Switch Abstraction Interface v0.9.1

Authors

Microsoft: Dmitry Malloy, Dave Maltz, CJ Williams

Dell: Mike Lazar, Shree Murthy, Sanjay Sane

Facebook: Omar Baldonado, Tian Fang, Adam Simpkins

Broadcom: Ben Gale

Intel: Uri Cummings, Dan Daly, and Miles Penner

Mellanox: Matty Kadosh, Itai Baz and Aviad Raveh

Contents

List of Changes

Version	Changes	Name	Date
0.9	Initial Draft		2014/10/06
0.9.1	Remove header files from the doc, plus various changes based on OCP inputs		2014/12/31

1 Introduction

This specification defines an abstraction interface for switching ASICs. The interface is designed to provide a vendor-independent way of controlling switching entities like hardware ASIC's or NPU's in a uniform manner. This specification also allows exposing vendor-specific functionality and extensions to existing features.

2 Intended audience

This document is intended primarily for the programmers who plan to use SAI API or develop SAI extensions.

3 Definitions

Adapter is a pluggable code module, supplied by either a vendor or control plane stack owner, that contains either ASIC SDK code itself or client module for ASIC SDK hosted in external process and implements the interfaces described in this specification; for all practical purposes it is equivalent of a "user-mode driver".

Adapter host is a Microsoft or vendor-supplied component that loads the adapter and exposes its functionality to the control plane stack.[1]

Control Plane stack is a set of software components that interacts with Adapter Host and provides higher level programming support to network control applications.

Switching Entity is an instance of a switching object; it represents a physical ASIC or software switch.

4 High-Level design notes

Switching adapters are user-mode drivers, typically supplied by ASIC vendors. Adapters are registered with the switching stack and then can be loaded as needed. It is a responsibility of the adapter to discover and bind to the specified underlying hardware, including loading of or attaching to kernel-mode drivers if needed.

Adapters are expected to be as simple as possible, ideally simple wrappers around vendor's SDKs. Our design strives to push the bookkeeping complexity from adapter into the adapter host wherever possible.

The adapter module is loaded into a hosting process ("adapter host") and then initialized. During initialization the adapter initiates discovery process of the specified instance of a switching entity. A switching entity is a top-level object in this API.

There cloud be multiple Adapter Hosts running at the same time. Each Adapter Host is responsible for managing a portion of functions within the ASIC. For example, an IP routing Adapter Host is responsible for syncing IP routes to the ASIC, while a counter Adapter Host is responsible for reading counters from the ASIC.

[1] While design and API's of the adapter host are outside of the scope of this spec, some interactions with adapter host are covered in this document.

Key assumptions, design decisions and API semantic clarifications:

- CRUD (Create/Read/Update/Delete) based API to manage the sai objects.
- Adapter is not the source of the critical persisted state. It can crash or be shut down and the above stack will be able to recover from such an event.
- Due to resiliency requirements, the adapter host manages exactly one instance of a switching entity. Therefore “switch instance id” that comes from SDK is contained in the adapter code and doesn’t percolate into SAI API’s.
- If specific attributes are required for element creation, but not specified as part of the create function, the default value from either the container element or the top-level switching entity is used.
- Deletion of an object that is referenced should fail (e.g. removal of router when interface exist).

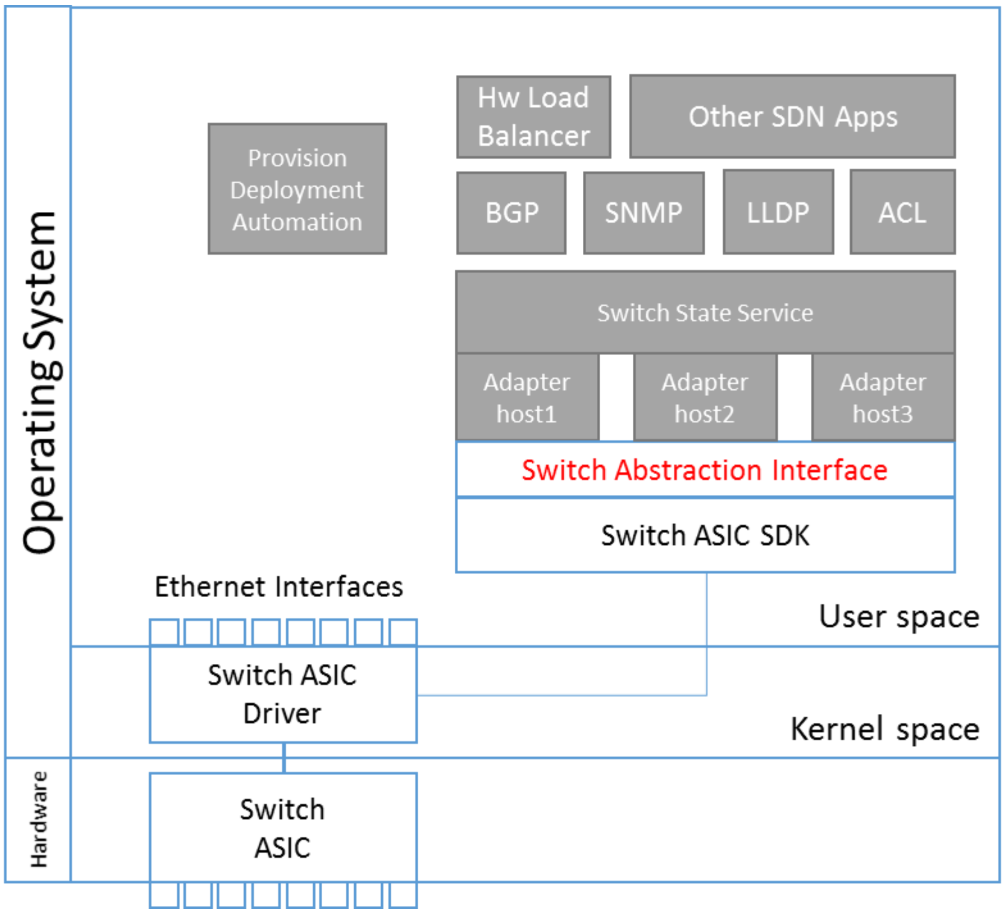

Figure 1 SAI in a plausible switch system architecture

5 API description

Proposed interface is a local interface between the Adapter Host and Adapter. ASIC functionality is exposed to the rest of the system by the Adapter Host through other mechanisms, which are not part of this specification.

The API is designed to be platform-agnostic (*nix/ Windows/etc…).

The API is attribute-based to minimize compatibility issues with versioning of structures and to allow API extensibility.

The API is a collection of C-style interfaces exposed from the adapter. These interfaces are grouped into three categories:

- Mandatory functionality. This is a set of “core” interfaces which are required to build a basic routing appliance. All vendors must support these interfaces and the switching stack will fail loading of the adapter if any of these interfaces are missing.

- Optional functionality. This is a set of additional interfaces which are not required, but enable various scenarios in a modern datacenter. Definition of these interfaces is common for all the vendors. These interfaces are not required by the switching stack to be exposed from the adapter. However, they become required if a given system configuration references any of these features.

- Custom functionality. This is a set of interfaces that are unique for a vendor and is not standardized. The default adapter host is not aware of these interfaces and a custom adapter host can be supplied by the vendor to expose these interfaces to the switching stack. We also intend to propose a more generic framework for exposing such functionality through default supplied adapter host in further drafts.

Interfaces are discovered by querying for method tables through the mechanism described in “Functionality query” section.

A near-future version of this specification will provide a mechanism for extending the functionality of mandatory and optional interfaces though custom attributes definition.

6 Adapter startup/shutdown sequence

Adapter Host is provided with the information about binary module with adapter code that must be loaded during startup.

After adapter module load, adapter host acquires addresses of the three well-known functions: sai_api_initialize(), sai_api_query() and sai_api_uninitialize ().

After that sai_api_initialize() is called. Function is supplied with method table of services provided by the adapter host to adapter module. Note: SDK initialization should NOT be performed here. This function is just to take care of any platform-specific differences related to module loading, and generally should be very simple. After initialization, sai_api_query() can be used for retrieval of various methods tables for SAI functionalities.

Support for functionality versioning: if either signature or semantics of any function of a method table changes, new sai_api_id must be introduced (e.g. SAI_API_VLAN2). Adapter Host starts inquiry of functionality method table from the highest version that is known to Adapter Host. If newer API is not available, Adapter Host is free to fallback to any earlier or the first version. If the first version of the mandatory functionality category is not available, Adapter Host fails the adapter startup process.

Once all mandatory functionalities are successfully queried, optional functionalities are requested. Failure to discover optional functionality does not lead to a failure of the system startup.

After all the functionalities being retrieved, Adapter Host proceeds to exercising SAI_API_SWITCH functionality.

For example, first method that Adapter Host calls is sai_initialize_switch() that performs full SDK initialization. In a system where there are multiple Adapter Hosts, only one of them calls the above function. Others will call sai_connect_switch() to connect to the SDK once the SDK has been initialized. The call to sai_connect_switch() will fail if the SDK hasn't been initialized. The sai_intialize_switch() function takes the following parameters:

- handle to a switch entity profile
- hardware id of that switch entity
- name of corresponding microcode
- callbacks table

The profile_id is used by the adapter to retrieve a profile which is a list of key-value string pairs that contain vendor-defined settings. Once the adapter has the profile id, it can use the function profile_get_value() and profile_get_next_value() from service_method_table_t to get the key-value string pairs, such as the SDK_LIBRARY_PATH, NPU_CONFIG_FILE_PATH. The service_method_table_t is provided by the adapter host during sai_api_intialize().

Format and contents of the "hardware id" string is defined by the vendor and contains enough information to identify the device. For example it can contain PCIe location.

Name of the microcode is a vendor-defined string if vendor allows loading of a different microcode module during startup. It can be either path to a module, or a name of a module, or something else. We may consider introducing having mandatory key in the profile (e.g. "microcode").

The function needs to discover the instance of the ASIC chip using "hardware id" string, map devices registers, query information to determine specific silicon revision, and given the microcode and the profile settings, initialize the SDK.

After this, driver enters normal operations and starts to handle calls from Adapter Host to various methods of different functionality tables. It is important to note that:

During shutdown, Adapter Host will invoke sai_switch_shutdown() as a counter-party for the initialize_switch() to perform SDK de-initialization. For Adapter Hosts which uses sai_connect_switch(), they will use sai_disconnect_switch() to disconnect from the SDK.

sai_api_uninitialize() does the cleanup of anything that was done in sai_api_initialize() and is a last call to the adapter before it's unloading.

Finally, Adapter Host unloads the adapter module.

7 Specification contents and timeline

For the list of accepted features please see the **History** section at the end of the specification.

Summary of supported functionalities is given in the table below:

Functionality	Description	Category	Version
sai_switch_api_t	Top-level switch object	Mandatory	0.9.1
sai_port_api_t	Port management	Mandatory	0.9.1
sai_fdb_api_t	Forwarding database	Mandatory	0.9.1
sai_vlan_api_t	VLAN management	Mandatory	0.9.1
sai_vr_api_t	Virtual router	Mandatory	0.9.1
sai_route_interface_api_t	Routing interface	Mandatory	0.9.1
sai_route_api_t	Routing table	Mandatory	0.9.1
sai_neighbor_api_t	Neighbor table	Mandatory	0.9.1
sai_next_hop_t	Next hop table	Mandatory	0.9.1
sai_next_hop_api_t	Next hop group	Mandatory	0.9.1
sai_qos_api_t	Quality of service	Mandatory	0.9.1
sai_acl_api_t	ACL management	Mandatory	0.9.1
	LAG	Mandatory	0.9.2
	STP	Mandatory	0.9.2
	Control packet send/recevie	Mandatory	0.9.2
	Port mirroring	Optional	TBD
	Tunneling	Optional	TBD
	Policer	Optional	TBD
	SFlow	Optional	TBD
	Datacenter bridging	Optional	TBD
	Openflow 1.x	Optional	TBD
	L3 Multicast groups	Optional	TBD

Areas that are proposed, considered or in progress, but not addressed in the current draft:

Functionality	Description
Bridges	Consider adding support for bridges
L2	L2 multicast packet forwarding table is postponed
Statistics	Refine the definitions of statistic blocks; make standards-compliant.
Statistics	Allow for vendor-specific diagnostics through stats

Statistics	Consider clear_stats_in_hadrware(). Its usage is a bit unclear and may break apps in multiple-readers scenarios.
Diagnostics	Provide tracing function in services_t table from Adapter Host.
Panel-to-switch-port table	Provide vendor-defined translation table from panel to switch ports, as well as port configuration setting (speeds, 10G/40G, copper/optics). This needs clarification ASAP.
sai_status_t	Encode functionality into sai_status_t codes. Consider using sai_api_id_t.
sai_status_t	Annotate methods with allowed error codes
	Split some structures to key and non-key versions
Function comments	Use doxygen
OEM-specific initialization	Provide a way to load and run OEM/ODM-supplied initialization code which may use adapter private APIs.
VLAN mapping	Provide external to internal VLAN mapping
FDB event throttling	Provide a way to throttle FDB notifications
Enumeration	Consider providing enumeration API for objects managed by the Adapter Host. Apart from debugging, usage is unclear if adapter is threated as stateless. For objects managed by the adapter (dynamic FDBs, etc…) only the callback mechanism is needed.
QoS	Per-port PFC
	Add MCAST ADMIN state to VRF (for IPV4/IPV6)
	Describe VRF ability to handle TTL1 in switch capabilities
	Describe VRF ability to have MAC_ADDRESS in capabilities
	Allow interfaces to have multiple DMACs (VRRP)
	Add finer grained flood controls (per-protocol) for switch object
	Consider adding per-port miss action drop support

8 Data Types (saitypes.h)

File contains cross-platform definitions for data types used in SAI.

- Defines cross-platform sai_ip_address_t as union of ipv4 and ipv6 addresses.
- Defines cross-platform sai_ip_prefix_t as union of ipv4 and ipv4 prefixes.
- The sai_mac_t is defined as uint8_t[6] in network order with mac[0] to be the first byte of the mac address.
- sai_attribute_value_t is the value of all sai attributes. It is defined as a union of all base sai types defined in this header file. sai_uint64_t u64 defined in the sai_attribute_value_t is used for all sai attribute values whose base type is uint8_t, uint16_t, uint32_t and uint64_t, whereas sai_int64_t s64 is used for all values whose base type is int8_t, int16_t, int32_t and int64_t.

9 Status codes (saistatus.h)

List of status codes returned from the SAI methods.

A number of attributes can be passed to the create_*sai_object* function where *sai_object* can be any sai object such as router interface, next hop. In case the create function call fails due to attribute releated errors such as invalid attribute, invalid attribute value, unsupported attribute, and unimplemented attribute, saistatus.h allows the return code to convey which attribute causes the error. The saistatus.h defines a base index and maximum index for all these four types of attribute related errors. When you get an error code that falls into their defined ranges, the offset from its corresponding base index denotes that attribute that causes the failure. For example, if you pass 5 attributes into a create_*sai_object* call, and the return code (rc) is 0x00010003. You can first use SAI_STATUS_IS_INVALID_ATTRIBUTE(rc) to determine if the error is due to invalid attribute or not. In this case, the answer is yes. Then, you can subtract the return code with SAI_STATUS_INVALID_ATTRIBUTE_0 to get the offset. In this case, you will get 3 meaning the 4th attribute in the call is an invalid attribute.

10 Functionality query (sai.h)

Mandatory. Switch adapter entry point for interface retrieval. If a signature or semantics of any method of an interface changes, new sai_api_t needs to be introduced, e.g. SAI_API_VLAN2. Adapter must also expose previous versions (at least the first version) of the interface to allow interoperability with older Adapter Hosts. The Adapter Host will always query an interface starting with the highest version that it knows about, falling back to previous version(s). If the first version of a mandatory interface is not available, Adapter Host fails the startup.

Adapter must keep track of which versions adapter host is using in order to be able to update the capabilities of the top-level Switching Entity.

11 Switch functionality (saiswitch.h)

Mandatory. This is a top-level object exposed from the adapter. This switch object controls the switch level behavior such as cut-through or store-forward, ecmp hash configuration.

12 Port functionality (saiport.h)

Mandatory. Implements physical port state manipulation as well as state (UP/DOWN) notifications. In the near future, it will also manage the port breakout mode.

13 Forwarding Database (MAC Table) functionality (saifdb.h)

Mandatory. Implements FDB entries manipulation as well as aging/learning notifications.

14 VLAN functionality (saivlan.h)

Mandatory. Implements VLAN management functions, such as add/remove port from a vlan.

15 Router functionality (sairouter.h)

Mandatory. Implements functions to manage virtual routers, such as creating and deleting virtual routers.

16 Router Interface functionality (sairouterintf.h)

Mandatory. Implements “router interface” object. The router interface is attached a specific virtual router. It can be either VLAN-based or port-based router interface. You can also specify source mac address for the router interface. However, if the ASIC does not support per-router interface source mac address, the source mac address will inherit from the source mac address from higher hierarchy such as virtual router or switch object.

17 Route functionality (sairoute.h)

Mandatory. Implements IP routing table (FIB) management functions. When an IP packet is received, the ASIC first determines if the packet is destined a certain router interface on the switch. If yes, then the ASIC lookups its destination IP address in the IP routing table, and forwards the packet to a next hop or a next hop group using ECMP.

18 Neighbor functionality (saineighbor.h)

Mandatory. Implements the IP neighbor table management functions. IP neighbor defines a L3 neighbor that is within the same IP subnet as one of the router interface on the router. The IP neighbor is associated with a specific router interface, is assigned to a specific IP address, and has a mac address. IP neighbor also has a packet action attribute to decide whether to forward, drop or trap the IP packet which comes from a different router interface with destination IP matching the neighbor IP. The IP neighbor does not define the outgoing port as this can be resolved using the FDB table.

19 Next Hop functionality (sainexthop.h)

Mandatory. Implements next hop objects such as IP next hop and IP tunnel next hop. Current version only has IP next hop defined. IP next hop must also be an IP neighbor. It defines an IP neighbor that a packet can be routed to. The user needs to create an IP neighbor before it creates an IP next hop. The sai associates them using the router interface ID and the IP address. The IP next hop is identified by a next hop ID which can be used in sai_route as well as other matching rules to forward traffic to an IP prefix to it. Intuitively, IP neighbor represents a device without any packet forwarding capability so that only traffic destined to its IP are forwarded to it, whereas IP next hop represents a device that can also forward other traffic. IP next hop is associated with a specific router interface, and is identified with an IP address.

20 Next Hop Group functionality (sainexthopgroup.h)

Mandatory. Implements next hop group functions. A router can have many next hops. In case of unicast, a packet can be spreaded among a group of next hops using ECMP. Next hop group contains a group of next hops.

21 ACL functionality (saiacl.h)

Mandatory. Implements generalized Access Control Lists management functions. The ACL contains three types of objects, ACL table, ACL entry and ACL counter. The ACL table contains a number of ACL entries. Each ACL table defines a set of unique matching fields for all its ACL entries. A packet can match rules in different ACL tables and take non-conflicting actions from all the matched rules. However, within an ACL table, if a packet matches multiple rules, only the actions from the rule of highest priority are executed. ACL counters can also be created and attached to an ACL entry in order to counter the number of packets or bytes that match the ACL entry.

22 Quality of Service functionality (saiqos.h)

Optional. Implements QoS functions. Manages the port scheduling mechanisms and CoS mapping.

23 History

Version	Description
0.14	Added description for startup/shutdown
0.14	Added warm restart description
0.14	Added support for “switch profile”
0.14	Use bool instead of bool_t_
0.14	Use char* instead of wchar_t*
0.14	Lowercase types; defined portable sockaddr_inet and ip_address_prefix
0.14	*nix includes
0.14	Handling multiple ports when adding/removing from vlans
0.14	Replaced #pragma once with guards
0.14	Changed sai_port_t to be uint32_t
0.14	Added deletion of FDB entries by vlan, by port_vlan
0.14	FDB callbacks merged into one. “Flushed” event added. Added reporting of multiple entries
0.14	Added FDB entry packet actions
0.14	Added bulk deletion of neighbors
0.14	Added ECMP attributes to switch
0.14	Added separate router admin state for V4 and v6
0.9.1	Added intended audience, high-level system architecture. Removed Adapter warm restart considerations.