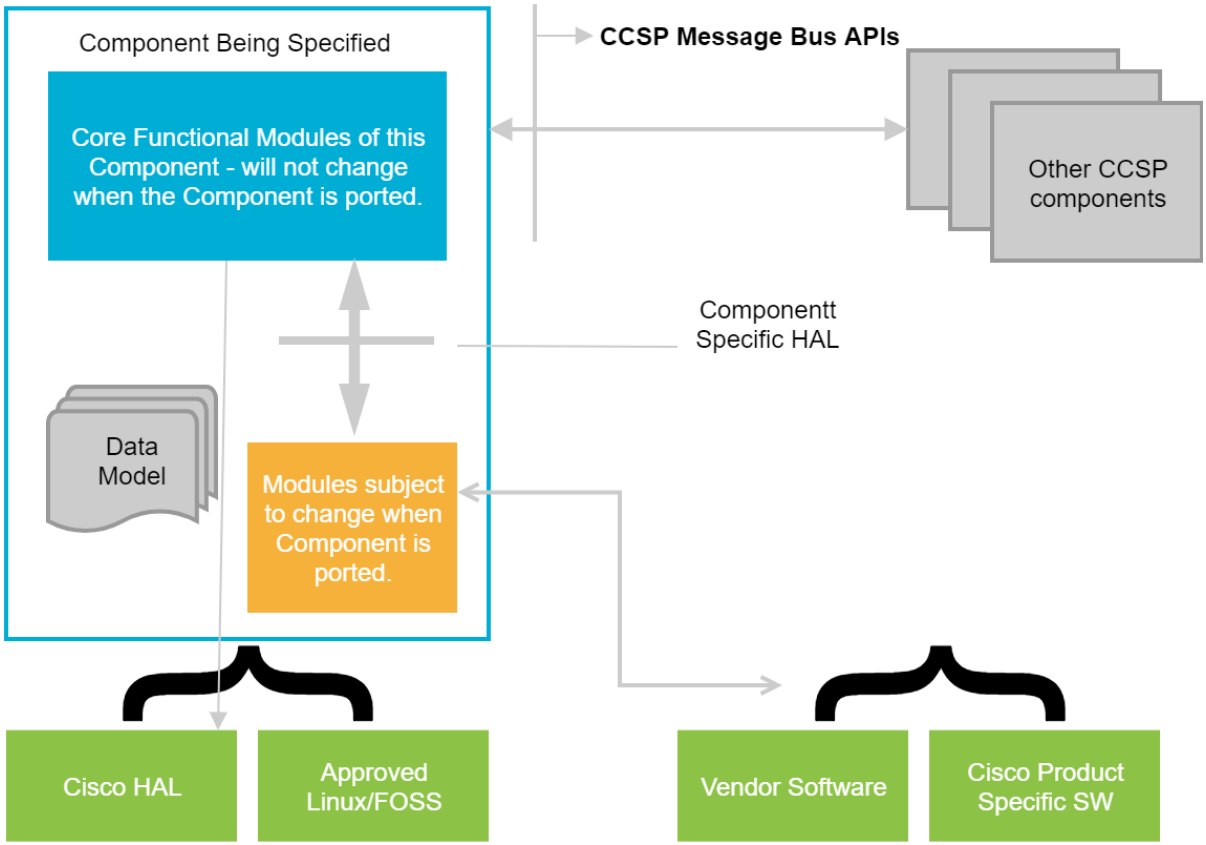
A CCSP component is one or more run-time processes, which consist of a reusable set of software to provide a defined set of service(s). A CCSP component can send and/or receive and handle requests via the CCSP Message Bus. All CCSP components extend from a Base CCSP Component that defines the core methods common to all CCSP components.

For performance optimization, more than one CCSP components can combine into a single run time process or decoupled into individual processes, without requiring any software change to the component. Having this capability allows each Component to be instantiated and work in its own process during testing and verification but later combined with other components into a single process (for performance optimization) in production environments. It also aids in resolving and isolating issues such as crashes, memory clobbers etc.

Component Interfaces

All components implement the base component interface as described in CCSP Base Component Interfaces.

The base interface defines core APIs that are common to all CCSP components. In addition each component provides well defined and abstracted APIs to expose component specific functionality based on the guidelines



CCSP Component Interfaces

Above diagram illustrates the component interfaces. It shows

- Northbound public CCSP Message Bus interfaces
- Internal portability and abstraction interfaces (In other words components may require a module that ensures portability across multiple operating environments)
- Southbound “common” interfaces (e.g., common FOSS libraries)

All CCSP Message Bus interfaces of the component must be made available via D-Bus/R-Bus introspection XML. The introspection XML defines all the interfaces and associated methods/APIs along with input and output parameters. The introspection XML is then used by a D-Bus/R-Bus binding tool to auto-generate client proxy and Adapter bindings.

Refer to https://www.freedesktop.org/wiki/Software/dbus/ for details on the D-Bus introspection XML and supported argument types.

CCSP Base Component Interfaces

This section defines the CCSP Base Component Interface. All CCSP components should implement these interfaces. The APIs defined in this section are the minimal set that each component should implement. More APIs will be added as needed.

The following illustration of the Base Component interface uses D-bus/R-Bus introspection XML to define methods.

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="/com/cisco/spvtg/ccsp/CCSPComponent">
<interface name="com.cisco.spvtg.ccsp.baseInterface">

<method name="initialize">
<arg type="i" name="status" direction="out" />
</method>
<method name="finalize">
<arg type="i" name="status" direction="out" />
</method>
<!--
This API frees up resources such as allocated memory, flush caches etc, if possible.
This is invoked by Test and Diagnostic Manager, as a proactive measure, when it detects low memory conditions.
-->
<method name="freeResources">
<arg type="i" name="priority" direction="in" />
<arg type="i" name="status" direction="out" />
</method>
<!-- Data model parameters "set" APIs
typedef struct {
const char *parameterName;
unsigned char *parameterValue;
dataType_e type;
} parameterValStruct_t;

typedef enum {
ccsp_string = 0,
ccsp_int,
ccsp_unsignedInt,
ccsp_boolean,
ccsp_dateTime,
ccsp_base64,
ccsp_long,
ccsp_unsignedLong,
ccsp_float,
```

```
ccsp_double,
ccsp_byte,  // char
(any other simple type that I may have missed),
ccsp_none
} datatype_e
-->
<method name="setParameterValues">
<arg type="i" name="sessionId" direction="in" />
<arg type="u" name="writeID" direction="in" />
<arg type="a(ssi)" name="parameterValStruct" direction="in" />
<arg type="i" name="size" direction="in" />
<arg type="b" name="commit" direction="in" />
<arg type="i" name="status" direction="out" />
</method>
<method name="setCommit">
<arg type="i" name="sessionId" direction="in" />
<arg type="u" name="writeID" direction="in" />
<arg type="b" name="commit" direction="in" />
<arg type="i" name="status" direction="out" />
</method>
<!-- Data model parameters "get" APIs -->
<method name="getParameterValues">
<arg type="as" name="parameterNames" direction="in" />
<arg type="i" name="size" direction="in" />
<arg type="a(ss)" name="parameterValStruct" direction="out" />
<arg type="i" name="status" direction="out" />
</method>
<!--
This API sets the attributes on data model parameters
typedef struct {
const char* parameterName;
boolean notificationChanged;
boolean notification;
enum access_e access; // (CCSP_RO, CCSP_RW, CCSP_WO)
boolean accessControlChanged;
unsigned int accessControlBitmask;
// 0x00000000 ACS
// 0x00000001 XMPP
// 0x00000002 CLI
// 0x00000004 WebUI
```

```
// ...

// 0xFFFFFFFF ANYBODY (reserved and default value for all parameters)

} parameterAttribStruct_t;

-->

<method name="setParameterAttributes">

<arg type="i" name="sessionId" direction="in" />

<arg type="a(sbbibu)" name="parameterAttributeStruct" direction="in" />

<arg type="i" name="size" direction="in" />

<arg type="i" name="status" direction="out" />

</method>

<method name="getParameterAttributes">

<arg type="as" name="parameterNames" direction="in" />

<arg type="i" name="size" direction="in" />

<arg type="a(sbii)" name="parameterAttributeStruct" direction="out" />

<arg type="i" name="status" direction="out" />

</method>

<!--

This API adds a row to a table object. The object name is a partial path and must end with a "." (dot). The API returns the instance number of the row.

-->

<method name="AddTblRow">

<arg type="i" name="sessionId" direction="in" />

<arg type="s" name="objectName" direction="in" />

<arg type="i" name="size" direction="in" />

<arg type="u" name="instanceNumber" direction="out" />

<arg type="i" name="status" direction="out" />

</method>

<!--

This API deletes a row from the table object. The object name is a partial path and must end with a "." (dot) after the instance number.

-->

<method name="DeleteTblRow">

<arg type="i" name="sessionId" direction="in" />

<arg type="u" name="writeID" direction="in" />

<arg type="s" name="objectName" direction="in" />

<arg type="i" name="size" direction="in" />

<arg type="i" name="status" direction="out" />

</method>

<!--

This API is used to return the supported parameter names under a data model object parameterName is either a complete Parameter name, or a partial path name of an object.

nextLevel
```

```
If false, the response MUST contain the Parameter or object whose name exactly matches the ParameterPath argument, plus all Parameters and objects that are descendents of the object given by the ParameterPath argument, if any (all levels below the specified object in the object hierarchy).

If true, the response MUST contain all Parameters and objects that are next-level children of the object given by the ParameterPath argument, if any.

parameterInfoStruct is defined as:

typedef struct {

comst char *name;

boolean writable;

}

-->

<method name="getParameterNames">

<arg type="s" name="parameterName" direction="in" />

<arg type="b" name="nextLevel" direction="in" />

<arg type="a(sb)" name="parameterInfoStruct" direction="out" />

<arg type="i" name="status" direction="out" />

</method>

<!--

This API is used in diagnostic mode. This must be used asynchronously. The use case is that the Test and Diagnostic Manager (TDM) CCSP component can leverage this feature in the Component Registrar to validate parameter types. The TDM sends commands to other components to run diagnostics. The TDM invokes a buscheck() request to each component one at a time in diagnostic mode. When each component receives buscheck(), it invokes the namespace type check API in the Component Registrar for each of the data model parameters accessed by this component and owned by another component. The Component Registrar verifies that each data model parameter is registered by a component and that the data model type specified in the API is the same as the data model type registered by the “owner” component. The component sends TDM a response to buscheck() with all checked parameter names and PASS/FAIL for each parameter. If during buscheck(), it is found that there are missing or unregistered parameters, appropriate errors are flagged.

-->

<method name="busCheck">

<arg type="i" name="status" direction="out" />

</method>

<!--

Signal contains the following information:

typedef struct {

const char *parameterName;

const char* oldValue;

const char* newValue;

dataType_e type;

 // data type

const char* subsystem_prefix; // subsystem prefix of the namespace

unsigned int writeID;

} parameterSigStruct_t;

-->

<signal name="ParameterValueChangeSignal">

<arg type="a(sssisu)" name="parameterSigStruct" direction="out" />
```

```
<arg type="i" name="size" direction="out" />
</signal>
```

CCSP Base Component Interfaces

CCSP Framework

The CCSP framework provides basic mechanisms for component management and message dispatching. This section describes all the Framework elements shown in CCSP High Level Architecture.

CCSP Message Bus

CCSP Message Bus is used by the CCSP components to communicate with each other and send notifications to registered subscribers in a single processor environment. CCSP message bus uses D-Bus/R-Bus for IPC.

For more details on D-Bus and R-Bus , refer CCSP Message Bus

CCSP Message Bus Adapter

CCSP Message Bus Adapter is a core framework Component that enables inter-processor communication. The other processor may also run CCSP Message Bus Adapter to facilitate communication.

The internal implementation of the CCSP Message Bus Adapter may use TCP/IP sockets for communicating with the peer message bus adapter running on the other processor.

CCSP Component Registrar (CR)

CCSP Component Registrar is a centralized database of all registered CCSP components/services. It supports dynamic learning of registered components and supported capabilities - message types/namespaces. In addition it signals events to registered subscribers when components get unregistered, either gracefully or due to a fault condition.

It can also be used to signal device profile changes where a profile is a pre-determined minimal set of active components needed to behave as an embedded system. The device profile and the components that make up the device profile, is ingested by the Component Registrar as a configuration file during initial boot up. The names of the component and the names in the configuration file must match.

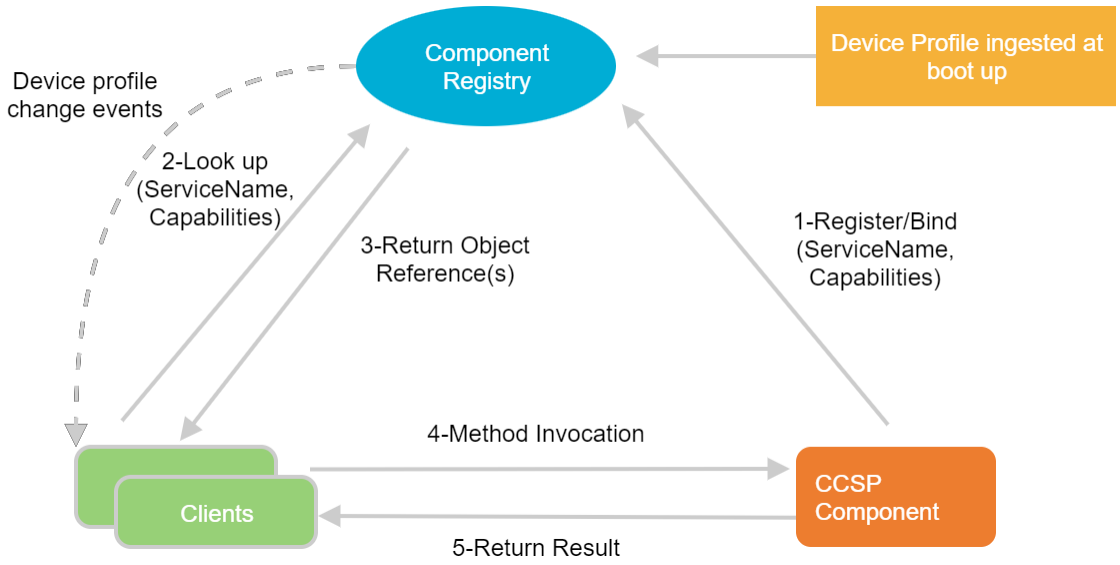
This enables an application to dynamically reflect the current capabilities of the device by a notification from the Component Registry if the system Profile changes.

At boot time the CCSP components register and advertise their supported capabilities to the Component Registry. They may also choose to subscribe for device profile change events. The following table is a simple illustration of what Component Registrar contains.

Component Name	Version	D-Bus Path	Namespace	Successfully Registered
com.cisco.spvtg.ccsp.PAM	1	"/com/cisco/spvtg/ccsp/PAM"	Device.DeviceInfo.Manufacturer Device.DeviceInfo.ManufacturerOUI Device.DeviceInfo.SerialNumber Device.DeviceInfo.SoftwareVersion	True
com.cisco.spvtg.ccsp.SSD	1	"/com/cisco/spvtg/ccsp/SSD"	Device.SoftwareModules. DeploymentUnitNumberOfEntries Device.SoftwareModules.DeploymentUnit.{i}.UUID Device.SoftwareModules.DeploymentUnit.{i}.Name Device.SoftwareModules.DeploymentUnit.{i}.UAlias	True
com.cisco.spvtg.ccsp.PSM	1	"/com/cisco/spvtg/ccsp/PSM"	com.cisco.spvtg.ccsp.command.factoryReset	True
com.cisco.spvtg.ccsp.TDM	1	"/com/cisco/spvtg/ccsp/TDM"	“Device.IP.Diagnostics.”	True

The Registry is used by applications to perform Service Discovery based on capabilities. The Protocol Agents in the CCSP control plane, for instance, queries the CR based on capabilities and data model namespace supported and routes messages to those components.

Below figure illustrates the features of Component Registry and how it is used by other internal components and client applications.



Component Registry

Protocol Agents

Protocol Agents are components that directly interface to the Cloud. These components facilitate remote administration/management of the device. The Protocol Agents provides the necessary abstraction to the internal CCSP architecture and components for interacting with the Cloud. The internal CCSP components are not required to be aware of any protocol specific details on the cloud interfaces. . The TR69 ACS uses HTTP/SOAP to communicate with the device. The TR69 Protocol Agent hides all the protocol specific details and communicates internally using internal namespaces and APIs over the CCSP message bus.

As another example, an SNMP Protocol Agent may also be instantiated to support device management via SNMP. Again this Protocol Agent hides all the protocol specific details and communicates internally using the internal namespaces and APS over the CCSP message bus.

There may be multiple instances of the same Protocol Agent in a CCSP subsystem, one for each WAN facing interface. For example if a device has multiple WAN facing IP addresses, a TR-069 Protocol Agent may be instantiated on each WAN facing IP address (if needed).

Protocol Agents perform the following functions:

- Authenticate and establish secure session with their corresponding cloud adaptors during initialization.
- Perform low level protocol handling for downstream and upstream traffic.
- Routes cloud messages to internal functional components registered for consumption/processing of those namespaces by looking up which component handles a particular namespace via the Component Registrar.
- “Action” Normalization
    - Internal CCSP components use normalized action / RPC methods to process requests. The normalized methods are defined in the base interface supported by all CCSP components.
- Protocol Agents may define an XML based Mapping Schema that defines the mapping from external constructs to internal constructs. These constructs may include:
    - External Objects and parameters to Internal Objects and parameters
    - Format Conversions
    - Internal Errors to External Errors
- Signals errors and routes responses to corresponding cloud adapters
- Performs all transactions as an atomic operation.
    - For example, if a transaction involves a “SetParameterList” action of 10 key-value pairs, then the Protocol Agent applies the changes to all of the specified key-value pairs atomically. That is, either all of the value changes are applied together, or none of the changes are applied at all.
    - In cases where a single transaction involves multiple Components, the PA aggregates the response from all the components before sending the results back to PA.
- Manages the order of operations within a transaction and across transactions.
    - The PA serializes all transactions from its cloud interface and only allows one transaction at a time.
- Generates and maintains Context for asynchronous notifications and transactions. This is explained in more details in the next section.

Asynchronous Notifications (Eventing)

The Protocol Agents (PA) maintains a notification table of external notification requests on the Data model parameters. On receiving requests for notifications/eventing topics from the cloud, the protocol agents

- Maps the cloud namespace to internal name space by looking up its schema mapper
- Adds the parameters to the notification table including their internal name space.
- Queries the Component Registrar (CR) using the internal namespace, to get the D-Bus path of the Functional Components that the request is intended for.
- Invokes the D-Bus API SetParameterAttributes() from the base component interface on the component to enable notifications on the parameters

The Functional components (FC) maintain a notification counter for each parameter that is set to 0 during initialization.

- When notifications are enabled on the parameter via setParameterAttributes (), the corresponding counter is incremented. Conversely when notifications are turned off, the counter is decremented. This is done because D-bus does not provide a mechanism to inform the component to stop generating signals if there are no registered subscribers for the signal. The component has no knowledge of signal recipients. Subscribers register interest with the D-Bus daemon for the signal. The D-Bus routes the signal to all subscribers.

The Functional Component defines a common signal to notify changes on all the data model parameters it supports. It generates that signal on D-Bus if and only if the notification counter for any parameter it supports is greater than 0. The signal contains the parameter name and its new value (old value may also be included, if needed).

The Protocol Agents subscribes to the component's signal with D-Bus.

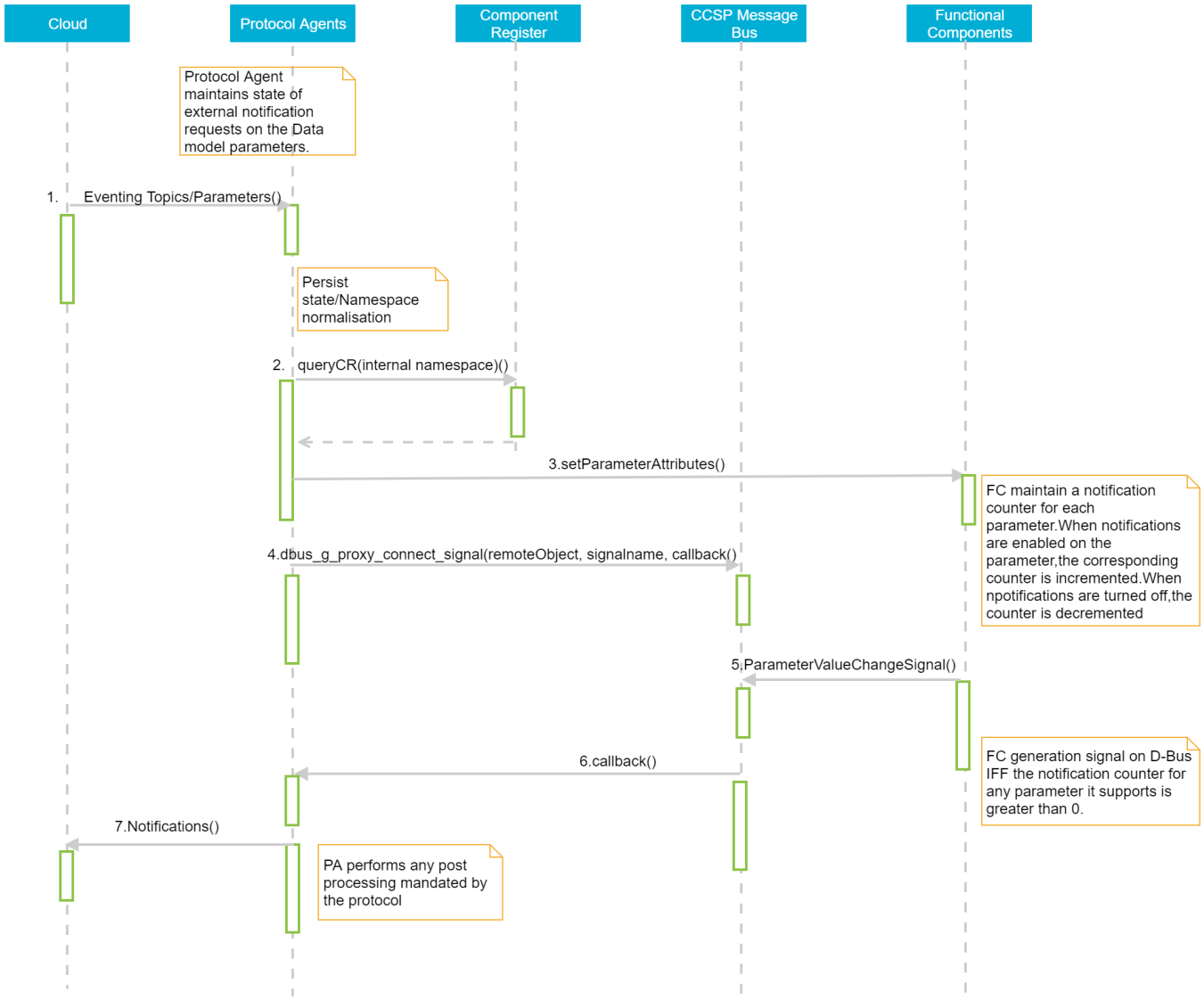
When the value changes for a parameter whose notification has been turned on, the Functional component generates the signal on D-Bus. The signal message contains the name of the signal; the bus name of the process sending the signal and the delta.

The PA gets the notification via D-Bus and notifies the change to its cloud adapter by looking up the notification table and performing any post processing mandated by the protocol.

If the notifications are turned off by the Cloud, the protocol agents,

- Deregisters the notifications by calling setParameterAttributes() on the Functional Component
    - The Functional Component decrement the notification counter. If the counter < 1, then Functional Component should not stop generating notifications on that signal.
- Clears/updates the notification table
- Unsubscribes the component's signal on the message bus if there are no other notifications active in that component.

The Protocol Agents must police themselves such that notifications should only be turned off if they had been turned on previously by them.



Asynchronous Notification

CCSP OS Abstraction Layer

CCSP platform is implemented and optimized for Linux OS. As such the OS abstraction is provided by the POSIX APIs. All CCSP components must use the POSIX APIs for all OS services such as threads, mutual exclusion, semaphores, shared memory etc.

Hardware Abstraction Layer (HAL)

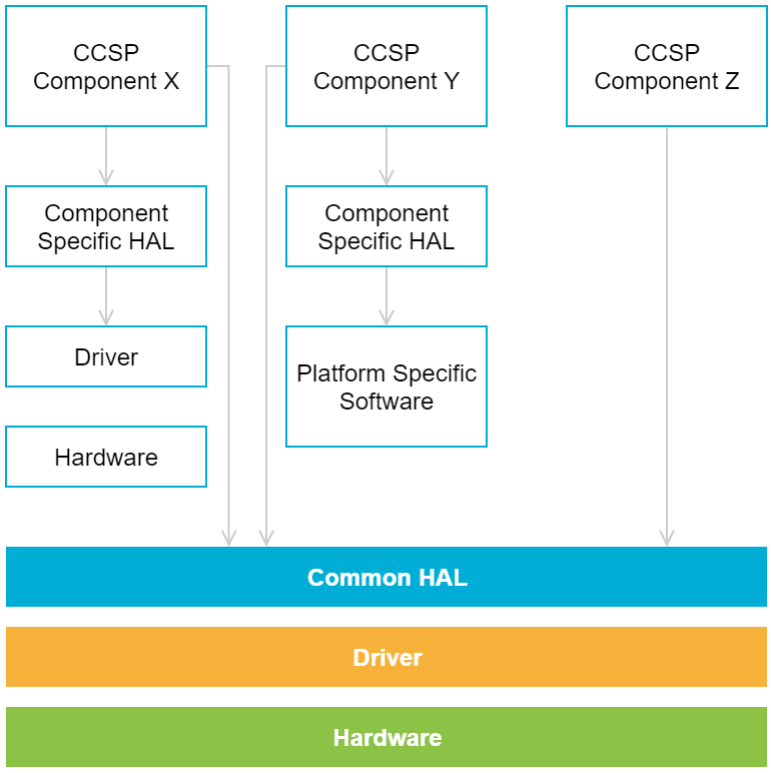
HALs provides the necessary abstraction to the CCSP components to interface with SoC vendor specific hardware components such as audio/video encoders/decoders. The HAL is a thin layer above SoC vendor's driver software distribution. This decouples the CCSP framework from any particular SoC platform.

The HAL is made up of two layers namely – Hardware Independent Layer and Hardware Dependent Layer. CCSP components must be written to hardware independent layer.

The following provides some suggestions or recommendations for defining and an Abstraction Layer for hardware access.

1.The components may define a component specific HAL to hardware drivers that are only used by that component. For instance the Video DRM Termination Manager (VDTM) Component may define a common DRM HAL that is not tied to a specific DRM. The VDTM component is the only component that uses this abstraction layer and therefore is not common to all the components. However using this component specific HAL abstraction allows the component to quickly integrate with multiple DRMs with minimal changes. It also eliminates any unnecessary dependency, of the component, with the common HAL.

2. A Common HAL provides the necessary abstraction to all the CCSP components to interface with common hardware components. Figure below illustrates the use of common HAL along with Component specific hardware abstraction.



Common HAL and Component Specific HAL

3. For the FOSS libraries that use common industry standard interfaces that are unlikely to change , such as DirectFB, OpenGL ES 2.0, that are ported by the SoC vendor on their respective platform, CCSP components should call directly into the interfaces exposed by these libraries. There is no additional layer of abstraction above and beyond what is provided by the FOSS libraries themselves. Figure-1 illustrates this clearly.

4. For certain FOSS libraries and third party licensed software, however, if the interfaces can potentially change or if the library is swapped, for performance gains, by another FOSS library providing equivalent services, but with different interfaces, then it is recommended to create an abstraction layer as part of the common HAL

5. If there is a use case for all CCSP components to access certain hardware capabilities, then such an interface should be made available in the CCSP Base component interface. This interface, however, should be part of the common HAL.

Data Plane

This section is intended to explicitly list out best practices and guide lines for fast communication between components and processes running on a single processor and across multi processors.

Single Process Communication

Components within a single process should communicate via well-defined APIs. The APIs should abstract, encapsulate and hide any internal representation of the component.

In order to pass large amounts of data between CCSP components, memory is allocated from the Process heap by the component generating the content. The reference (pointer) to this heap can then be passed to other interested components as parameter. This avoids making unnecessary copies of the data being shared. However, this requires the format and structure of the data to be known in order for the components to process data correctly.

Message schemas along with their versions should be defined and published for all communication between CCSP components.

It is recommended to pass as parameters, the version of the message format of the data, in addition to reference to the heap. This allows for the detection of any incompatibilities if the message format is changed without being communicated to other components.

Care must be taken to free the memory after processing is complete.

Inter Process communication

Inter process communication is facilitated by CCSP Message Bus that uses D-Bus/R-Bus IPC. Components across processes communicate via well-defined APIs using the bus daemon. The D-bus/R-Bus daemon provides a publish-subscribe interface and routes signals/events to all registered subscribers.

In order to move large amounts of data between components across process boundaries, UNIX file descriptors should be passed as parameters over D-Bus/R-Bus. The idea is to use D-Bus/R-Bus for IPC method calls between a client and a server and a dedicated pipe for passing large results from the server back to the client. This allows very fast transmission rates.

The file descriptor could be a pointer to the shared memory object allocated via shm_open ()

As mentioned in Single Process Communication above, this approach too requires the format and structure of the data to be known in advance for the components to process data correctly. It is recommended to define a common message format structure and pass version of the message format to the communicating component along with the file descriptor.

Inter Processor communication

Inter processor communication is facilitated by the CCSP Message Bus Adapter. As explained in CCSP Framework, the Message Bus Adapter is a component that runs on all the processor cores and uses TCP/IP sockets to communicate with each other. This allows the processor cores to potentially run different OS stacks and yet be able to seamlessly communicate with each other.

On the CCSP frame work the message bus Adapter exposes its services via APIs on the D-Bus/R-Bus. Other CCSP components use these APIs on the message bus Adapter communicate with components running on other processor cores.

Inter Subsystem Communication

CCSP architecture is intended to be sufficiently flexible to allow communication between CCSP components that reside across multiple subsystems. This document references several complex use cases with multiple subsystems. Simpler products without this complexity are also supported. This document defines the following subsystems along with their associated prefix.

- Router (eRT)
- Cable Modem (eCM)

It should be noted that the Cable Modem subsystem and the Router subsystem will always coexist in a DOCSIS based WAN front end Gateway devices.

These subsystems may be implemented using the following configurations.

- Across processor boundaries, or
- In the same processor as logical subsystems with potentially different session bus for IPC, or
- In the same processor all using the same session bus for IPC.

Before continuing with the inter subsystem communication architecture, it will be useful to highlight some of the capabilities of the CCSP Message Bus, in order to set the context and also emphasize that we are leveraging the message bus capabilities to its fullest extent and not redefining something that is already tested and proven.

More details available at D-Bus Remote Communication Capabilities

Inter Subsystem Communication Architecture

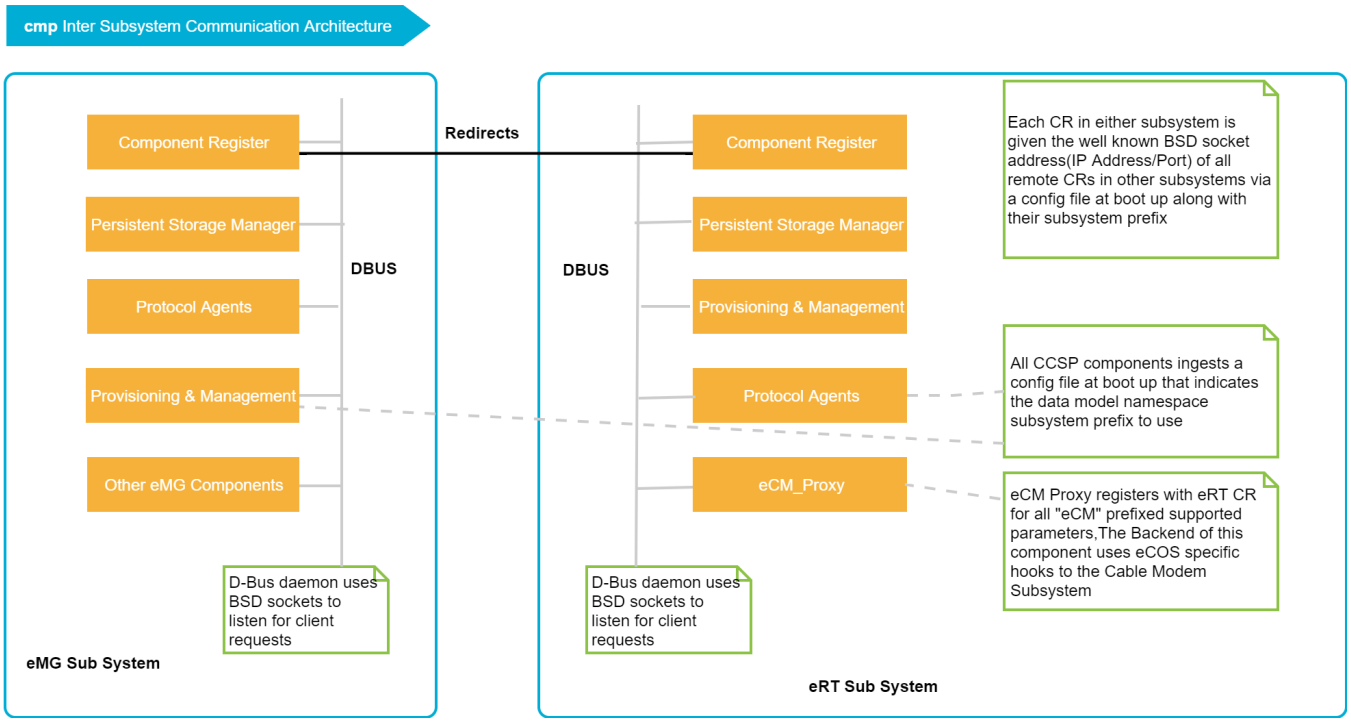
The following architecture is based on the assumption that each subsystem is managed independently by external Cloud Management Servers. However the assumption is not a restriction and works/scales even if all subsystem were managed by a single subsystem.

The architecture is flexible in that it works in any of the scenarios in which subsystems may be implemented. That is across processor or uni-processor with logically separated subsystems.

The details of the architecture are described next followed by specific use cases.

- Each subsystem has its own D-bus session bus that uses BSD sockets, by default, to listen for client requests. The clients connect to the session bus using this default BSD socket. The default behavior can easily changed to use UNIX domain name sockets for local IPC via the D-Bus configuration file. As indicated earlier the D-Bus is able to "listen" on multiple addresses (BSD sockets, UNIX domain name sockets etc).
- There is a Component Registrar (CR), one for each subsystem. In other words each subsystem has its own CR.
    - Each CR is given the well-known BSD socket address (IP Address/Port) of all remote CRs in other subsystems via a configuration file at boot up along with their associated subsystem prefix. This could also be just the D-Bus path names of the CR in all subsystem. The path name contains the subsystem prefix and uniquely maps to a well-known IP Address/Port. The common D-Bus bindings can hide such details from the component developers and provide the necessary abstraction.
    - Each CR only registers fully qualified data model parameters for its local subsystem with no exceptions. In other words there are no object level registrations in the CR.
        - All registered data model parameters have a subsystem prefix associated to their names.
        - Any cross referenced data model parameter in a subsystem will NOT straddle across subsystem boundary. Any cross referencing is local to a subsystem.
            - If a data model parameter value refers to another data model object (cross referencing/indirection), then that value is not required to have a subsystem prefix.
            - If the ACS (or any other component) wishes to get further details on the cross referenced parameter value, it will use the subsystem prefix of the "current" subsystem in which the component belongs.
    - Each CR can discover components from other subsystem by redirecting the request to the CR on the corresponding subsystem based on subsystem prefix.
    - The CR includes the D-Bus path (BSD socket address) of the remote CR as part of the response when discovering components supporting namespace. The Functional Components may choose to cache/learn the remote CR so that subsequent requests for discovery of parameters using remote subsystem prefix can be directed to this remote CR directly instead of going to the local CR. The idea is very similar to ICMP redirects used on IP networks. This is dynamic and does not require having any a prior knowledge of any remote CR. In turn this provides a level of performance optimization to the Functional Components.
- Each Functional Component in the subsystem ingests a configuration file at boot up that gives
    - The subsystem prefix that the component belongs to
    - The address (D-Bus path / BSD socket) of the local subsystem CR.
- Using a configuration file allows the common CCSP components to not hard code a prefix to the namespace, thereby preventing duplication of code. However, the components must be able to process the prefix when operations (Set/Get) are performed on the parameters.
    - The idea of CR advertising a subsystem prefix at run time was also considered. However that does not work if different subsystems are on the same processor and using the same session bus. The goal is to support all use cases/architectures without requiring (or minimizing) any change to the CCSP components.
- The components use the subsystem prefix from the configuration file to register their supported data model namespaces with their local Component Registrar.
- If a CCSP functional component wants to "Set/Get" a parameter that is owned by a component in another subsystem, then it must have a prior knowledge of the prefix for the other subsystem to the parameter name. In other words the prefix for all dependencies of a component must be known in advance (compile time).
    - The local component discovers the remote component registered to own the namespace via its local CR.
    - The CR responds with the D-Bus path of the remote component and the D- Bus address of the remote session bus. As mentioned earlier, including the subsystem prefix in name of the component can hide the session bus address details from the components. The common D-Bus bindings can map the name to session bus addresses and provide the necessary abstraction.
    - The local component communicates directly with the remote component.
- There are multiple Protocol Agents (PA), one for each WAN facing IP address/subsystem.
    - Similar to the Functional Components, each PA in a subsystem is given the address (Dbus path/ BSD socket) of the local CR along with the local subsystem prefix via the configuration file at boot up.
    - Each PA has a mapper file that maps external cloud namespace to internal CCSP namespace among other things. The PA mapper file will also indicate Data model parameters that map to another subsystem along with the subsystem prefix.
    - Similar to the Functional Components, the PA discovers the remote subsystem component registered to own the namespace via its local CR and communicates with it directly over D-Bus BSD sockets.

Figure below illustrates the inter subsystem communication architecture.



Inter Subsystem Communication Architecture

Use Cases

eCM and eRT communication

- The eCM and eRT subsystems will always coexists in all use cases that use DOCSIS as their WAN frontend.
- An eCM Message Bus Adapter component registers with the eRT CR for all "eCM" prefixed supported parameters.
- Any request for eCM parameters will be routed to this component which uses SNMP or other protocols to talk to the cable modem firmware in the backend

Session Integrity

Some CCSP components such as the TR69 Protocol Agents require session integrity. In other words from the time a session is initiated until the session is terminated, the device must ensure the transactional integrity of all Parameters accessible via the protocol Agent. During the course of a session, all configurable Parameters of the device must appear to the Protocol Agent as a consistent set modified only by the corresponding ACS. Throughout the session the device must shield the ACS from seeing any updates to the Parameters performed by other entities/components. This includes the values of configurable parameters as well as presence or absence of configurable parameters and objects. Session integrity can be achieved by leveraging Message Bus signalling.

Access Control

As with session integrity, some components such as TR69 Protocol Agents require write access control on data model parameters. A TR69 ACS may choose to have exclusive write access on specified parameters and prohibit other components from modifying those parameters. It should be noted that all Protocol Agents have write access to all parameters at all times. This requirement is to allow for cases when a protocol agent may need to assert exclusive write access to the specified parameters.

In order to support multiple cloud interfaces it is essential that the internal CCSP architecture and components be agnostic to any cloud semantics.