WiFi overview

Johannes Martin Berg

2009-02-29
We’ll cover

- mac80211
- wext (and quickly forget about it)
- cfg80211/nl80211
- wpa_supplicant
- hostapd
Architecture – planned

userspace

nl80211  wext (for backward compatibility)

cfg80211

cfg80211_ops

mac80211 and future fullmac drivers
mac80211

• is a subsystem to the Linux kernel
• implements shared code for soft-MAC/half-MAC wireless devices
• contains MLME and other code, despite the name
Some notable additions to mac80211:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>HT/aggregation support</td>
<td>Intel</td>
</tr>
<tr>
<td>802.11s draft support</td>
<td>cozybit through o11s.org</td>
</tr>
<tr>
<td>802.11w draft support</td>
<td>Jouni Malinen (Atheros)</td>
</tr>
<tr>
<td>PS (infrastructure mode)</td>
<td>Kalle Valo (Nokia)</td>
</tr>
<tr>
<td>beacon processing offload</td>
<td>Vivek Natarajan (Atheros)</td>
</tr>
<tr>
<td></td>
<td>Kalle Valo (Nokia)</td>
</tr>
</tbody>
</table>
• TX/RX paths (including software en-/decryption)
• control paths for managed, IBSS, mesh
• some things for AP (e.g. powersave buffering)
• ...
• ieee80211_local/ieee80211_hw
• sta_info/ieee80211_sta
• ieee80211_conf
• ieee80211_bss_conf
• ieee80211_key/ieee80211_key_conf
• ieee80211_tx_info
• ieee80211_rx_status
• ieee80211_sub_if_data/ieee80211_vif
• each instance of these (hw is embedded into local) represents a wireless device
• ieee80211_hw is the part of ieee80211_local that is visible to drivers
• contains all operating information about a wireless device
• represents any station (peer)
• could be mesh peer, IBSS peer, AP, WDS peer
• would also be used for DLS peer
• ieee80211_sta is driver-visible part
• ieee80211_find_sta for drivers
• lifetime managed mostly with RCU
• hardware configuration
• most importantly - current channel
• intention: hardware specific parameters
• BSS configuration
• for all kinds of BSSes (IBSS/AP/managed)
• contains e.g. basic rate bitmap
• intention: per BSS parameters in case hardware supports creating/associating with multiple BSSes
Data structures – ieee80211_key/ieee80211_key_conf

- represents an encryption/decryption key
- ieee80211_key_conf given to driver for hardware acceleration
- ieee80211_key contains internal book-keeping and software encryption state
• most complicated data structure
• lives inside skb’s control buffer (cb)
• goes through three stages (substructure for each)
  • initialisation by mac80211 (control)
  • use by driver (driver_data/rate_driver_data)
  • use for TX status reporting (status)
• contains status about a received frame
• passed by driver to mac80211 with a received frame
• contains information about each virtual interface
• ieee80211_vif is passed to driver for those virtual interfaces the
driver knows about (not monitor, VLAN)
• contains sub-structures depending on mode
  • ieee80211_if_ap
  • ieee80211_if_wds
  • ieee80211_if_vlan
  • ieee80211_if_managed
  • ieee80211_if_ibss
  • ieee80211_if_mesh
- configuration
- receive path
- transmit path
- management/MLME
- all initiated from userspace (wext or nl80211)
- for managed and IBSS modes: triggers statemachine (on workqueue)
- some operations passed through to driver more or less directly (e.g. channel setting)
Main flows – receive path

- packet received by driver
- passed to mac80211’s rx function (ieee80211_rx) with rx_status info
- for each interface that the packet might belong to
  - RX handlers are invoked
  - data: converted to 802.3, delivered to networking stack
  - management: delivered to MLME
Main flows – transmit path

- packet handed to virtual interface’s ieee80211_subif_start_xmit
- converted to 802.11 format
- sent to master interface
- packet handed to ieee80211_master_start_xmit
- transmit handlers run, control information created
- packet given to driver
Main flows – transmit path

- ieee80211_tx_h_check_assoc
- ieee80211_tx_h_ps_buf
- ieee80211_tx_h_select_key
- ieee80211_tx_h_michael_mic_add
- ieee80211_tx_h_rate_ctrl
- ieee80211_tx_h_misc
- ieee80211_tx_h_sequence
- ieee80211_tx_h_fragment
- ieee80211_tx_h_encrypt
- ieee80211_tx_h_calculate_duration
- ieee80211_tx_h_stats
Main flows – management/MLME

START

SCAN_FIND_BSS
- scan done
- no BSS info
- no current prep

DIRECT_PROBE
- have recent info

AUTHENTICATE
- open disabled, shared enabled, WEP key configured
- not open/shared, leap enabled
- unsupported

AUTH_OPEN
- not open/shared, leap enabled
- unsupported

AUTH_SHARED
- unsupported

AUTH_LEAP
- unsupported

AUTH_SHARED_CHALLENGE
- reply OK

ASSOCIATE
- assoc response
- derived, disassoc
- not successful
- 3 x 0.2s
- denied

ASSOCIATED
- 3 x 0.2s
- 3 x 0.2s
- 3 x 0.2s
- not successful
- 3 x 0.2s
- denied

AP_PROBE
- 3 x 0.2s
- no frames for 2s

DISABLED
- 1 x 2s
- 3 x 0.2s
- unsupported
- leap disabled
Main flows – management/MLME

- START
  - SCAN_FIND_BSS
  - DIRECT_PROBE
  - AUTHENTICATE
  - AUTH_OPEN
  - AUTH_SHARED
  - AUTH_SHARED_CHALLENGE
  - AUTH_LEAP
  - ASSOCIATE
  - ASSOCIATED
  - AP_PROBE
  - DISABLED

- scan done
- no BSS info
- no current prepaid
- have recent info
- open disabled, shared enabled, WEP key configured
- BSS not found after 2 scans
- 3 x 0.2s
- denied
- assoc response
- not successful
- unsupported
Ok, so you didn’t want to know that precisely.

- requests from user are translated to internal variables
- state machine runs on user request
- normal procedure:
  - probe request/response
  - auth request/response
  - assoc request/response
  - notification to userspace
Simpler for IBSS:

- try to find IBSS
- join IBSS or create IBSS
- if no peers periodically try to find IBSS to join
Three main points

- configuration (from userspace)
- mac80211/rate control
- mac80211/driver
• Wireless extensions (wext, legacy)
• cfg80211 (which userspace talks to via nl80211)
• Rate control is semantically not part of driver
• per-driver selection of rate control algorithm
• rate control fills ieee80211_tx_info rate information
• rate control informed of TX status
• many driver methods (ieee80211_ops)
• mac80211 also has a lot of exported functions
• refer to include/net/mac80211.h
• config flows: mostly rtnl
• a lot of RCU-based synchronisation (sta_info, key management)
• mutex for interface list management
• spinlocks for various tightly constrained spots like sta list management, sta_info members etc.
• some more specialised locks
Quick questions on mac80211?
• all code is in net/wireless/wext.c
• not much code – drivers need to implement a lot
• userspace sets each parameter one by one
• driver tries to work with these parameters
• problem: is the user going to send a BSSID after the SSID?
Wireless extensions – handoff points

- netdev.wireless_handlers
  - contains array of standard and private handlers
  - handlers called by userspace via ioctl
- drivers send events via netlink
- a lot already handled in cfg80211 wext-compat
• thin layer between userspace and drivers/mac80211
• mainly sanity checking, protocol translations
• thicker than wext – sanity checking, bookkeeping, compat layer, ...

Linux Wireless
- userspace access to cfg80211 functionality
- defined in include/linux/nl80211.h
- currently used in userspace by iw, crda, wpa_supplicant, hostapd
• drivers register a struct wiphy with cfg80211
• this includes hardware capabilities like
  • bands and channels
  • bitrates per band
  • HT capabilities
  • supported interface modes
• needs to be done before registering netdevs
• netdev ieee80211_ptr links to registered wiphy
regulatory enforcement (overview)

- still work in progress
- relies on userspace helper (crda) to provide restriction information
- will update the list of registered channels and (optionally) notify driver
- create/remove virtual interfaces
- change type of virtual interfaces (provides wext handler)
- change ‘monitor flags’
- keeps track of interfaces associated with a wireless device
- will set all interfaces down on rfkill
• optional
• mostly for mac80211, though other appropriate uses exist
• only matching PHY parameters possible, all virtual interfaces are on one channel
• driver responsible for rejecting impossible configurations like IBSS+IBSS or similar
virtual interface types

• ad-hoc (IBSS)
• managed
• AP and AP_VLAN
• WDS
• mesh point
• monitor
  • can set monitor flags: control frames, other BSS frames
  • special case: cooked monitor
  • cooked monitor sees all frames no other virtual interface consumed
• monitor (replacing things like CONFIG_IPW2200_PROMISCUOUS and module parameter)
• switching modes like with iwconfig
• allow multiple interfaces, combining e.g. WDS and AP for wireless backhaul
• many more features than wext:
  • multiple SSIDs
  • channel specification
  • allows IE insertion
- NetworkManager/connman
- wpa_supplicant
- hostapd
- “userspace SME”
• internally modular architecture, supports multiple backends
• current version supports nl80211, wext no longer required
• current version can try nl80211 and fall back to wext
• actively maintained by Jouni Malinen (Atheros)
• implements (almost) the entire AP MLME
• works with mac80211 through nl80211
• requires working radiotap packet injection
• requires many of the nl80211 callbacks
• requires ‘cooked’ monitor interfaces
• actively maintained by Jouni Malinen (Atheros)
• API has separate auth/assoc
• needs to support multiple authentications simultaneously (WIP)
• supports adding arbitrary IEs into auth/assoc frames
• together this allows 802.11r
• auth/assoc state machine needed in cfg80211 for wext
• to do: add SME to cfg80211 for wext
Thanks for listening.

Questions?

http://wireless.kernel.org/
• beacon processing
  • beacon miss actions
  • signal strength monitoring
  • beacon change monitoring

• offload
  • don’t use software for above tasks
  • have device (firmware) do this
  • results in much fewer CPU wakeups
virtual interfaces

vif 1  vif 2  vif 3  ...

master interface

driver/hardware
• allow, in theory, multiple network interfaces on single hardware
• for example WDS and AP interfaces (to be bridged)
• for example multiple AP interfaces (multi-BSS)
• any number of monitor interfaces
• any number of AP_VLAN interfaces (to implement multi-SSID with single BSSID)
relevance to drivers

- drivers need to allow each interface type
- drivers need to support certain operations for certain interface types
- drivers can support multiple virtual interfaces
- but: drivers not notified of monitor interfaces
• used to configure hardware filters
• best-effort, not all filter flags need to be supported
• best-effort, not all filters need to be supported
• filter flags say which frames to pass to mac80211 – thus a filter flag is supported if that type of frames passed to mac80211
• passing more frames than requested is always permitted but may affect performance
monitor interfaces

- handled entirely in mac80211
- may affect filters depending on configuration
- it is possible to create a monitor interface that does not affect filters, can be useful for debugging (iw phy phy0 interface add moni0 type monitor flags none)
Even backup slides end somewhere.