SINE: Smart InterNet Evolution

Project Report – Fall 2012

Yan Zou (yz2437@columbia.edu)
Sabari Murugesan (sm3478@columbia.edu)
Kshitij Raj Dogra (kd2413@columbia.edu))
Abhijeet Tirthgirikar (apt2120@columbia.edu)

CONTENT

Smart Internet Evolution

The Smart Internet Evolution (SINE) [1] architecture proposes a unified network architecture, which makes better use of heterogeneous dynamic networks and provides transparent control of network access. The security functionality of the SINE architecture must support authentication, integrity and confidentiality across all layers, and at the same time align with the design objectives to maintain mobile/wireless communication performance and usability.

SINE architecture proposes a network control function (NCF), which decouples the transport and network layers. SINE implements NCF using a dual stack configuration of Host Identity Protocol (HIP) and Mobile IPv6 (MIPv6). Similarly, SINE proposes a link control function (LCF), which decouples the network and link layer. SINE implements LCF using IEEE 802.21 Media Independent Handoff (MIH) framework. The architecture supports heterogeneous network technologies, which necessitate distinct security schemes for different network access technologies. The mobile node (MN) needs to maintain multiple credentials for network access (i.e., access provider) and service access (e.g., application/content provider). Therefore, credential management in a heterogeneous network environment is an essential part of the security architecture. The SINE network architecture is decoupled into the infrastructure and service planes. The infrastructure plane provides *data* network access and the service plane *controls* the network access by managing identities across infrastructure providers using federated identity management.

The Smart Internet Evolution (SINE) architecture proposes a new mobility approach with an enhanced networking stack that supports multi-homing, mobility, multipath and disruption tolerance as core functionalities [1]. The architecture proposes a control middleware paradigm based on a policy-based modular decision making system. SINE is designed to support heterogeneous networks, either in terms of network technology (LTE, Wi-Fi, WiMAX) and service providers.

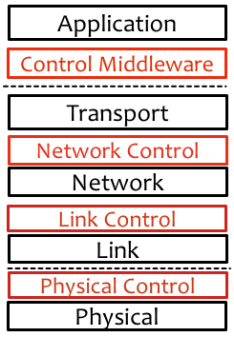


Fig 1: Mobility Enhanced Internet Protocol Suite

The SINE architecture introduces an enhanced 5-layer Internet protocol suite, figure 1, including network, link and physical control functions, and networking APIs, which are backward compatible with BSD socket APIs. For NCF, SINE proposes a dual stack configuration of HIP and

Mobile IPv6 protocol. This provides backward compatibility with IPv6 and allows HIP to be deployed incrementally. For LCF, SINE uses IEEE 802.21 MIH framework [2] for handover optimizations.

SINE divides the physical network into two planes, service and infrastructure, figure 2. The service plane is focused on network services and applications and is responsible for performing security management of the MNs controlling the access to different administrative domains and/or heterogeneous technologies using a federated identity management framework. The infrastructure plane focuses on the physical network infrastructure and the service plane controls the network access of MNs. With this approach, the service providers are decoupled from the physical infrastructure. The decoupling allows service providers to offer services across multiple infrastructure providers resulting in a scalable network service. Also, multiple service providers can share the same physical infrastructure, for example, Sprint physical infrastructure being used by Virgin and Boost Mobile service providers.

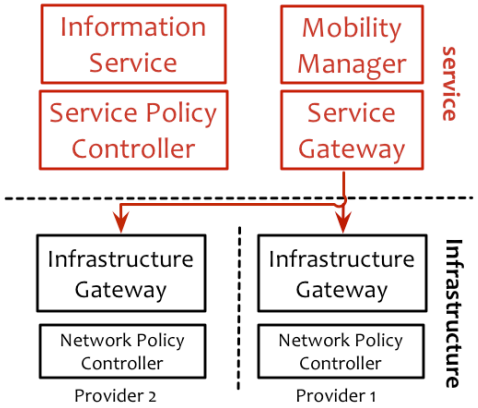


Fig 2: Infrastructure and Service Plane

Control Middleware

The control middleware abstracts the User Applications from the underlying network system. While SINE's network system offers features such as Multihoming, Mobility and Disruption Tolerance using an evolutionary stack of networking protocols, SINE's middleware functions as a medium that orchestrates it's interactions with the host and end-users. In the Internet world, the middleware would be analogous to the View-Controller of the MVC architecture associated with the Web design. This design offers us the flexibility to enhance our middleware interface independent of the network protocols developed for SINE. The middleware has two interface points for capturing the Application – Middleware interaction and the Middleware – Network System interaction.

Apart from providing interfaces, the middleware also makes core routing decisions that are based on pre-defined rules (see Policy Engine) and run-time conditions (see SMART Routing). The placement of middleware components plays an important role in the SINE platform. The middleware needs to be closer to the user-space to interact with user applications and also closer to the network system inside the kernel for seamless integration. The former is accomplished by providing wrapper libraries and services in the user-space and the latter by implementing the Policy Engine inside the kernel. Policy Engine can be implemented as a loadable kernel module and the interactions can be performed similar to that of a device driver. Policy Engine integrates with the Network manager and captures the Middleware – Network System interface discussed earlier.

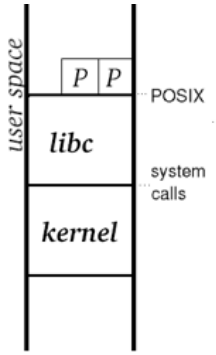


Fig 3: User Space/Kernel Space in an Operating System

As we already know, SINE supports native as well as legacy applications. The native applications can make use of the API's in the libraries to interact with the SINE platform. For legacy systems, SINE provides SOCKS support in the form of a service running on the platform. The SOCKS service is a separate process on the Operating System that is run as the root user. The choice of SOCKS stems from the fact that most applications like web browsers & Networking applications (Drop Box, VLC Media Player etc.) have in-built support for SOCKS and can delegate socket operations. Our SOCKS service is built on top of a SOCKS server that is natively compiled over SINE platform.

2.1 Networking API

At the highest layer inside the middleware is its Networking API, similar to the BSD Sockets API in terms of design, but with additional socket options that drives the SINE system functionalities. Applications can be built with this set of new Networking API's to utilize the services effectively. It is worth emphasizing that it is not mandatory for Applications to switch to our API's for using our features. An application can be agnostic of the middleware and still utilize most of its features. The use of API's only enriches an Applications functionality to better interact with the Network system.

Most applications today on Linux variants use the Berkeley sockets to interact with the Operating System's network sub-system. We have designed our library of socket API's as an abstraction to BSD sockets and hence most of today's applications can be easily rebuilt in order to use SINE functionalities.

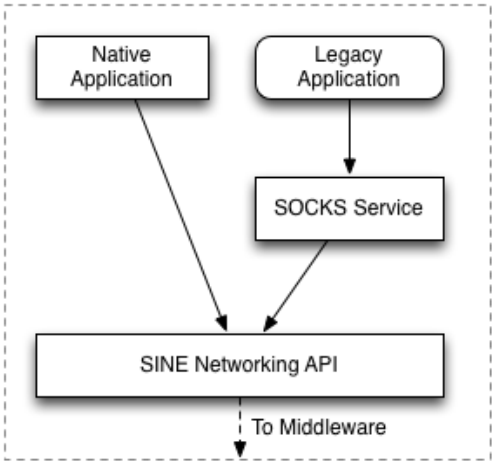


Fig 4: SINE Networking API & Service Support

2.1.2 Legacy Applications

As we had already mentioned, an Application can remain agnostic of the middleware because SINE can also function as a SOCKS server. The applications can then easily delegate the socket operations to SINE middleware via the SOCKS protocol. Thus, the bare minimum requirement for an application to use our services is to support SOCKS. We also have a solution for applications that do not support SOCKS, but that requires the application to be rebuilt. We call this packaging as Sinefy (similar to Socksify), wherein we dynamically link the socket calls to sine socket API calls. You can read more in the packaging section (See Packaging and Installation) of SINE middleware.

2.3 Connection Management

Regardless of the type of Application – Middleware interface, the remaining aspects of SINE platform behave in the same manner. The SINE libraries functions as an intermediary before forwarding the socket calls from applications to the Operating system's network subsystem. Every socket operating updates the state of an entity known as the Connection Manager that is within the SINE middleware. The main function of the Connection Manager is to hold the state of each socket that is opened by an application. The connection manager exposes a querying interface that is utilized by the Policy Engine & Network Manager to obtain the set of sockets to work with. Hence, the connection manager can be envisioned as the repository for SINE platform holding the most up-to-date socket related information.

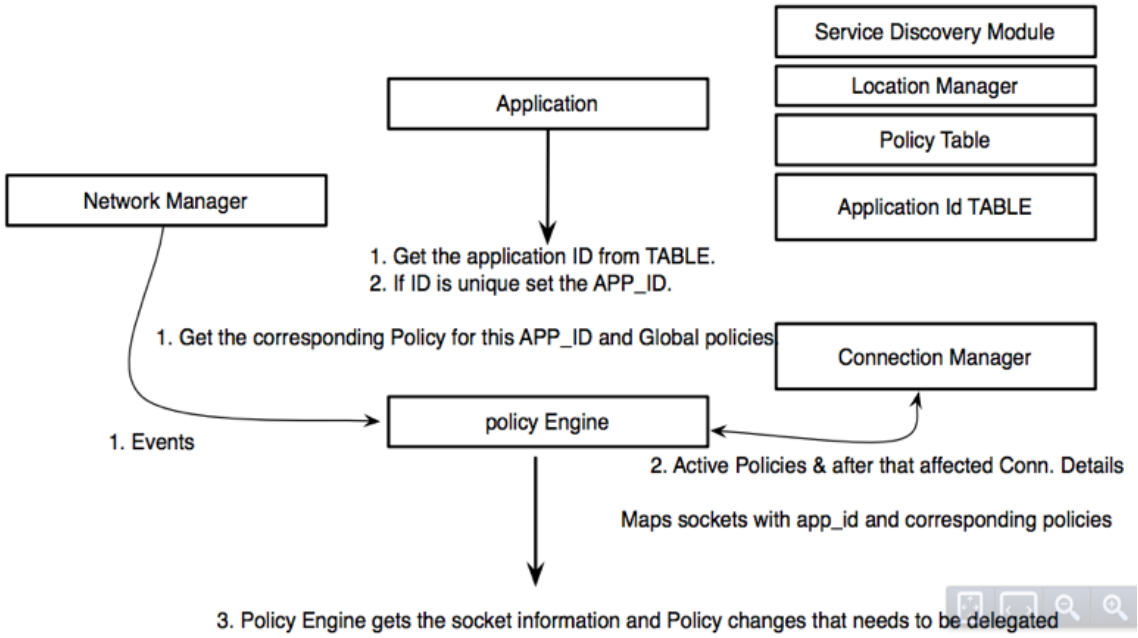


Fig 5: Connection Manager – Policy Engine – Network Manager Interactions

This mandates a crucial requirement and guideline when integrating applications with SINE platform. Every socket operation has to be done through the networking libraries provided by SINE platform. Any deviation from this would result in corrupting the Connection table and may cause problems in the working on Policy Engine & Network Manager. Application developers need to strictly discard references to socket.h and use sine_socket.h library headers in their applications.

2.3.1 Connection Manager

Currently, the connection manager uses a doubly linked list data structure to represent socket information. The sockets are not arranged per application basis. Each application has a unique ID within the scope of a host that is passed on from the Interface layer.

SOCKFD	AF_TYPE	PROTO_TYPE	PORT	APP_GUID	POLICY_GUID
512	AF_INET	SOCK_STREAM	9001	121001	1001
980	AF_INET	SOCK_DGRAM	5000	121002	1004

Fig 6: Connection Manager Sample Record Format

Whenever a new socket is opened by the application, a new node is created in the linked list and the application ID & the state of the socket is stored inside it along with socket File descriptor. Every time a socket's status is changed, we retrieve the node corresponding to the socket file descriptor and update its status. When a socket is closed, we delete the corresponding node in the list.

Concurrency is a very crucial aspect and can be implemented using Reader-Writer locks. This is because a query operation would result in a wrong state if the list is allowed to be modified at the same time. Our implementation lacks the concurrency aspect at the moment. A better data structure could be using a Hash table with the Application ID as the key and value being a linked list of socket information. This optimization would result in a faster access to retrieve the list of sockets corresponding application, one of the most used query criteria.

The interaction between Connection Manager & Policy Engine and between Connection Manager & Network Manager is covered in their respective section.

2.4 Policy Engine

Policy-based Network Management (PBNM) is a management paradigm that separates administration operations from other basic network operations. It provides a flexible and robust mechanism to manage network resources and services e.g. bandwidth allocation, quality of service, access rights, traffic prioritization and security to different network elements [1]. PBNM defines policy's architecture as consisting of four components namely, condition, action, role and priority. We can say that a policy is a combination of a set of conditions with an associated action to reach a goal.

2.4.1 Policy Language

A simple policy in our implementation consists of the following components

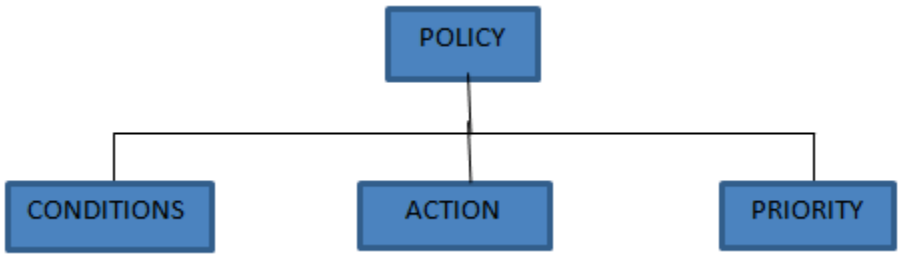


Fig 7: Policy Structure

Action – An action in our implementation is the choice of the interface. While describing a policy, a user specifies which interface to choose when certain sets of conditions are satisfied. These actions are determined at the Policy Decision Point. The decision point states the interface which the HIP or MIPv6 should bind to. The module called Network Manager performs this procedure. The active socket calls for this application use only that interface for any transmission.

Conditions – It specifies the criteria based upon which decisions are made by the PDP. Also whenever there is a change in any of their values, an event is invoked to the Policy Listener. We have identified following 5 parameters, which are used while specifying any condition

Interface – This specifies the type of network connection the application is using and its status. For example –
Ethernet – eth0
WLAN – wlan0
LTE – ppp0
While evaluating a policy, the PDP contacts Network Manager to find whether the required interface is up or down.

Location – As mentioned before, location is the geological coordinates of the user. While taking decisions based upon this metric, the system information is used as well. For a user who has moved to a new location as per the GPS, but notifies it as the same type of network as it had before should not be considered as an event.

Cost – Cost is an attribute of the interface. The Network Manager makes a note of the current cost of each available interface. They are stored in the form of 'Kilobytes/$'.

Bandwidth - Bandwidth is an attribute of the interface. The Network Manager makes a note of the current bandwidth of each available interface. They are stored in the form of 'Megabits per second' (Mbps). When there is a change in this metric for an active interface, an event is triggered to the Policy Listener.

Maximum Download Limit – Flow Manager tracks the number of bytes downloaded for each application. When the specified limit is reached an event is triggered to the Policy Listener.

Our policies prioritize a set of rules based upon the order in which they are defined. Rules that are higher up are given more preference as compared to the ones below. In other words, all the rules are ORed, and the first one that evaluates true is chosen.

Also we have chosen the implementation of Policy Framework Definition Language (PFDL). It is a tool language to describe various kinds of user and application policies. The PFDL can simply express lists of IF <condition> THEN <action> type of rules. Any such list forms a policy <condition> above is in fact a disjunctive normal form of single condition expressions, and <action> is a list of single action statements. The semantics of such rules is clear: if the

evaluation of the condition expression with the current state of the policy server (policy decision point) and the specific client (policy enforcement point) request succeeds, the action list can be performed [3]. The language for our model is the following

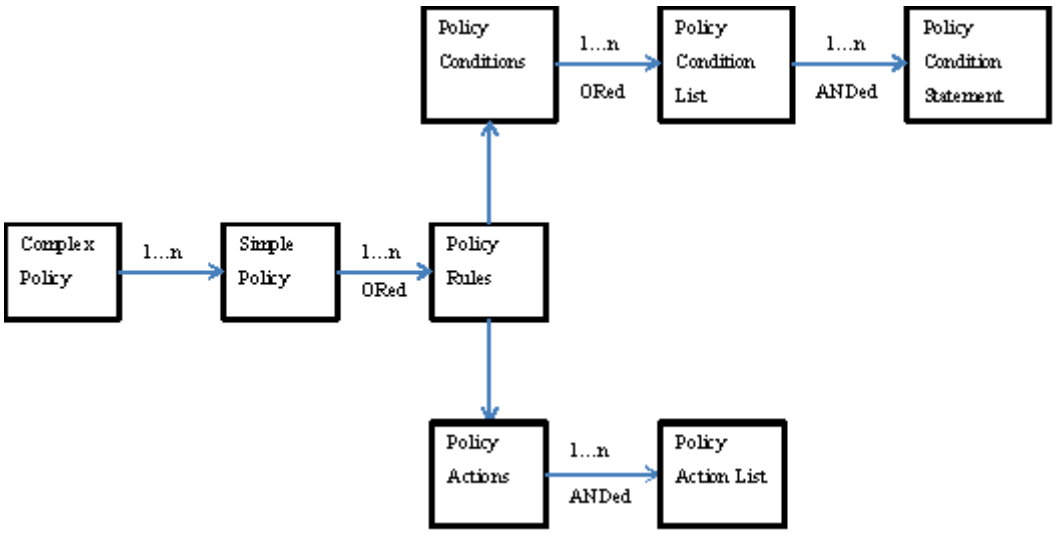


Fig 8: PFDL implementation

Now we demonstrate a sample policy, which is a part of our implementation and follows this scheme –

Application – myNetflix

1. Use interface 'WLAN'
    a. When location is 'Office' and cost <= 40 and bandwidth <= 200 and download limit <=200
    b. When location is 'Home' and cost <= 20 and bandwidth <= 100 and download limit <=250
2. When location is 'Columbia'
    a. Use LTE, if cost <=10 and bandwidth <= 256 and download limit<= 100

(Cost and bandwidth are 'Kilobytes/$' and Mbps respectively.)

This policy states that – If interface 'WLAN' is up and the any of the two conditions are satisfied (in priority) then network manager should use WLAN as an interface to bind to. If none of the two conditions are satisfied then the second rule is evaluated which says, if the location is 'Columbia' and the other conditions are satisfied, then use LTE. If none of the conditions are satisfied, then we check the remaining two rules.

This is stored in the form of a configuration file in the Policy Repository as follows:-

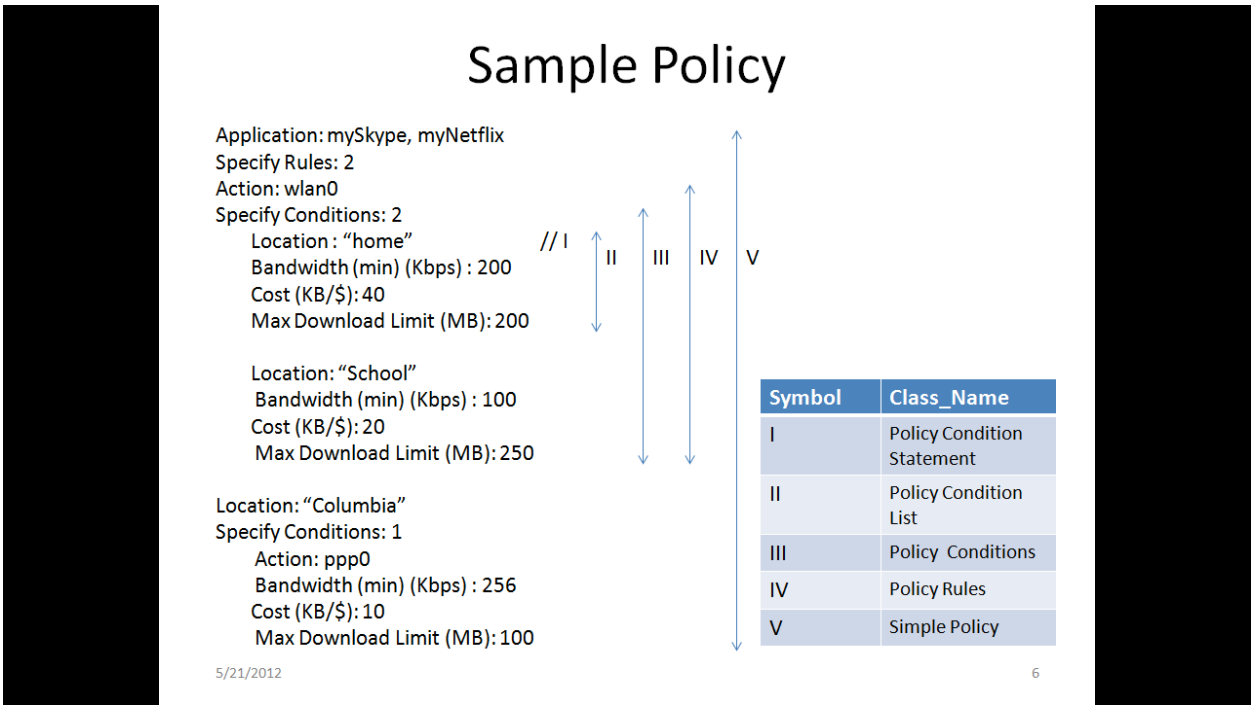


Fig 9 Sample Policy Configuration

2.4.2 Implementation

We have implemented an 'Event Condition Action Policy' Model based on user defined policies. Policy-Based Management in SINE is about implementation of a set of rules which specify the allocation of network resources on a per user or application basis to best support the established business objectives. In future we will scale the policy definitions and also include policies defined by the network. These policies will be exclusive and would state rules like – use LTE if throughput or latency falls below a certain level.

2.4.3 Architecture

We have modified the PBNM architecture to the following –

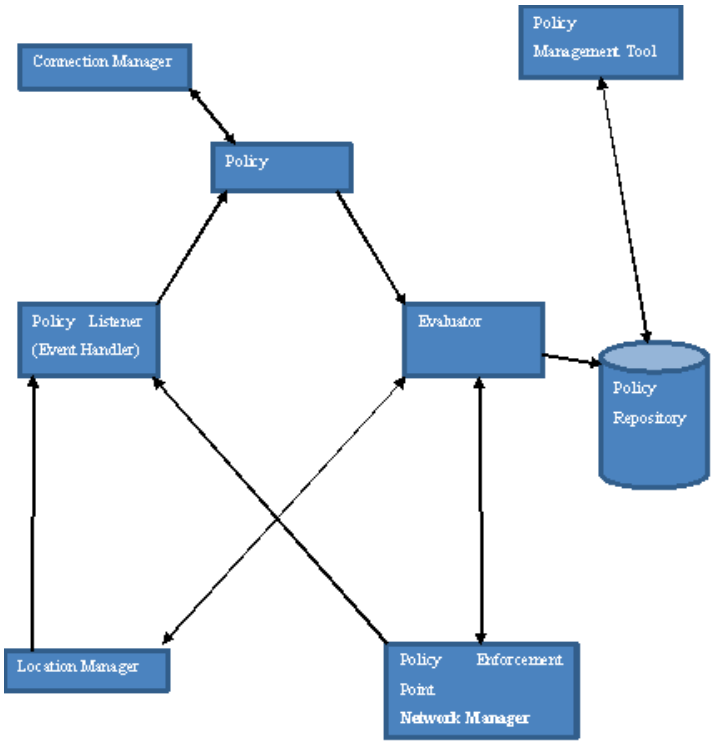


Fig 10: Policy Architecture

This architecture consists of a policy management tool, which is responsible for creating, editing and storing the user policies. We have a 'Policy Repository', which stores them as configuration files. Also the Policy Decision Point (PDP) called 'Evaluator' is where the evaluation for a specific policy is performed and a desired interface to be used is determined. These decisions are then sent to the Network Manager, which is the Policy Enforcement Point (PEP). As mentioned later, our policy has several metrics like location of the mobile host, available cost and bandwidth of the interface etc. While evaluating the policy at PDP, it contacts the Location Manager for the current location and the Network Manager for the cost and bandwidth for each interface.

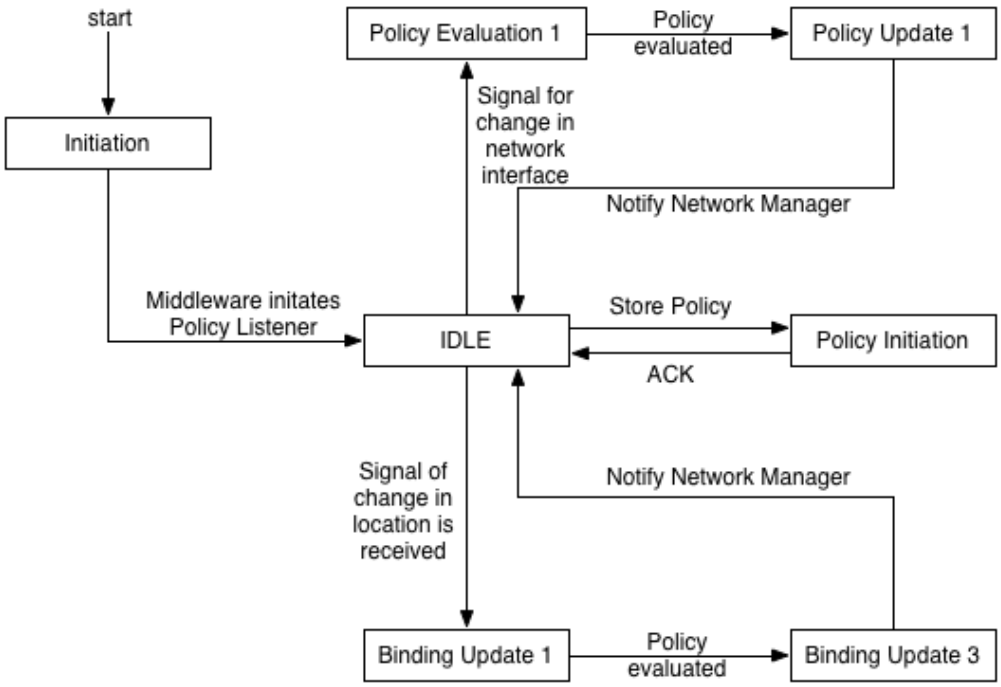


Fig 11: Policy Engine State Machine

Also we have two components as a part of our Policy Management called the Policy Listener and the Policy Controller. The Policy Controller interacts with the Connection Manager for the Application ID and the socket file descriptors participating in the network connection. On the other hand the Policy Listener listens to any events from the Network Manager and the Location Manager. On receiving an event, the listener calls the controller, which in turn invokes the PDP along with the Application ID and File Descriptor.

2.4.4 Policy Generation

The policy generation consists of two phases.

The system user registers his current location in terms of the geological coordinates and the network identifier with the policy engine. The latter is done in the following ways -

1. WLAN : SSID
   32-byte (maximum) character strings
2. LAN :
   Use Geo-Location
3. LTE :
   International Mobile Subscriber Identity or IMSI - unique number associated with mobile phone users. It is 15 digits long and stored in the SIM inside the phone and is sent by the phone to the network.

Together they identify a network associated with a user. For instance a user may have more than one type of home networks. In addition to the user registering his network, the Policy Engine has a set of known networks already registered with it, like the 'Boingo' network. The pre-registration ensures that a user's movement to a public network, which is already recognized, should not be treated as a network change. The user makes policies in earlier defined manner.

2.5 Network Manager

As stated in paper [1], the Network Manager (NM) maintains and monitors the information about all active network interfaces. NM provides four kinds of services in the middleware: (1) current interfaces state, (2) nearby active networks, (3) network information using MIH Information Service, and (4) Alert subscription whenever a particular network is nearby. For the current implementation, we only use MIH protocol to provide link information, thus only (1) and (3) are considered in this article.

The article is organized as follows: Section 2 discusses the overall model design of Network Manager, including the protocols used and the architecture; Section 3 illustrates in detail how each component is implemented; Section 4 lists all the API's that Network Manager provides for Policy Engine and discusses how the components communicate and work with each other; Section 5 concludes.

2.5.1 Design

The Network Manager is currently designed to provide network information to Policy Engine and send commands to lower layers to do handover.

2.5.2 Protocols

NM directly talks to three different protocols: (1) IEEE 802.21, Media Independent Handover (MIH), which triggers events like link up/down and also provides all available link status and network information; (2) Host Identity Protocol (HIP), which provides only one particular identity for this host independent of wherever it is and whichever interface it uses; (3) Mobile IPv6 Protocol (MIPv6), which creates a tunnel with Home Agent to ensure a fixed Home Address for this host, no matter what network the host is currently in.

2.5.3 Architecture

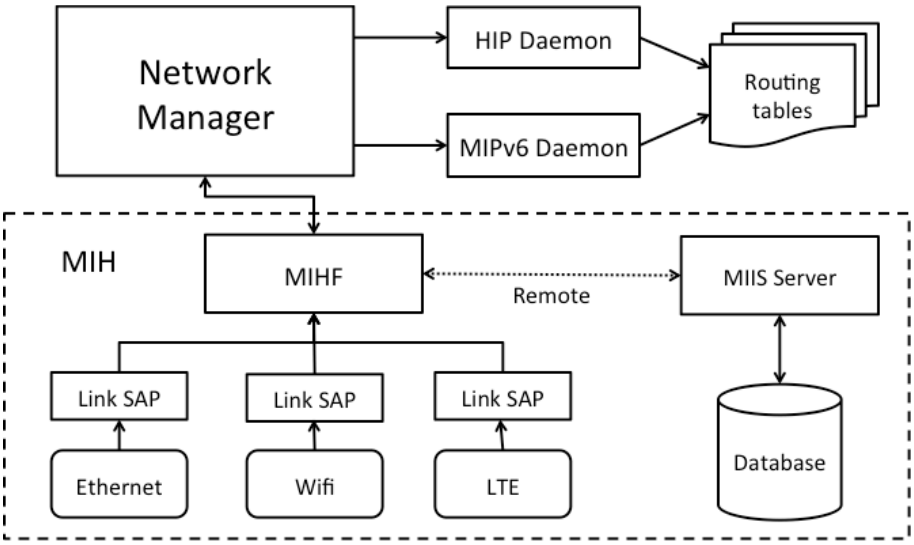


Fig 12: Network Manager Architecture and Function Flow

As we can see from the Figure 2-1, NM directly talks to MIHF (Media Independent Handover Function, core component of MIH protocol), HIP Daemon and MIPv6 Daemon.

When there is a link event happens, such as wireless card is turned on, Link SAP (Service Access Point) will get the event and notify MIHF, which will forward the event to NM. NM will further pass the event on to Policy Engine (PE) to make new decisions on which interface to use for each application. In the process of making decisions, PE may request more network information like cost and bandwidth. It sends information query command to NM, and NM passes the command to MIHF. MIHF communicates with remote MIIS (Media Independent Information Service) Server to obtain more information of certain networks, and then sends the information back to NM and on to PE. After PE has decided which protocol and interface to use, it sends command through NM to HIP daemon or MIPv6 daemon to do the local handover. HIP and MIPv6 will do certain actions (like sending Binding Update) according to the handover command.

2.6 Security Manager

The security manager consists of authentication and authorization module. Security manager authorizes the mobile node to connect to a foreign network interface or perform a handover from one network to other network using single sign-on system. There are several ways to provide the single sign-on functionality; SAML, OAuth, OpenID to name a few.

SAML (Security assertion Markup Language) is an XML based open standard for exchanging authentication and authorization information between security domains. It was developed by OASIS. The identity provider passes a SAML assertion to the service provider, which contains *statements* that service providers use to make access-control decisions. There are a

couple of implementations based on SAML architecture widely in use today - Shibboleth and Eduroam.

The security manager in SINE project implements this using Shibboleth authentication which uses federated Single Sign-on. Both the techniques are discussed as below –

Shibboleth: The Shibboleth is standard based open source software for web single sign-on across or within organizational boundaries. Shibboleth is based on federated identity standard called Security Assertion Markup Language (SAML), which provides a federated single sign-on and attribute exchange framework. Shibboleth also provides extended privacy functionality allowing the browser user and their home site to control the attributes released to each application. Using Shibboleth-enabled access simplifies management of identity and permissions for organizations supporting users and applications.
Shibboleth has two major halves: an Identity Provider (IdP), and a Service Provider (SP). The Identity Provider supplies information about users to services, and the Service Provider gathers information about users to protect resources. In the typical use case, a user authenticates with his or her organizational credentials. The organization (or identity provider) passes the minimal identity information necessary to the service manager to enable an authorization decision and user ends up at the protected resource with a successful session.

Eduroam: Eduroam (education roaming) is a secure, worldwide roaming access service developed for the international research and education community. It allows students, researchers and staff from participating institutions to obtain Internet connectivity across campus and when visiting other participating institutions by simply opening their laptop. The eduroam principle is based on the fact that the user's home institution does the user's authentication, whereas the visited network does the authorization decision allowing access to the network resources.

When a user tries to log on to the wireless network of a visited eduroam-enabled institution, the user's authentication request is sent to the user's home institution. This is done via a hierarchical system of RADIUS servers. The user's home institution verifies the user's credentials and sends to the visited institution (via the RADIUS servers) the result of such verification.

Apart from these two software solutions we now also have Hotspot 2.0 coming up as a commercial solution for ease of roaming between Wi-Fi networks and cellular carriers. It provides a standard automated mechanism for signing on and eliminates manually selecting a network or entering a password.

Shibboleth and Eduroam are similar technologies that provide solutions to two different objectives. Eduroam provides the network access technology to make it easier for users with valid accounts to log on to networks, both at home and when visiting participating organization networks. A user in this case may have no access to the Internet. Shibboleth additionally provides access to the protected online resources. It is a streamlined way of exchanging

information between an individual and providers of digital data resources to authorize the user's access to the resources. It has been designed to protect both the security of access to the data and the privacy of the individual viewing it since authentication and authorization is controlled by the home organization.

Federated Identity Management – Shibboleth

As mentioned previously the shibboleth infrastructure has two major components: Identity Provider and Service Provider. There is also a third component called Discovery service which can be used to provide identity provider selection. The discovery service is analogous to "*where are you from"* service.

Identity Provider

The Identity Provider consists of a front facing Apache webserver, which houses the Single sign-on authentication system. The webserver is connected to authentication DB to verify user credentials. The webserver relays the connections to Apache tomcat server, which has a handle server, and an attribute authority that queries the user attributes in the user DB. The flows through these blocks are explained in the complete working. The Identity Provider consists of a front facing Apache webserver, which houses the Single sign-on authentication system. The webserver is connected to authentication DB to verify user credentials. The webserver relays the connections to Apache tomcat server that has a handle server and an attribute authority that queries the user attributes in the user DB. The flows through these blocks are explained in the complete working.

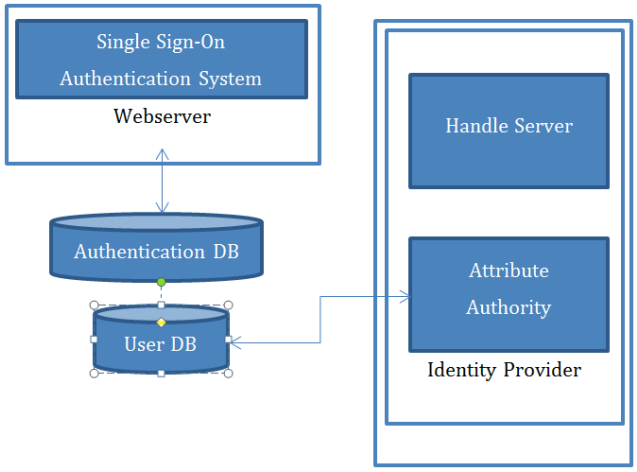


Fig 13 Identity Provider

Service Provider

The service provider entity consists of Apache webserver loaded with a shib module. The shib module has an attribute extractor and an attribute filter which provides the required access control for the web resources. The web server defines the protected web resources. It also consists of a shibboleth Daemon which does the message processing.

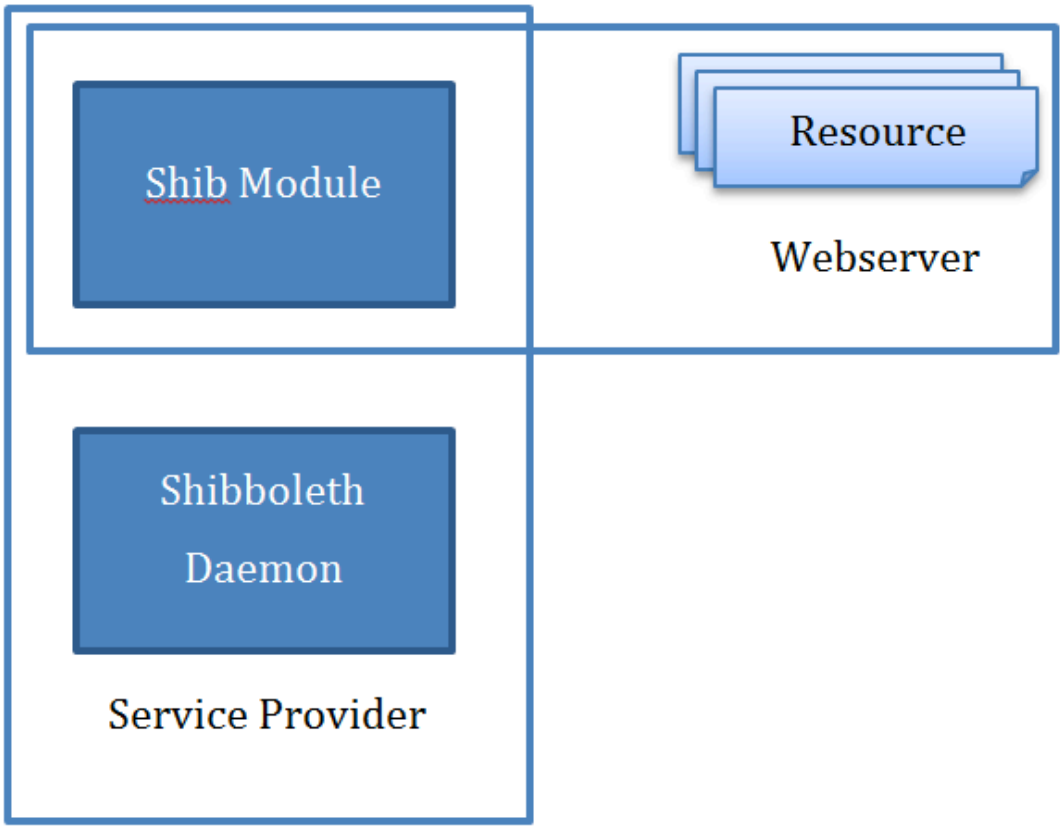
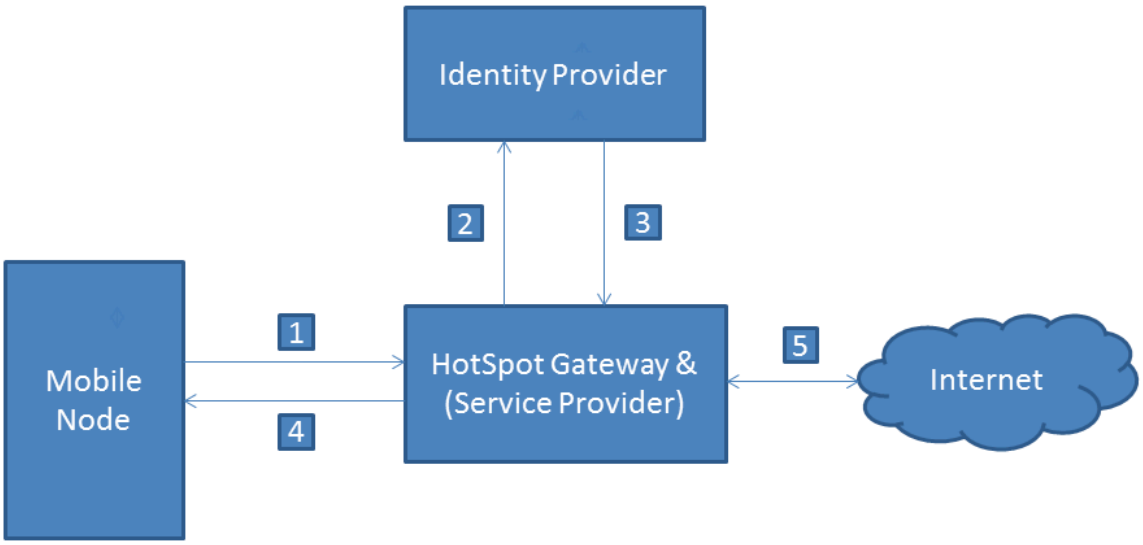


Fig 14 Service Provider

Complete Security Setup

As part of the security demo, we have enabled authorization on the Wi-Fi and LTE interfaces. When a policy decision is made to switch to a new interface (Wi-Fi or LTE) the policy manager invokes the security manager for authentication. The security manager uses the shibboleth module for authentication.



Security setup with Shibboleth

Fig 15 Security Setup with Shibboleth

Functional Flow:

Following are the steps that take place during the complete process-

1. The security manager (MN) opens a browser and accesses the URL of the service provider (SP) of the Wi-Fi interface. (https://sp.irtlab.org/). This sends a HTTP GET request to the SP machine.

2. Since, this user is not yet an authenticated Shibboleth user, the web server at the service provider redirects the web-browser to the webpage of the Identity Provider of the user (or home organization) single sign-on authentication system. The authentication request is passed through the browser, and the client is redirected (via GET or POST) to an endpoint at the IdP.

3. The IdP examines the request and decides how it would like to authenticate the user based on rules established for the SP in relying-party.xml. Here we have configured our setup for Remote User Authentication out of the four different types of authentication that can be used https://wiki.shibboleth.net/confluence/display/SHIB2/IdPUserAuthn.

4. The user is redirected to the selected login handler, authenticates (or tries to) using the method selected and eventually control passes back to the profile handler with their username set.

5. The IdP now uses the user's name, the SP, and the protocol and binding/profile selected to decide what information to send the SP. It gathers a set of attributes for the user using the attribute resolver after which the attribute filter pares down the information to be included in the response. The resulting information could be as little as "someone authenticated successfully".

6. This information is packaged as SAML assertion. This assertion is signed with the IdP's key and encrypted using SP's key. This is placed into the response and passed through the browser to be sent to the SP.

7. This response is received by the Assertion Consumer Service at the SP. The ACS decodes and decrypts this assertion after which the SP creates a new user session and translates the user attributes into a cacheable form.

8. Finally, the browser is redirected to the URL in step 1.

9. If a successful authentication is received and a new user session is created, a log process running at the Service Provider communicates to the policy engine a successful authentication.

10. Following this response from the SP log process, the policy engine then sets the network interface for use.

Security Manager Finite State Diagram

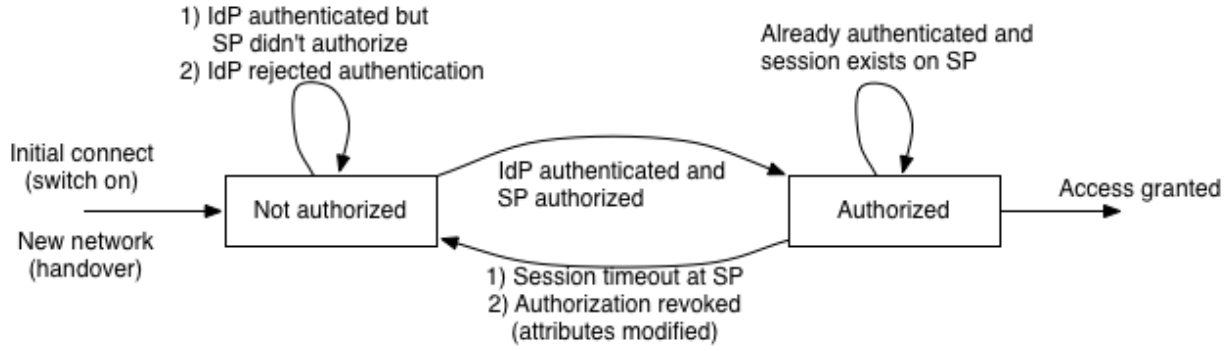


Fig Security Manager Finite Diagram

Security Manager API - bool sine_shib_browser(char* ifname)
Initiates the authentication process using the shib module for the ifname (interface name) passed as parameter and returns 1 if authenticated, 0 otherwise.
Parameters: char* ifname (interface name)
Return value: bool (1- authenticated, 0 – not authenticated)

Network Control and Link Control

As the shim layer of Network Layer, we have a set of network control functions to manage IP addresses, network access, routing, and mobility. We have chosen two existing protocols - HIP and MIPv6, to fully support mobility. The two protocols can work together in one stack, so that access to servers without HIP support can also have mobility through MIPv6.

The detailed implementation of each protocol, how the two protocols can work together to route different packets will be discussed below:

3.1 Network Control

3.1.1 Host Identity Protocol

HIP is short for Host Identity Protocol, and it works as a shim layer of Layer 3. "HIP is based on a Sigma-compliant Diffie-Hellman key exchange, using public key identifiers from a new Host Identity namespace for mutual peer authentication."[2] In other words, it provides each host with a Host Identifier (HI) and uses this HI to represent a host in network communication instead of using IP addresses. The hash of HI is called HIT, which is 128-bits long and can fit into IPv6 address field. HIP can support mobility and multi-homing easily. Each host keeps a binding list of its communication peers, which is a binding between HIT and location information such as IP addresses. There are Binding Initiation and Binding Update processes to maintain the Security Association and the binding list during communication. HIP also provides something like DNS service to resolve identities to location information used for Binding Initiation.
Two open source implementations for HIP are found – OpenHIP[5] and HIPL[6]. We choose OpenHIP because it supports multiple Operating Systems including Windows, Mac OS X, and Linux, and can be installed in either user space or Linux kernel. It is licensed under GNU GPLv2 and is widely used.

We pick the user space implementation considering its great flexibility and convenience to modify the codes and re-build. OpenHIP provides both 32-bit LSI for IPv4 (1.0.0.0/8) and 128-bit HIT for IPv6 (2001:10::/28). "The last 24-bits of LSI are either the lower 24-bits of the HIT or some value specified in the identities file."[5] In order to capture the packets, it requires the user to use LSI instead of normal IPv4 address. An entry is set into routing table that forwards all the packets with destination prefix 1.0.0.0/8 to a virtual device, the TAP driver, which will then forward the packets to the HIP daemon process running in user space. Same thing happens for IPv6 with a routing entry that forwards all packets with prefix 2001:10::/28 to the TAP driver.

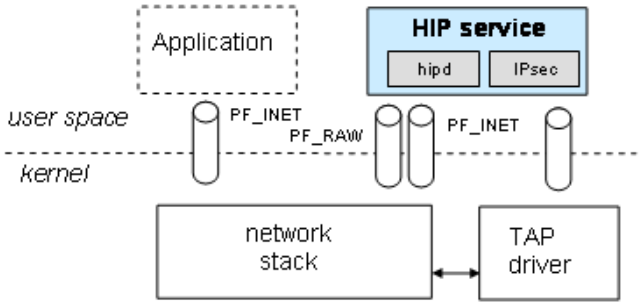


Fig 13: HIP Working Flow

Changes have been made to the OpenHIP codes in order to make it cooperate with Network Manager (NM). A listening thread is added to the HIP daemon process that receives commands from NM and sets preferred interface for each peer.

The HIP working flow is shown in Figure 3-1 on the next page. After daemon starts, it generates HIT (which may have already been generated and stored in some identity file) and sets the routing table entries:

1.0.0.0/8 DEV HIP0
2001:10::/28 DEV HIP0

hip0 is the created virtual device that captures all the HIP packets. Then the daemon goes into an infinite loop waiting for events. Whenever the daemon receives a packet with a new destination LSI that does not have a binding address, it starts binding initiation that gets the peer address and exchanges Security Association. It also sets routing entries into routing table, saying that all packets, whose destination address is that peer address, use the preferred interface. The preferred interface is determined by Policy Engine and notified to HIP daemon by NM as soon as the application tries to connect to peer. Later, when NM sends a command to switch interface for that peer address, HIP daemon will do binding update with that particular peer and change the corresponding routing table entries.

As described in HIP protocol, Binding Initiation contains four packets I1, R1, I2, R2. I1 and I2 are sent from the Initiator to trigger the initiation and exchange D-H keys. R1 and R2 are sent from the Responder to exchange D-H keys. Binding Update process contains three packets, pretty much like TCP three-way handshaking.

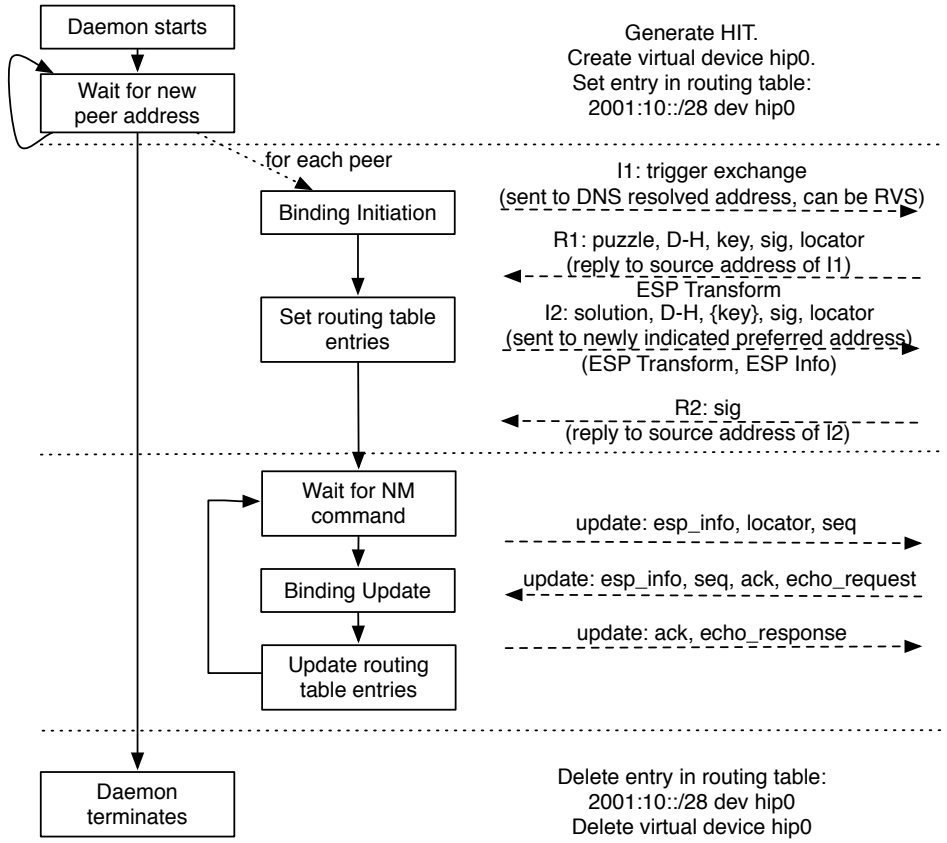


Fig 14: HIP Functional Flow

The following Finite State Machine graph might be better to illustrate both the working flow of Initiator and Responder. From Figure 3-2, we can see that the daemon is mainly waiting for 4 events. Two events that are marked red, new peer address and NM command to switch interface, are events from local machine. The other two, I1 received and Binding Update received, are events from remote hosts that want to initiate a binding to this host or notify the updated address information. It is clear that the NM only controls the preferred network interface that will be used to communicate with each peer. All other things are left to HIP protocol.

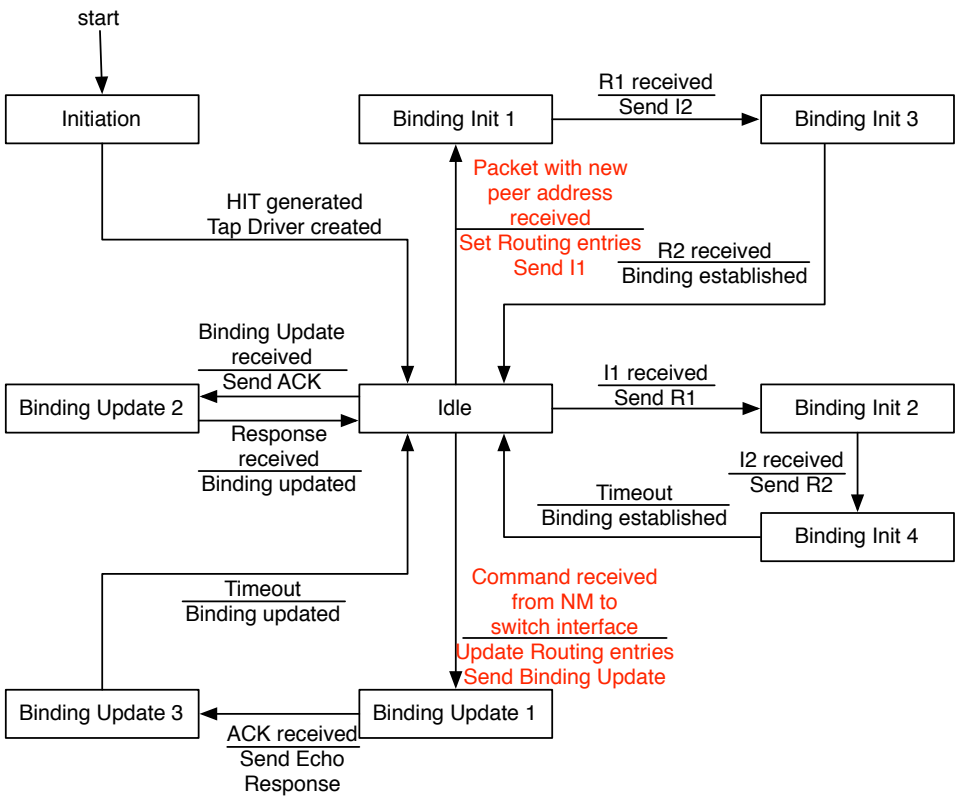


Figure 3-3 HIP Finite State Machine

3.1.2 Mobile IPv6

Mobile IPv6 is a protocol that provides mobility for IPv6. It assumes that every Mobile Node (MN) has a home network, where it uses its permanent Home Address and connects to Home Agent (HA) to get access to Internet. When the MN moves from home network to a foreign network and obtains a new address called Care-of Address (CoA), it registers with HA and creates a tunnel between them, so that all the packets targeted at the Home Address can be intercepted by HA and forwarded to the new CoA through the tunnel. The packets sent from MN can also be tunneled to HA first and the forwarded to Internet. It is like MN is always using the home address. Mobile IPv6 also provides Routing Optimization when the Corresponding Node (CN) supports MIPv6 as well, in which MN directly registers CoA with CN so that packets no longer have to go through HA.

We choose UMIP, the most famous open source MIPv6 Implementation. UMIP contains a MIPv6 daemon. It can detect whether the host is in home network or not. If it is, the daemon will clear all configurations to use normal IPv6 that sends all packets through the local HA inside the home network. But if it is in foreign network or if the CoA is changed, the daemon will send Binding Update to HA in order to notify the newest CoA and create a tunnel between MN and HA to do the forwarding. It also set routing entries to make all packets go through the tunnel interface to HA by default.

Similar to HIP, we modify the codes of UMIP to add a separate thread in MIPv6 daemon process to receive Network Manager (NM) commands and set preferred interfaces. The original functions of home network detection and CoA discovery are disabled. Binding Update will only be sent when Network Manager commands are received. Figure 3-3 shows the working flow:

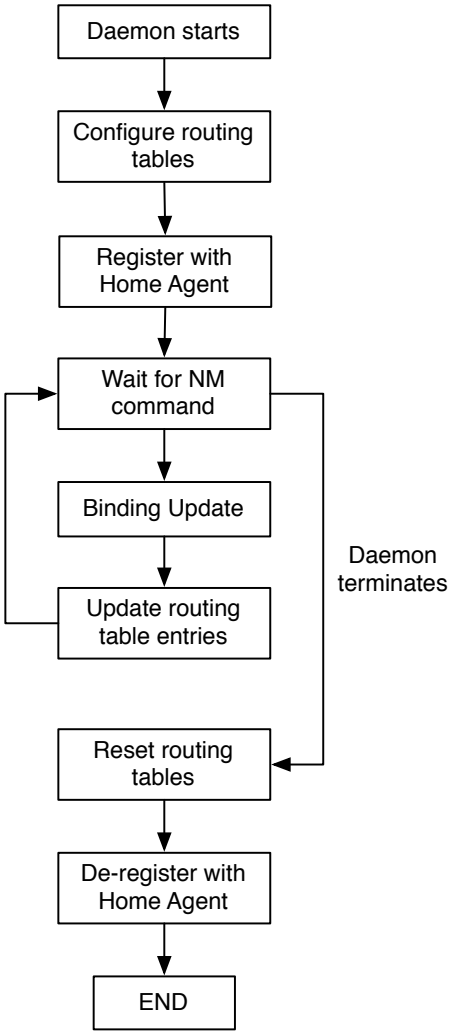


Figure 3-4 MIPv6 Working Flow

In the Finite State Machine of MIPv6, the process of Routing Optimization (RO) that happens on both MN and CN is also included. Apart from RO, we can see that, the Binding Update to register new CoA with HA will only be sent and routing entries will only be updated when commands of switching interface are received from Network Manager.

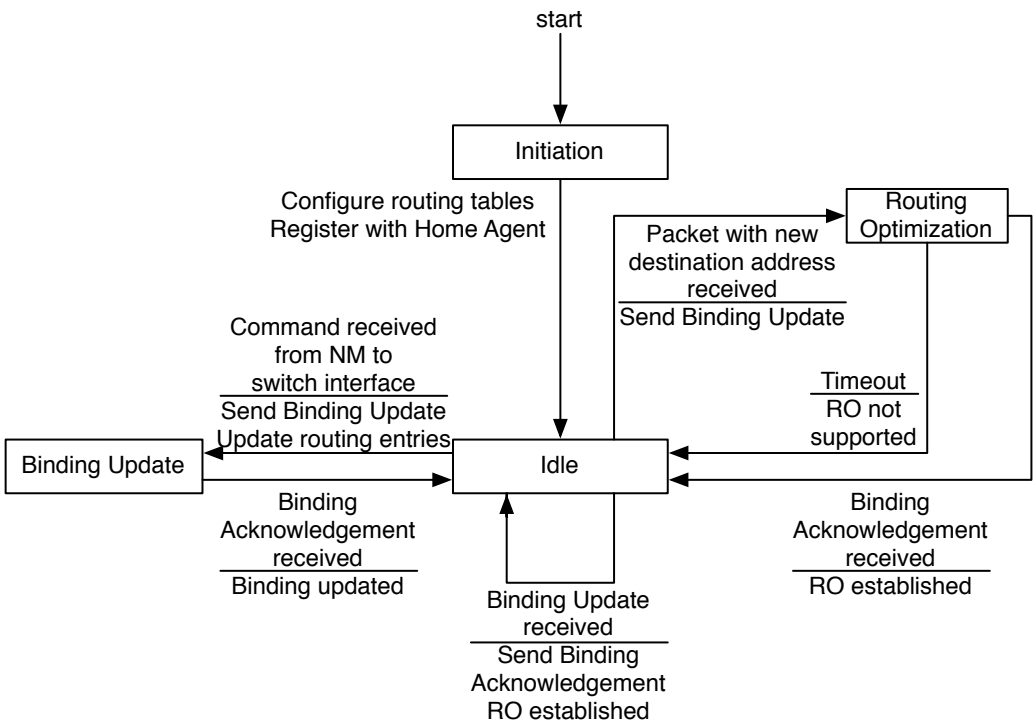


Figure 3-5 MIPv6 Finite State Machine

3.1.3 Interface Oriented Routing Tables

In the design of Policy Manager, each application can set its own preferred interface. Policy Engine is able to choose different interfaces or different networks for different applications. There are cases when one application (App A) uses Wi-Fi, while the other application (App B) is using LTE, and when LTE is down, App B is able to switch to Wi-Fi but App A doesn't need to switch. In order to implement this, it requires the Layer 3 protocol to distinguish packets from different applications and send packets to different networks following the decision of Policy Engine. However, it is hard to do so as Layer 3 can only see the IP addresses in our current TCP/IP stack.

To solve this problem, we can simply differentiate packets according to the destination IP address. Packets with different destination addresses will be sent through different interfaces. This is a very reasonable simplification because of the following two reasons: 1) very few applications share the same destination address; 2) same destination address from different applications means the same routes towards the target hosts, and it is very reasonable for the Policy Engine to choose the same interface for this destination address.
In the actual implementation, we use traditional routing tables to control the packets flow. An entry in the routing tables specifies which interface it will use for that particular destination address prefix. For example, there is a host machine that has Wi-Fi (interface name wlan0) and LTE (interface name ppp0), and there are two applications (App A and App B) running on this

machine. App A talks to 10.1.1.1 and App B talks to 192.168.0.1. When Policy Engine decides that App A should use Wi-Fi and App B should use LTE, we can simply add the following entries to the routing table:

10.1.1.1 DEV WLAN0
192.168.0.1 DEV PPP0

Further, we can optimize this solution by designing the Interface Oriented Routing Tables (IORT). For each interface that the host has, a separate routing table is created to contain all the destination addresses that will use the corresponding interface, as shown in Figure 2-2. In this way, we can simply do local handovers by just moving the address entries from one routing table to another. For example, App A is currently connected to 10.1.1.1 through cable network (eth0), when Policy Engine decides to switch from cable to Wi-Fi for App A, it obtains all the destination addresses of App A from the Connection Manager and sends a command to Network Manager instructing it to move the entry 10.1.1.1 from eth0 table to wlan0 table. Meanwhile, the Network Manager will also notify HIP or MIPv6 protocol to send binding updates based on which protocol the application is using.

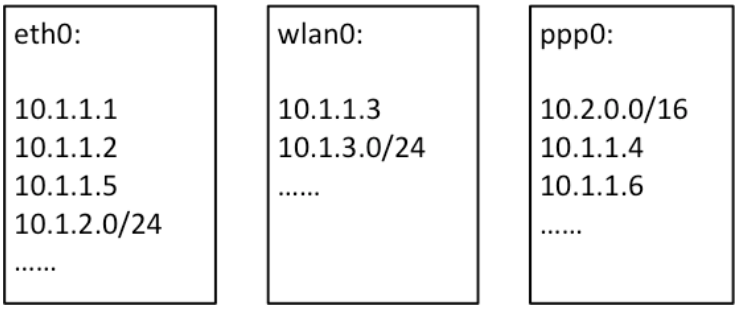


Figure 3-6. Interface Oriented Routing Tables

3.1.4 HIP and MIPv6 Dual Stack

As compared in paper [9], HIP has a better performance than MIPv6 in many aspects. So HIP will always be the first choice. However, HIP requires that both hosts should support HIP protocol, while MIPv6 only requires the Mobile Node to have mobility support. Therefore, we will always use HIP when the destination supports it and turn to MIPv6 otherwise. They will work in a single namespace. The key issue is how to use both HIP and MIPv6 on the same machine for different applications.

The solution is quite straightforward. We differentiate the protocol simply based on the destination address space.

First of all, HIP supports IPv4 but MIPv6 doesn't, thus there will be no conflict in IPv4 stack.
For IPv6, HIP and MIPv6 share the same routing tables, but the packet flows are different. User space implementation captures all the packets with destination address 2001:10/28, which is

the HIT address space [10]. It does the ESP encapsulation for all the captured packets and sends them again. Thus HIP packets will go through the routing tables twice. But MIPv6 daemon does not capture packets, it simply sends Binding Update and, creates tunnel and changes routing table accordingly. All the MIPv6 goes through the routing tables once.

In order to separate the two protocols, we change the routing tables and classify the routing entries into 3 different priorities:

1) Entry

   2001:10::/28 dev hip0

   This has the 1st priority. It guarantees that all HIP packets using HIT address will be captured by HIP daemon.
2) All the routing entries set by HIP daemon and MIPv6 daemon have the 2nd priority. As discussed in 3.1.3, we have Interface Oriented Routing Tables (IORT) here and each table represents an interface. After HIP daemon encapsulates the packets and sends them back to routing tables, they will have the normal IPv6 destination addresses that will not be matched with the HIP entry again. In this case, HIP daemon will set 2nd priority routing entries in IORTs to force these packets to be routed through the preferred interface decided by Policy Engine. These entries have full-length address as destination prefix, which guarantees that the packet will match those entries when checking routing tables. To guarantee all HIP packets are correctly routed, there should an entry for each peer address that specifies a particular interface.
3) Packets still not matching any routing entries must be MIPv6 packets. If the host is in home network, all the left packets can be directly forwarded to HA, which has the 3rd priority.
4) If the host is in foreign network, before we forward the packets to Home Agent through tunnel, we need to decide if the destination also supports mobility or not. If it supports, the MIPv6 daemon will set entries for these optimized routes as 2nd priority, together with the entries set by HIP daemon in IORTs. Note that there will be no conflicts between HIP and MIPv6 in IORT, as HIP and MIPv6 packets always have different destination address. After that, all the left packets will be sent to HA through the tunnel as the 3rd priority.

The logic packet flow is shown in Figure 3-5 below:

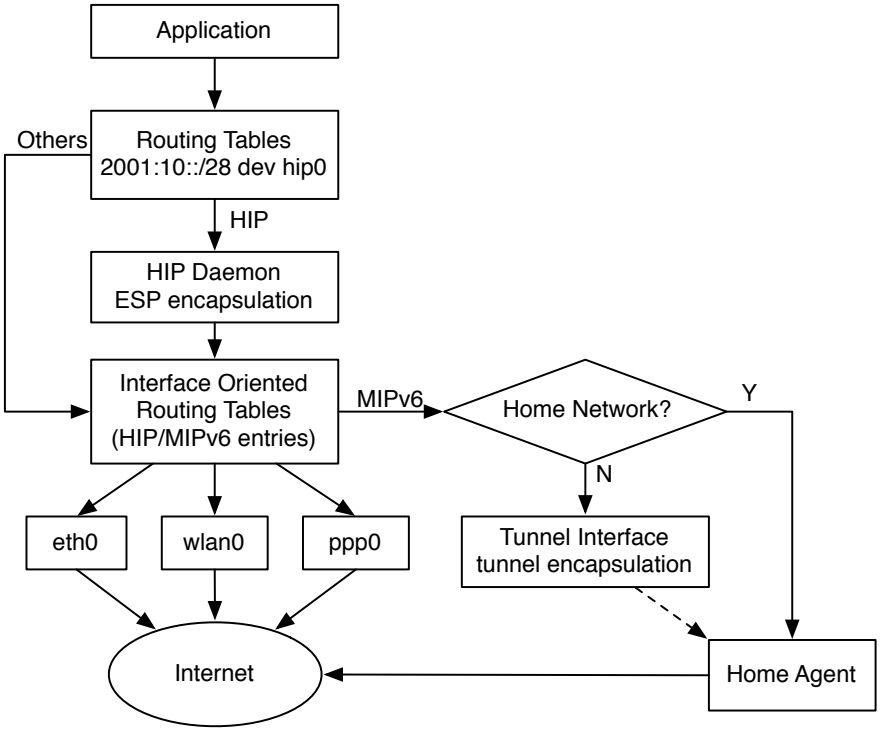


Figure 3-7 Logic Packet Flow

3.2 Link Control

For link control functions, we choose MIH protocol. MIH stands for Media Independent Handover (802.21). It is a protocol that collects information from different kinds of link media and networks and reports to MIH users. It provides three key services, Event Service (MIES), Command Service (MICS) and Information Service (MIIS). We basically leverage the MIES and MIIS to register events and get information for our Policy Engine to make handover decisions.

The protocol is pretty simple. As shown in Figure 3-6, MIH contains 4 main components – MIH Function, MIH User, Link Service Access Point (SAP), and MIIS Server. Each component is a separate process.

1) The core component is called MIH Function (MIHF), which communicates with all other components. It forwards the events from Link SAPs to MIH User, forwards the commands from MIH User to Link SAPs, and queries the MIIS Server for network information such as cost and bandwidth.
2) MIH User is the application that registers with MIHF in order to use the three services that MIH protocol provides. In our design, the MIH User is the Network Manager.
3) Link SAPs have media specific SAPs that can collect events and information from different types of devices, and provides a universal interface to MIHF for events and commands.
4) MIIS Server can be either a local or a remote process (mostly remote), which can query the information database to get network information. MIHF can send request messages

into the network to query all detailed information about that network, and MIIS Server will respond to those requests.

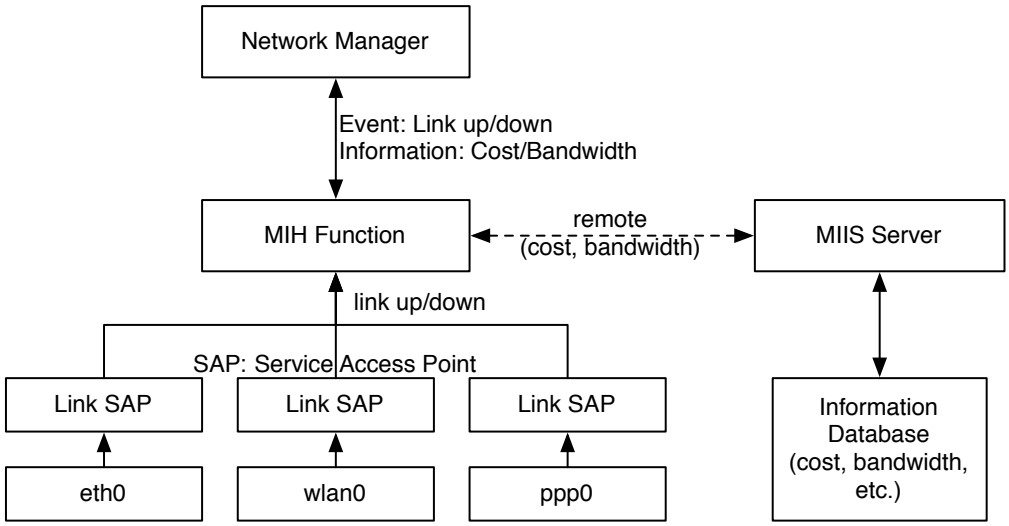


Figure 3-8 MIH Architecture

In our design, Network Manager (NM) is the MIH User, which registers with MIHF. After that, every link up/down event will be forwarded to NM and NM will trigger the Event Listener inside Policy Engine. Also, NM can send Information request to MIHF and get the corresponding response.

Integration and Functional Flow

This section talks about the API that Network Manager provides for Policy Engine. We will also discuss how everything is integrated and work together.

4.1 Network Manager API

Name	Description	Parameters	Return Value
getIfStatus	Query if the interface is up or down	const char* ifname (interface name)	int (1 – up, 0 – down)
getCost	Query the cost of the network that the interface is connected to	const char* ifname (interface name)	int (cost value)
getBandwidth	Query the bandwidth of the network that the interface is connected to	const char* ifname (interface name)	int (bandwidth value)
setHipIf	Set the preferred interface in HIP daemon for that particular socket connection	int socket (socket descriptor) const char* ifname (interface name)	void
setMipIf	Set the preferred interface in MIPv6 daemon	const char* ifname (interface name)	void

Table 4-1 Network Manager API Table

4.2 Functional Flow

When an application is going to establish a new connection to some new peer address using connect socket, the control flow inside the SINE middleware is shown as follows:

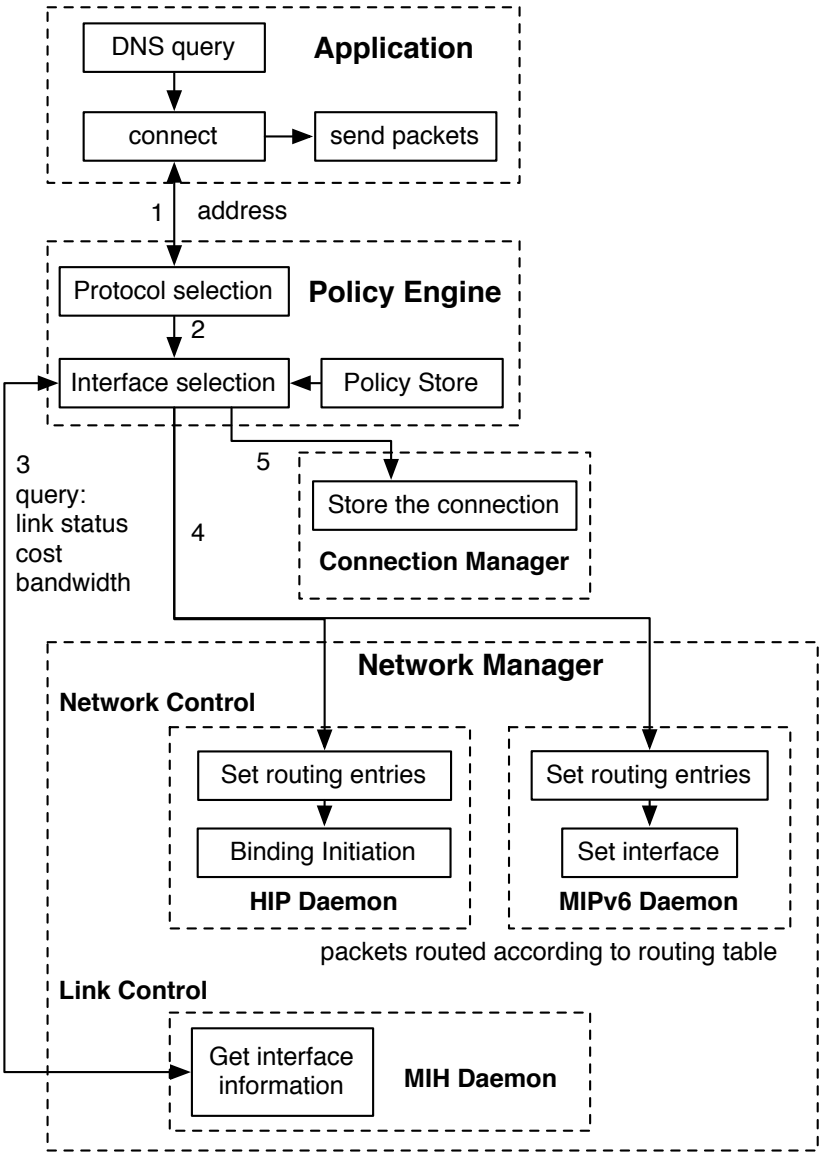


Figure 4-1 New Connection Control Flow

Holding a host name, the application first uses DNS service to get the IP address (the obtained address can be a HIT from HIP DNS server), it then calls the SINE connect method using this address. The sine connect method will actually send this address to Policy Engine (PE), and PE will decide which protocol to use according to the address space. It then gets the policies from the policy store and tries to find a matching policy according to the network information (interface, cost, bandwidth) it achieves from NM. After PE makes a decision, it records the connection into Connection Manager and sends command to NM to set preferred interfaces for this connection in either HIP or MIPv6 daemon depending on the protocol decision it has made. After that, the real connect packet will be routed into Internet according to the routing table that HIP or MIPv6 has just set. All the data packets will follow this route.

When there is a link up/down event that happens, which is probably triggered by removing a network interface, inserting a new network interface, or host moving from one network to another kind of network, the NM will notify the Event Listener inside Policy Engine in order to trigger the Policy Engine to make a new decision and do handovers:

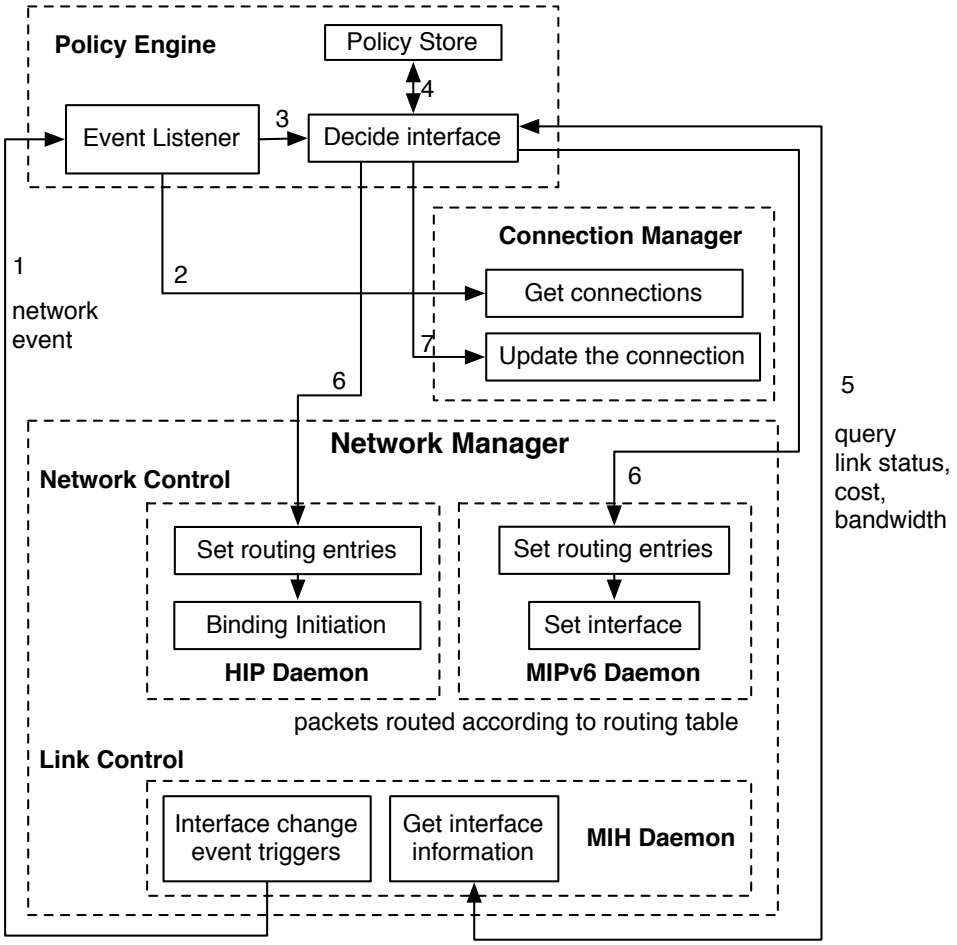


Figure 4-2 Network Change Event Control Flow

Whenever there is a link status change event happens, NM will notify the Event Listener inside PE and triggers a new decision. PE then queries all influenced connections from the Connection Manager and make decisions one by one according to the policies. For each connection, it does the same thing as described in 4.2.1. It queries link status, cost and bandwidth information from NM, finds matching policies and then sets the preferred interface in HIP or MIPv6 daemon. The following data packets will go through the new routing table that HIP or MIPv6 has updated.

4.3 SMART Routing

As we already mentioned, SINE middleware makes crucial routing decisions to utilize SINE protocols between the host and destination nodes. When the source & destination nodes are SINE enabled, the onus is on the Network system to work with appropriate protocols and control the communication. In such scenarios, only the Policy Engine component of SINE middleware evaluates the policy settings corresponding to the application and communicates the constraints to the Network System.

When the destination is not SINE enabled, the middleware resorts to a SMART routing technique. The SMART routing starts with the discovery of a SMART router in the path between the source & destination nodes. The SMART Router is just another node running the SINE platform and is capable of acting as an intermediary in the transmission of data. Smart routing does not affect the Policy engine functionalities describe in the previous scenario. In the absence of SMART router, SINE middleware defaults to plain old TCP/IP methods for communication. None of the SINE features can be used in this mode.

- SINE Enabled Destinations: The communication between two participating nodes does not involve any intermediate nodes. Regardless of the Application – Middleware interfacing method, middleware invokes the Policy Engine, which in turn evaluates the policy settings corresponding to the application.
- Non – SINE Destinations: For Non-SINE destinations, the middleware uses a smart routing strategy subject to certain pre-requisites. If there existed a SMART Router in the path between the source & destination nodes, all the features supported by SINE Platform are applicable to the route from the host to the intermediary router. From the router, the communication follows the traditional way. Let's take a look at what we mean by a SMART Router.

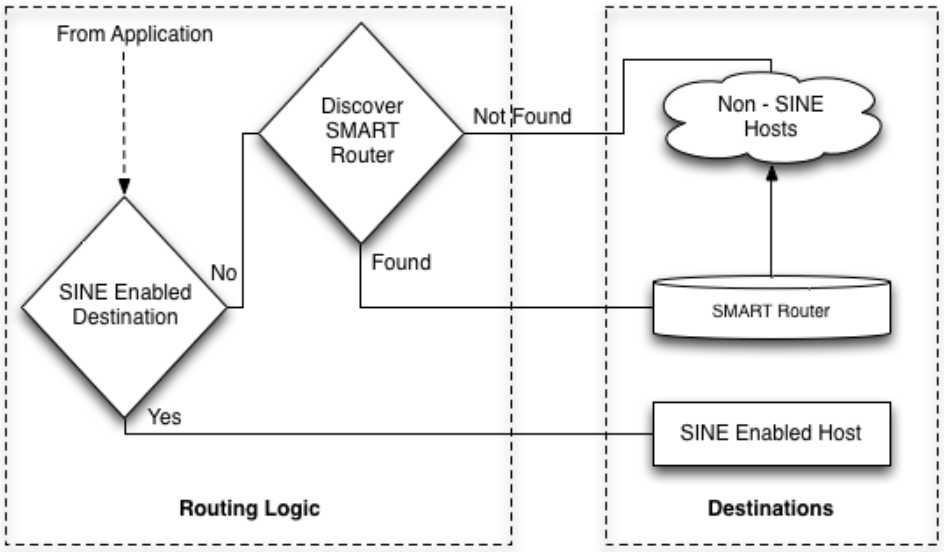


Fig SINE Routing Logic with SMART Router

4.3.1 SMART Router

A SMART Router is a node in the network that is SINE enabled and exposes SOCKS Service to the outside world. In a way, the SMART Router is a SINE Host that not only serves it's applications but only acts a SOCKS proxy for applications belonging to other hosts. The need for such an entity arises due to the non-availability of SINE support at the destination host. Instead of resorting to the traditional communication, we maximize the use of SINE enabled path by having such an intermediate party in the network. The communication between the source and the SMART router cannot be differentiated from a SINE to SINE scenario. The only difference can be that the payload being transferred is that of a session layer instead of an application layer. The mechanism by which SINE hosts relay their Socket calls to the router are described in the implementation section below.

The source node has to be configured for a SMART Router at the moment. But we have envisioned ideas to automate the discovery of a SMART Router.

Discovery

Currently, we configure the IP Address of the SMART Router in every SINE enabled host. But due to varying path taken by packets to reach a destination, we need a router discovery module. The discovery of such a router can be done by sending probing packets similar to ICMP to the destination. A SMART router if found on the path can be configured to respond to such requests. Once such a router is discovered by the source, all further communication can be delegated via this super-node. It is up to the implementation of the Network Protocols to make this a secure communication. Since Network Manager works with protocols like HIP that security aspect in-built to the platform. Security in general has to be considered in such implementations. We have currently deferred this aspect to future enhancements.

Implementation using SOCKS Relay

The implementation of a SMART router can be done using some of the concepts of SOCKS Relay. A SOCKS Relay is a mechanism by which a SOCKS Proxy forwards the session layer payload to another SOCKS proxy in the network. Since the SOCKS proxy running on a SMART Router can listen to external connections, we configure a SOCKS relay in source node to the relay SOCKS requests to the SMART Router if the destination is not SINE enabled. An SRELAY server can easily configured using its configuration file. But this aspect needs to be automated for the SINE platform.

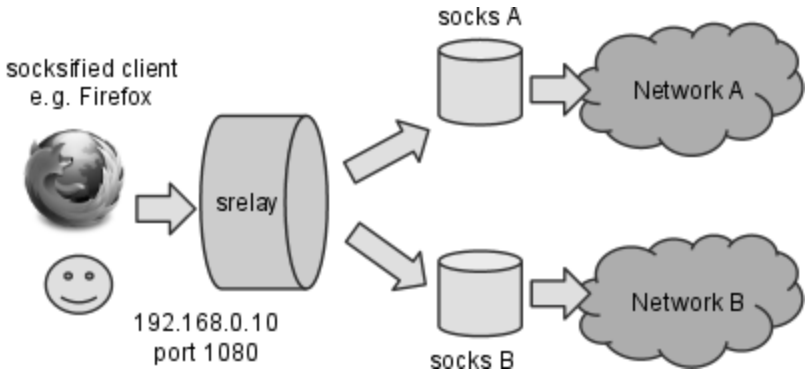


Fig 7: S-Relay Mechanism [Courtesy: - http://socks-relay.sourceforge.net]

Configuration in /etc/srelay.conf:

```
# FOR Non-SINE destinations – behave as a SOCKS Relay
# dest              dest-port       next-hop    next-port
<NON_SINE_NET>  any       <SMART_ROUTER>    1080

# For SINE Enabled Nodes – behave as regular SOCKS Proxy
# dest              dest-port       next-hop    next-port
0.0.0.0                 any
```

Failed Discovery of SMART Router

The last scenario is the failure to detect SMART Router in the path between the source and destination. In such scenarios, we have no choice but to follow the traditional TCP/IP way for communication. In such a scenario, all features of SINE platform are turned off.

Conclusion

In this report, we discuss the architecture of SINE platform and its components. We have given emphasis to the various integration points be it at the user-space or in the kernel. We show that the SINE Middleware is at the core orchestrating the control flow with the Network System as the workhorse. Middleware is also responsible for routing decisions for communicating with a SINE supported end-point or a Non-SINE end-point. Connection Manager is an entity within the middleware responsible for maintain the state of socket connections. Policy Engine inside the SINE middleware controls the business logic to select interfaces and drives the key decisions. We discussed in depth about the policy language and the policy architecture (Policy Based Network Management). The language considers the following parameters while making a decision - interface status, cost of the current network, bandwidth of the interface and location. However it still lacks some features and should include a functional language rather than use predicate logic.

We also discuss in detail the architecture and implementation of Network Manager, as well as how it is integrated with Policy Engine and how it works in SINE middleware. It communicates with three lower layer protocols including MIH, HIP and MIPv6. Those protocols are implemented by open source codes but have been modified to allow Network Manager to get control over them, like getting link information from MIH, setting preferred interfaces in HIP, MIPv6 and changing the routing tables. The Network Manager provides link status, cost and bandwidth information query to Policy Engine and accepts commands to set preferred interfaces of HIP and MIPv6 and routing entries for local handover. However, it still lacks some functions that we originally propose such as detecting nearby networks and giving alerts, which we will study further in the future.

We also cover the security aspect of this design using an open source federated identity solution. The security manager uses the open source federated identity solution called Shibboleth. We have successfully extended this web single sign-on functionality to the SINE project to demonstrate the authentication of network interfaces. The identity provider and the service provider entities in Shibboleth correspond to home and foreign network interfaces. The decision to switch the network interface from the policy manager triggers the security manager to start the authentication process which returns an authorization decision to policy manager.

Future Work

There is good scope for innovation in the design of SINE middleware. Some of the modules left for future innovators include the briefly discussed aspect of SMART Router discovery. Being a crucial entity in spreading the adoptability of SINE platform, it is an important to deliver a fool proof mechanism that is secure and reliable. Also, there is a great interest to improvise the design of Connection Manager to respond faster to the every changing state of sockets and still maintain the integrity of data.

In order to optimize the policy engine we have following proposals - Rather than using the principle of ORs, the policy engine should be intelligent enough to use the best interface in terms of cost or bandwidth, though it is ranked as a lower option. The Policy Engine should have two types of policies – strict policy and a lenient policy. The Engine should have a mechanism wherein it learns from past experience the margin by which a particular condition of a rule missed to be satisfied. This can be used as an offset to the integer values provided by the policy maker in the future event handlings. To further extend the policy engine from the user interface mode, we plan to build our own grammar and implement a compiler.

By leveraging the MIH protocol, the current Network Manager can detect some basic network events such as link up/down. But there are plenty of other kinds of network events that need to be detected, including network discovery, network connection, signal change, and so on. More technologies need to be studied to achieve the ultimate goal of the Network Manager. Also, the network control part, which is currently done by sending commands to HIP and Mobile IPv6 protocols, needs to be de-coupled from the Network Manager. A new component called Policy Driver is proposed to do the actions according to the decisions of Policy Engine.

The current implementation of security manager uses a web-based user authentication to demonstrate the shibboleth functionality. In future, we can remove the web-browser from the authentication process and use the certificate store in the mobile devices to provide the credentials bypassing the user authentication using browser completely.

References

[1] A. Singh, S. Addepalli, G. Ormazabal, H. Schulzrinne, SINE: Smart InterNet Evolution, A Pervasive Vision for a Fully Integrated Next Generation Network, work in progress, Dec 2012.
[2] R. Moskowitz, P. Nikander, P. Jokela, T. Henderson, Host Identity Protocol, IETF RFC 5201, April 2008.
[3] P. Nikander, T. Henderson, C. Vogt, J. Arkko, End-Host Mobility and Multihoming with the Host Identity Protocol, IETF RFC 5206, April 2008
[4] R. Draves, Default Address Selection for Internet Protocol version 6, IETF RFC 3484, February 2003
[5] OpenHIP Project, http://www.openhip.org/
[6] InfraHIP, HIP for Linux Project, http://infrahip.hiit.fi/
[7] C. Perkins, D. Johnson, J. Arkko, Mobility Support in IPv6, IETF RFC 6275, July 2011
[8] UMIP Project, http://www.umip.org/
[9] Petri Jokela, Teemu Rinta-aho, Tony Jokikyyny, Jorma Wall, Martti Kuparinen, Heikki Mahkonen, Jan Melén, Tero Kauppinen, Jouni Korhonen, Handover Performance with HIP and MIPv6, Wireless Communication Systems, 2004, pp. 324-328
[10] P. Nikander, J. Laganier, F. Dupont, An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID), IETF RFC 4843, April 2007
[11] V. Gupta, 802.21 Tutorial, IEEE 802.21, http://www.ieee802.org/21/, July 2006
[12] A. Agarwal, Pramod P J, and D. K Jain, Implementation of IEEE 802.21 based Media Independent Handover Services, Proceedings of the 32nd Asia-Pacific Advanced Network Meeting Implementation, August 2011
[13] L A N Man, Standard Committee, IEEE Standard for Local and metropolitan area networks— Part 21: Media Independent Handover Services, IEEE Std 802212008, January 2009
[14] ODTONE: Open 802.21 Project, http://helios.av.it.pt/projects/odtone
[15] E. Piri and K. Pentikousis, Towards a GNU/Linux IEEE 802.21 Implementation, Communications, IEEE International Conference, June 2009
[16] http://shibboleth.internet2.edu/
[17] https://wiki.shibboleth.net/confluence/display/SHIB2/Home
[18] http://www.switch.ch/aai/demo/
[19] Shibboleth installation details are uploaded on SINE wiki-

- Identity Provider
  https://wiki.cs.columbia.edu/display/sine/Identity+Provider+Installation+Guide
- Service Provider
  https://wiki.cs.columbia.edu/display/sine/Service+Provider+Installation+Guide
- Setting up and configuring Test Bed
  https://wiki.cs.columbia.edu/display/sine/Setting+up+Shibboleth+Test+Bed

Appendix

SINE Socket Interface (sine_socket.h)

```
int sine_socket (int af, int type, int protocol, int app_guid);
int sine_bind (int s, const void *addr, int  addrlen);
int sine_send(int s, void *msg, int len, int flags);
int sine_connect(int s, void *addr, int  addrlen);
int sine_getsockopt (int s, int  level, int optname, void *optval, int
*optlen);
int sine_select(int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
int sine_shutdown(int s, int how);
int sine_close(int s);
int sine_listen(int s, int backlog);
int sine_recv(int s, void *buf, int len, int flags) ;
int sine_accept(int s, void *addr, int *addrlen);
int sine_setsockopt (int         s,
                     int         level,
                     int         optname,
                     void       *optval,
                     int         optlen);

void sine_kill(void);
int sine_fcntl      (int fd);
void init_policy_engine();
```

Connection Manager Interface (connectionMgr.h)

```
enum connection_status {
   NEW,
   SEND,
   RECV,
   CONNECT,
   LISTEN,
   FIN
};

struct connection{
   int sockfd;
   int parent_sockfd;
   int app_guid;
   int policy_guid;
   enum connection_status status;
   socklen_t addrlen;
   struct sockaddr *addr;
   struct connection *next;
};
```

```
void init_connection_tbl();
int update_connection_bind(int sockfd, const void *addr, int addrlen);
int update_connection_status(int sockfd, enum connection_status
status);
void add_connection(int sockfd, int app_guid);
void add_child_connection(int sockfd, int new_sockfd);
void print_connections();
char * convert_status(enum connection_status status);
```

Policy Engine Interface (policy.h)

```
char* policyModel(int appID, int sockfd);
void policyListenerController();
```

Network Manager Interface (nm.h)

```
void sine_getIfStatus(const char* ifname, char* result);
int sine_getCost(const char* ifname);
int sine_getBandwidth(const char* ifname);
char* sine_getInfo(const char* ifname);
void sine_setHipIf(int sockfd, const char* ifname);
void sine_confirmHipIf(int sockfd, const char* ifname);
void sine_setMipIf(int sockfd, const char* ifname);
bool sine_appRegister(int sockfd);
bool sine_appDeregister(int sockfd);
```

Security Manager Interface (sm.h)

```
bool sine_shib(char* ifname);
bool sine_shib_browser(char* ifname);
bool sine_hip_encrypt(int encrypt_enable);
```