OpenSolaris NWAM Phase 0.5 "picea"

James Carlson
james.d.carlson@sun.com

Version 0.6

Abstract

This document describes the design of the NWAM Phase 0.5 (aka *picea*) extensions for networking control GUI interaction in OpenSolaris. The document is broken into three sections: a description of what was removed from the daemon, a user's guide for the new library interface, and finally a discussion of the internal operation of these interfaces. [1]

1 Status Quo

1.1 The Old Way

The existing "Phase 0" NWAM (Network Automagic) daemon in Nevada contains a set of built-in rules regarding interface selection, and a number of points where `/usr/bin/zenity` is invoked as an external utility to gather user input when the next step for the daemon is unclear.

This design has a number of important artifacts. Significant among them is that `nwamd` itself does not have a direct connection to the console user. Instead, it must "poll" the console by reading through the utmpx entries, scanning for "console", and then assuming that this is the right user to contact by way of setuid and X $DISPLAY set to ":0.0".

The scheme works, but the polling aspect introduces delays: the daemon sits around for as long as two minutes after the user logs in, doing nothing, and often making the user think that something is wrong. In addition to that, the interface works on demand from the daemon. The user isn't in control, and can't ask for information when desired. This often produces an impression that nwamd is hard to control or that its operation is inscrutible. And because the policies are essentially hard-coded into nwamd itself, the user cannot easily modify those policies when desired.

1.2 The New Goal

Because it will take some time to develop the entire NWAM Phase 1 design to address these problems, and because the problems are serious enough to inconvenience users, we will make a short-term adjustment to address the most acute issues.

This short-term plan is "NWAM Phase 0.5" or "*picea*." It contains, by necessity, some work that is assumed to be "throw-away." It is possible that some of the elements designed for *picea* may end up being useful for Phase 1, but that is not an intentional goal of this plan.

It is also not a *picea* goal to fix all known problems with NWAM. The primary issues are the GUI design and lack of observability, and these will be addressed. Other issues, such as relative difficulty in configuring static addresses through the configuration file, lack of support for multiple active interfaces, and stability of underlying drivers (or lack thereof) will not necessarily be addressed.

1.3 The Removed Bits

The removed portions include:

1. The `valid_graphical_user` function in `util.c` is gone, along with the `popup_query` and `popup_info` SMF attributes. The daemon no longer needs to scan to find the user; the GUI will find the daemon.

2. The `display` function in `interface.c` has been removed. All events are now queued through the door-based interface to the GUI, which

then has responsibility for determining what to display to the user (and when to do so).

3. About one third of `wireless.c` disappears. This includes the extensive `zenity` formatting routines and related string handling, as well as most of the AP selection logic. The code that remains merely checks for a single known (already visited) AP in the case where the GUI is absent: if there's just one such AP and it has the best signal strength, then we automatically connect, just as before.

2 New Library Interface

The new library, called cleverly enough `libnwam`, provides a set of functions intended for use by a GUI. The library has one blocking (and thread-cancelable) function for getting queued events. The daemon expects that as long as the GUI is running, it will have at least one thread actively calling this function to gather events.

If the GUI blocks elsewhere and fails to gather waiting events for more than 10 seconds (an arbitrarily-chosen time), the daemon assumes that the GUI is gone, and reverts to the default auto-configuration-based behavior that the old nwamd would exhibit when unable to invoke zenity. Similarly, if the GUI processes the events too slowly, such that the nwamd internal queue overflows, the daemon will revert to the default automatic behavior and flush waiting events.

At initial connection time, defined as the first event-wait call when the daemon has no active client, a special "initial" message is delivered to the client. The client is expected to ignore this message if received at start-up time; it indicates that the daemon has transitioned from automatic mode into active client mode. If the "initial" message is received at any other time, it means that an error has occurred: the daemon has lost track of the client. Note that a lack of an "initial" message is not an error; it will happen when the GUI disconnects and then reconnects within 10 seconds and no queue overflow has occurred. Queued events are not lost on disconnect.

All of the library interfaces are accessible through ⟨libnwam.h⟩ (using -lnwam), and are designed to be usable without needing other header files or libraries.

2.1 Events

Events are returned using the `libnwam_event_data_t` structure by the `libnwam_wait_event` function. See the next subsection for function calling details.

Not all structure members are used for all events. Each event lists the structure members that are valid for that event.

deInitial is the initial event. It signals when the daemon has changed state from "no active client present" to "active client present." If seen at any time other than initial client start-up, it indicates a communications error with the client.

Other than the `led_type` member, no structure members are used.

deInterfaceUp signals an interface transitioning "up" at the IP layer. This is separate from the LLP-switching messages, because an interface may go up and down without necessarily switching LLP, such as when being renumbered by DHCP.

Note that *picea* supports only one active interface at a time. If another interface goes up, and it is not a higher priority interface, then the daemon will take the interface back down.

The `led_v4address`, `led_prefixlen` and `led_interface` structure members are provided with this event.

deInterfaceDown signals that an interface has transitioned "down" at the IP layer. It may indicate an interface that has failed, been taken down administratively, or has been automatically shut down by nwamd.

This event provides `led_interface` and `led_cause`.

deInterfaceAdded signals a new hardware interface detected by the hotplug subsystem.

This event provides `led_interface`.

deInterfaceRemoved signals that a hardware interface has been removed from the system.

This event provides `led_interface`.

deWlanConnectFail occurs only on wireless interfaces. It indicates that the previous attempt to an AP has failed. Note that there are two sources of connection attempts: commands from the client and automatic attempts by nwamd itself in the special case of a single known AP.

This event provides `led_interface`.

deWlanDisconnect occurs only on wireless interfaces. It indicates that the interface has disconnected from an AP.

This event provides `led_interface` and `led_wlan`.

deWlanConnected occurs only on wireless interfaces. It indicates that the interface has been connected to an AP.

This event provides `led_interface` and `led_wlan`.

deLLPSelected signals selection of a "primary" interface (identical to LLP in nwamd) for operation.

This event provides `led_interface`.

deLLPUnselected signals de-selection of a "primary" interface (LLP) by the daemon.

This event provides `led_interface` and `led_cause`.

deULPActivated indicates that the user-supplied check-conditions script has provided an Upper Layer Profile based on the LLP.

This event provides `led_interface`, but the string returned is the name of the user-supplied ULP.

deULPDeactivated indicates that the ULP has been deactivated due to changes in the underlying LLP.

This event provides `led_interface`, but the string returned is the name of the user-supplied ULP.

deScanChange indicates that a wireless AP scan has completed, and that new APs were discovered during that scan, or APs were dropped from the list.

This event provides `led_interface`, and sets the WLAN cache (see the function interfaces below).

deScanSame indicates that a wireless AP scan has completed, but that no new APs were discovered during that scan.

This event provides `led_interface`, and sets the WLAN cache (see the function interfaces below).

deWlanKeyNeeded indicates that nwamd progress for this interface is currently blocked because the indicated access point requires a key. The user must provide a key, select a different access point, or switch LLPs (interfaces) to continue. The daemon will automatically continue if the GUI disconnects or if a more preferred interface becomes available.

If the GUI disconnects, nwamd will wait 10 seconds, and then invoke the default event, which is to fail the connection.

This event provides `led_interface` and `led_wlan`.

deWlanSelectionNeeded indicates that nwamd progress for this interface is blocked because there is no clear choice of AP. It occurs when there is no known AP in the list, or when there is more than one AP in the list, or the single known AP doesn't have the highest signal strength among all APs.

The user must select one of the WLANs in the captured list or switch LLPs to continue. The daemon will automatically continue if the GUI disconnects, a rescan shows a single known AP, or a more preferred interface becomes available.

If the GUI disconnects, nwamd will wait 10 seconds, and then invoke the default event, which is to run the autoconf procedure to select an arbitrary AP.

This event provides `led_interface`, and sets the WLAN cache (see the function interfaces below).

2.2 Functions

Only one function is intended to block for an indeterminate amount of time: the event wait function. A thread blocked in that one function can be canceled. All others block only as necessary to obtain synchronization with the daemon.

Events are discrete and global to the daemon. This means that if multiple threads (or separate clients) call the blocking event wait function, they will each receive some arbitrary subset of the events. For that reason there should be at most one caller.

The functions internally use `door_call`, and loop on EINTR for ease-of-use. They will return EBADF if the server itself fails. If this happens, the failure state is persistent; all subsequent calls will return EBADF until the client closes the connection with libnwam_fini and then a successful call to libnwam_init.

There are three main data structures returned to the caller. All use storage allocated by the library, and must be freed using the provided functions when the caller is finished with the data.

int libnwam_init(int waittime);

This function opens the connection to the nwamd service. The waittime parameter is 0 to open immediately or fail, -1 to wait "forever", or a positive integer representing the number of seconds to wait for a connection with the daemon. It returns 0 on success, and -1 with errno set on failure (only if waittime is not -1). The documented errno values are EMFILE (too many descriptors already open), ENOENT (server is not running), and EACCES (access denied).

int libnwam_fini(void);

This function closes the connection to the nwamd service. Process exit will also close the connection normally. This function nreturn 0 on success, and -1 with errno set on failure. No special errno values are documented.

libnwam_event_data_t *libnwam_wait_event(void);

This function waits for an event to occur, then allocates storage for that event, removes it from the queue, and returns the storage to the caller. The caller must free this storage using libnwam_free_event when finished handling the event. On failure, this function returns NULL and sets errno. The only documented error is EBADF, which means that the connection to the server has failed.

void libnwam_free_event(libnwam_event_data_t *led);
This function frees an event record allocated by the above function.

libnwam_llp_t *libnwam_get_llp_list(uint_t *nllp);
This returns an allocated array of LLP (per-interface) records describing the interfaces known to nwam on the system. The caller-supplied unsigned integer parameter is assigned the number of records in the array. On error or when there are no network interfaces in the system, the function returns NULL and sets errno. The only documented error is EBADF, which means that the connection to the server has failed.

void libnwam_free_llp_list(libnwam_llp_t *llp);
This function deallocates the storage that is returned by the function above. The user must call this function to return allocated storage to the heap.

int libnwam_set_llp_priority(const char *ifname, int prio);
The relative priority of the named interface is set to the given value and updated in stable storage. Priorities start at zero (highest priority) and go upwards. Interfaces are initially given priority values arbitrarily by nwamd: the first interface read from the configuration file is 0, the next is 1, until the end of file, then the network interfaces in the system are scanned in an undefined order, and assigned sequential priority numbers in the order encountered. The behavior when two LLPs have the same priority is undefined, but not an error. Negative values are not legal. The function returns 0 on success, and -1 with errno set on failure. The documented errors are ENXIO (the named interface does not exist), EINVAL (negative priority specified), and EBADF (connection to the server has failed).

int libnwam_lock_llp(const char *ifname);
The named interface is "locked" so that nwamd's state machine will select no other interface, regardless of failure. To unlock the interface, specify the empty string. The function returns 0 on success, and -1 with errno set on failure. The documented errors are ENXIO (the named interface does not exist) and EBADF (connection to the server has failed).

libnwam_wlan_t *libnwam_get_wlan_list(uint_t *nwlan);
This returns a list of APs as an allocated array, and the supplied unsigned integer parameter is set to the number of APs in the array. This list is established when any one of the events that are documented as setting the WLAN cache is returned (deScanNewAPs, deScanSame, and deWlanSelectionNeeded). The function returns NULL on error or if there are no cached APs. The only documented error is EBADF, which means that the connection to the server has failed.

void libnwam_free_wlan_list(libnwam_wlan_t *wlan);
This function frees an AP list allocated by the above function.

libnwam_known_ap_t *libnwam_get_known_ap_list(uint_t *nap);
This returns a list of known APs as an allocated array, and the supplied unsigned integer parameter is set to the number of APs in the array. The "known" APs are those that this system has connected to in the past, and thus will automatically connect to again in the future. The function returns NULL on error or if there are no known APs. The only documented error is EBADF, which means that the connection to the server has failed.

void libnwam_free_known_ap_list(libnwam_known_ap_t *ka);
This function frees a known AP list allocated by the above function.

int libnwam_add_known_ap(const char *essid, const char *bssid);
This function adds a new AP to the known AP list in stable storage. Both pointers must be non-NULL. The ESSID must be a non-empty string. On success, it returns 0. On error, it returns -1 and sets errno. Defined error codes are EEXIST (AP is already known) and EBADF (connection to the server has failed).

int libnwam_delete_known_ap(const char *essid, const char *bssid);
This function adds an AP from the known AP list in stable storage. Both pointers must be non-NULL. The ESSID must be a non-empty string. On success, it returns 0. On error, it returns -1 and sets errno. Defined error codes are ENXIO (AP is not known) and EBADF (connection to the server has failed).

int libnwam_select_wlan(const char *ifname, const char *essid, const char *bssid);

This function selects an access point (AP) for the indicated wireless interface. The essid and bssid values may be the empty string, in which case the driver's autoconfiguration routine is run. If the daemon previously indicated a need for a selection (deWlanSelectionNeeded), then successful return from this function causes the daemon to proceed with interface configuration. If the interface was already connected, then it is disconnected from its current AP and connected to the selected one. This may cause the interface to go down and back up (restarting DHCP) if the ESSID changes. On success, it returns 0. On error, it returns -1 and sets errno. Defined error codes are ENXIO (unknown interface), EINVAL (not a wireless interface or bad essid/bssid format), ENETUNREACH (connect failed), ENODEV (no such AP known), and EBADF (connection to the server has failed).

int libnwam_wlan_key(const char *ifname, const char *essid, const char *bssid, const char *key);

This function sets or changes the encryption key (in cleartext) for an access point (AP) on the indicated wireless interface. If the daemon previously indicated a need for a key (deWlanKeyNeeded), then this function causes the daemon to proceed with interface configuration. On success, it returns 0. On error, it returns -1 and sets errno. Defined error codes are ENXIO (unknown interface), EINVAL (not a wireless interface, or connect failed), ENODEV (no such AP known), and EBADF (connection to the server has failed).

int libnwam_start_rescan(const char *ifname);

This triggers an immediate rescan for access points on the indicated wireless interface. If a scan is already in progress, a new one is not started. Return values are 0 on success, and -1 with errno set on failure. Defined error codes are ENXIO (unknown interface), EINVAL (not a wireless interface), EINPROGRESS (scan already running), and EBADF (connection to the server has failed).

2.3 Data Structures

2.3.1 libnwam_event_data_t

The event data structure contains the following members:

led_type (nwam_descr_evtype_t) is an enumerated value representing the descriptive event type (see Events above).

led_cause (nwam_diag_cause_t) is an enumerated value representing the diagnostic cause of the event. This field is set only with deInterface-Down and deLLPUnselected.

led_v4address (struct in_addr) is the IPv4 address of the interface; used with deInterfaceUp.

led_prefixlen (int) is the length of the IPv4 netmask on the interface (0 to 32); used with deInterfaceUp.

led_wlan (libnwam_wlan_attr_t; see structure below) are the Access Point attributes; used with deWlanKeyNeeded, deWlanDisconnect, and deWlanConnected.

led_interface (char *) is the name of the interface; used with all events except deInitial.

2.3.2 libnwam_llp_t

The Link Layer Profile (interface) structure is used to convey information about the known system interfaces.

llp_interface (const char *) is the interface name.

llp_pri (int) is the interface priority; lower numbers indicate higher priority, with 0 being highest priority. Interface priority is used to select a profile when multiple interfaces are usable.

llp_type (enum interface_type) is the type of interface, and may be one of IF_WIRED, IF_WIRELESS, IF_TUN (tunnel), or IF_UNKNOWN.

llp_ipv4src (libnwam_ipv4src_t) is the daemon's configured address data source for this interface; IPV4SRC_STATIC or IPV4SRC_DHCP.

llp_primary (boolean_t) is a flag set if the interface is currently selected as the primary interface. At most one interface in a list will have this set.

llp_locked (boolean_t) is a flag set if the interface is specified as "locked" by the client. At most one interface in a list will have this set.

2.3.3 libnwam_wlan_t

The WLAN structure describes a scanned access point.

wlan_attrs (libnwam_wlan_attr_t; see structure below) are the access point attributes.

wlan_interface (const char *) is the name of the wireless interface.

wlan_known (boolean_t) is a flag set if the AP is "known," or has been selected by the user in the past.

wlan_haskey (boolean_t) is a flag set if the AP has a stored encryption key.

wlan_connected (boolean_t) is a flag set if the interface is connected to this AP.

2.3.4 libnwam_wlan_attr_t

The WLAN attributes structure is used with both the event structure (the led_wlan member) and the WLAN structure (the wlan_attrs) member.

wla_essid (const char *) is the ESSID ("network name") for the access point. It is not necessarily unique.

wla_bssid (const char *) is the BSSID ("base station MAC address") for the access point. It is intended to be a unique identifier.

wla_secmode (const char *) is the security (or encryption) mode, and may be "none", "wep", or "wpa".

wla_strength (const char *) is the signal strength, and is given as "very weak", "weak", "good", "very good", or "excellent".

wla_mode (const char *) is the IEEE 802.11 communications mode in use. It is currently a single letter, “a”, “b”, or “g”.

wla_speed (const char *) is a string representing the nominal speed of the network, in megabits per second.

wla_auth (const char *) is the authentication type of the network, and is “open” or “shared”.

wla_bsstype (const char *) is the type of network, and is “bss” (basic service set; normal AP operation), “ibss” (independent basic service set; ad-hoc network), and “any” (automatic).

wla_channel (int) is the radio channel in use.

2.3.5 libnwam_known_ap_t

The Known AP structure describes a previously-visited (“known”) access point. These are maintained in stable storage.

ka_essid (const char *) is the ESSID (“network name”) for the access point. It is not necessarily unique.

ka_bssid (const char *) is the BSSID (“base station MAC address”) for the access point. It is intended to be a unique identifier, and may be the empty string for wildcard entries.

ka_haskey (boolean_t) is a flag set if the AP has a stored encryption key.

3 Internal Door-Based Service

3.1 Synchronization

The overall design of nwamd is event-driven, though not actually a single state machine. The events are queued up by the action of multiple threads, and the main thread services these events and modifies the various data structures based on the events.

The intended GUI programming interface for `libnwam` is a set of simple synchronous commands, in order to avoid as much as possible the complexities involved with a strictly message-passing design.

In order to make this happen, and to pass back sensible errors to the GUI client, we need to provide synchronization between the door-based threads from the GUI client and the main event-servicing loop in nwamd.

This could be done by having the door functions send a message, then wait for a response to be delivered by the main thread, but that would require new logic to gather the response, deal with unexpected responses, and deal with a lack of a response (for robustness), and deal with abandoned responses as well (for canceled threads). It's similar in complexity to the STREAMS ioctl(2) logic.

A simpler approach is to use a new mutex, called `machine_lock`, to guard the main event dispatcher, and provide exclusion between the door threads and the main dispatcher when manipulating internal data structures.

3.2 Active Client Maintenance

The "active client" state exists when we know that we have a client making calls to the event-returning door function. When we are in this state, the daemon defers policy decisions to the client, and queues door events as they occur. When not in this state, the daemon makes automatic decisions based on built-in defaults, and does not queue door events. We must distinguish these states in order to allow for unattended operation when the user has not logged in (e.g., during system boot).

The "active" state exists when there's actually a client blocked waiting for a new event. At other times, the client is not known to exist. Because switching out of this state is destructive (it causes default actions to be taken), we wish to make the state cover a short period after that call returns until a new call is made.

It would be possible to rely on DOOR_UNREF_MULTI to detect the client termination instead of relying on time. If we did that, though, a GUI client that is halted by a debugger or stuck waiting for some other interface to respond could cause `nwamd` to become unresponsive. The result could be a loss of network connectivity due to a lack of event handling.

Even if we assume that this failure mode never happens, there's another problem with design: it's possible for the queue of waiting events to overflow. In that case, we must do something, and the only logical thing to do (other than halting the daemon to wait for the GUI to catch up) is to

treat it as client failure, just as with a timeout.

We choose the simpler scheme. When the event-wait function is called, we set the active-client flag. When it returns, and there are no more client threads blocked on events, then we send a new `EV_DOOR_TIME` event to the state machine. This event causes the main thread to call back in through `check_door_life`, which checks a 10-second timer (`client_expire`), and clears the active flag if the timer fires. This same logic is used on queue overflow: the queue is reset, and the active flag is cleared.

As an optimization, the new event is sent only if the current timer would expire further in the future than the one needed to detect a dead client. Thus, the usual activity is that once the client returns and causes the timer to be set, we wait for the timer to expire, and then push the timer for a new 10 second interval, rather than starting and restarting it for each call.

On switching out of active state, a `EV_RESELECT` event is sent to the main event queue. This causes the interface processing logic to retry its delayed operations. With the client gone, the `request_wlan_selection` and `request_wlan_key` functions will fail, causing the calling code to invoke the default auto-select actions.

3.3 Door Event Queue

The door event queue is a simple static array with get/put pointers and a single mutex protecting the use of the array. Callers must copy data into or out of the array and advance the pointer with the lock held, and this is enforced by internal utility functions.

An earlier version of this design used the global machine lock for the event queue as well, but this proved problematic, as there were cases where some callers may have that lock already held and others do not. Rather than use recursive mutexes (which are rancid in the best case, and highly toxic when used with condition variables in the worst case), I split this out into a separate mutex that's held only for the duration of queue manipulation.

Due to the way the `door_return` function works, it is necessary for the caller to use stack space or statics (not malloc(3C) or lock- or reference-protected memory) to return data to the caller. For events, this is just a simple fixed-length structure copied to a stack variable. For the functions

that return lists, alloca(3C) must be used to obtain usable stack space.

4 Security

To control access to nwamd from the GUI, we will create a new authorization defined this way in auth_attr:

```
solaris.network.autoconf:::\
  Network Automatic Configuration::\
  help=NetworkAutoconf.html
```

There will be a corresponding profile defined in prof_attr:

```
Network Autoconf:::\
  Manage network automatic configuration via nwamd:\
  auths=solaris.network.autoconf;help=RtNetAutoconf.html
```

This authorization will be assigned to the "Console User" and "Network Management" profiles in prof_attr, and the usual OpenSolaris user will get this authorization by being the Primary Administrator.

To check that the user has the necessary authorization to select network interfaces for the system, we will call door_ucred(3C) to get the user's real UID, and then call chkauthattr(3SECDB) to verify that this user has solaris.network.autoconf.

All calls into the library (including libnwam_init) require the specified authorization and the daemon checks for it. It would be possible to have multiple authorizations (one providing read-only status rights, and another providing read-write access to configuration), but because the picea interface is designed to have a single GUI process with all features rather than multiple simultaneous clients, such a division would serve no purpose. NWAM Phase 1 will likely need to address this differently.

Appendices

A Public Documentation License Notice

tation if you comply with the terms of this License. A copy of the License is available at:

`http://www.opensolaris.org/os/community/documentation/license`