內容

[CONNECT 2](#_Toc468438698)

[1.用户输入信息点击connect按钮后, 2](#_Toc468438699)

[2.wifiServiceImpl收到两条消息并处理 2](#_Toc468438700)

[3.WifiStateMachine收到SAVE\_NETWORK和CONNECT\_NETWORK两条消息首先处理SAVE\_NETWORK 2](#_Toc468438701)

[4.WifiStateMachine处理connect\_network 2](#_Toc468438702)

[5.wpas处理selectnetwork 3](#_Toc468438703)

[6.处理reconnect,什么都没做 10](#_Toc468438704)

[7.收到drv event (NL80211\_CMD\_CONNECT) 11](#_Toc468438705)

[8. EAPOL-Key交换流程分析 14](#_Toc468438706)

[9.wifiMonitor收到CTRL-EVENT-CONNECTED 26](#_Toc468438707)

[10.wifiStateMachine收到NETWORK\_CONNECTION\_EVENT并处理 26](#_Toc468438708)

# CONNECT

### 1.用户输入信息点击connect按钮后,

触发wifiSettings的onSubmit.

触发wifiManager.save(config,mSaveListener)和connect,

wifiManager.save

connet又调Wifimanger.connect()

WifiManager最终发送SAVE\_NETWORK和CONNECT\_NETWORK给wifiserviceImpl

### 2.wifiServiceImpl收到两条消息并处理

wifiServiceIpl将两条消息发给wifiStateMachine

### 3.WifiStateMachine收到SAVE\_NETWORK和CONNECT\_NETWORK两条消息首先处理SAVE\_NETWORK

WifiStateMachine当前处于disconnectedState

首先处理SAVE\_NETWORK,最终由connectmodeState处理.

result = mWifiConfigManager.saveNetwork(config, WifiConfiguration.UNKNOWN\_UID);

NetworkUpdateResult saveNetwork(WifiConfiguration config, int uid)

{

WifiConfiguration conf;

NetworkUpdateResult result = addOrUpdateNetworkNative(config, uid);

int netId = result.getNetworkId();

conf = mConfiguredNetworks.getForCurrentUser(netId);

…….

saveConfig();

...

return result;

}

4.WifiStateMachine处理connect\_network**,,**由connectmodestate处理

netId = message.arg1;

config = (WifiConfiguration) message.obj;

mWifiConnectionStatistics.numWifiManagerJoinAttempt++;

boolean updatedExisting = false;

if (config != null)

{

String configKey = config.configKey(true /\* allowCached \*/);

WifiConfiguration savedConfig = mWifiConfigManager.getWifiConfiguration(configKey);

if (savedConfig != null)

{

config = savedConfig;

config.ephemeral = false;

mWifiConfigManager.updateNetworkSelectionStatus(config,

WifiConfiguration.NetworkSelectionStatus.NETWORK\_SELECTION\_ENABLE);

updatedExisting = true;

}

result = mWifiConfigManager.saveNetwork(config, message.sendingUid);

netId = result.getNetworkId();

}

config = mWifiConfigManager.getWifiConfiguration(netId);

mTargetNetworkId = netId;

autoRoamSetBSSID(netId, "any");

……///待补充

if (mWifiConfigManager.selectNetwork(config, /\* updatePriorities = \*/ true,

message.sendingUid) && mWifiNative.reconnect())

{

lastConnectAttemptTimestamp = System.currentTimeMillis();

targetWificonfiguration = mWifiConfigManager.getWifiConfiguration(netId);

mSupplicantStateTracker.sendMessage(WifiManager.CONNECT\_NETWORK);

replyToMessage(message, WifiManager.CONNECT\_NETWORK\_SUCCEEDED);

if (didDisconnect) {

transitionTo(mDisconnectingState);

} else if (updatedExisting && getCurrentState() == mConnectedState &&

getCurrentWifiConfiguration().networkId == netId) {

updateCapabilities(config);

} else

{

transitionTo(mDisconnectedState);

}

### 5.wpas处理selectnetwork

wpa\_supplicant\_ctrl\_iface\_select\_network首先获取该网络信息，然后调用wpa\_supplicant\_select\_network尝试与该网络associate

static int wpa\_supplicant\_ctrl\_iface\_select\_network(struct wpa\_supplicant \*wpa\_s, char \*cmd)

{

int id;

struct wpa\_ssid \*ssid;

char \*pos;

if (os\_strncmp(cmd, "any", 3) == 0)

{

ssid = NULL;

} else

{

id = atoi(cmd);

ssid = wpa\_config\_get\_network(wpa\_s->conf, id);

if (ssid == NULL) {

return -1;

}

if (ssid->disabled == 2) {

return -1;

}

}

pos = os\_strstr(cmd, " freq=");

……….

wpa\_s->scan\_min\_time.sec = 0;

wpa\_s->scan\_min\_time.sec = 0;

wpa\_supplicant\_select\_network(wpa\_s, ssid);

}

#### 5.1 wpa\_supplicant\_select\_network

和网络联系

///attempt association with a network

1. 标记其他网络disabled,
2. 检测当前网络是否是请求的网络
3. 根据ssid->mode设置wpa\_s->connect\_without\_scan
4. 调用wpa\_supplicant\_fast\_associate
5. wpas\_notify\_network\_selected

void wpa\_supplicant\_select\_network(struct wpa\_supplicant \*wpa\_s, struct wpa\_ssid \*ssid)

{

struct wpa\_ssid \*other\_ssid;

int disconnected = 0;

if (ssid)

wpas\_clear\_temp\_disabled(wpa\_s, ssid, 1);

for (other\_ssid = wpa\_s->conf->ssid; other\_ssid; other\_ssid = other\_ssid->next)

{

int was\_disabled = other\_ssid->disabled;

if (was\_disabled == 2)

continue; /\* do not change persistent P2P group data \*/

other\_ssid->disabled = ssid ? (ssid->id != other\_ssid->id) : 0;

if (was\_disabled && !other\_ssid->disabled)

wpas\_clear\_temp\_disabled(wpa\_s, other\_ssid, 0);

if (was\_disabled != other\_ssid->disabled)

wpas\_notify\_network\_enabled\_changed(wpa\_s, other\_ssid);

}

………….

if (ssid)

{

wpa\_s->current\_ssid = ssid;

eapol\_sm\_notify\_config(wpa\_s->eapol, NULL, NULL);

wpa\_s->connect\_without\_scan =

(ssid->mode == WPAS\_MODE\_MESH) ? ssid : NULL;

os\_free(wpa\_s->next\_scan\_freqs);

wpa\_s->next\_scan\_freqs = NULL;

} else {

wpa\_s->connect\_without\_scan = NULL;

}

wpa\_s->disconnected = 0;

wpa\_s->reassociate = 1;

if (wpa\_s->connect\_without\_scan[[1]](#footnote-1) ||

wpa\_supplica[[2]](#footnote-2)nt\_fast\_associate(wpa\_s) != 1) {

wpa\_s->scan\_req = NORMAL\_SCAN\_REQ;

wpa\_supplicant\_req\_scan(wpa\_s, 0, disconnected ? 100000 : 0);

}

if (ssid)

wpas\_notify\_network\_selected(wpa\_s, ssid);

}

#### 5.2wpa\_supplicant\_fast\_associate

int wpa\_supplicant\_fast\_associate(struct wpa\_supplicant \*wpa\_s)

{

return wpas\_select\_network\_from\_last\_scan(wpa\_s, 0, 1);

}

1. 获取目标无线网络的wpa\_bss对象
2. 判断是否漫游，是则什么都不做
3. 注销当前网络
4. 调用wpa\_supplicant\_connect

static int wpas\_select\_network\_from\_last\_scan(struct wpa\_supplicant \*wpa\_s,

int new\_scan, int own\_request)

{

struct wpa\_bss \*selected;

struct wpa\_ssid \*ssid = NULL;

int time\_to\_reenable = wpas\_reenabled\_network\_time(wpa\_s);

……….

selected = wpa\_supplicant\_pick\_network(wpa\_s, &ssid);

if (selected) {

int skip;

skip = !wpa\_supplicant\_need\_to\_roam(wpa\_s, selected, ssid);

if (skip) {

if (new\_scan)

wpa\_supplicant\_rsn\_preauth\_scan\_results(wpa\_s);

return 0;

}

if (ssid != wpa\_s->current\_ssid &&

wpa\_s->wpa\_state >= WPA\_AUTHENTICATING) {

wpa\_s->own\_disconnect\_req = 1;

wpa\_supplicant\_deauthenticate(

wpa\_s, WLAN\_REASON\_DEAUTH\_LEAVING);

}

if (wpa\_supplicant\_connect(wpa\_s, selected, ssid) < 0) {

wpa\_dbg(wpa\_s, MSG\_DEBUG, "Connect failed");

return -1;

}

if (new\_scan)

wpa\_supplicant\_rsn\_preauth\_scan\_results(wpa\_s);

return 1;

}

}

#### 5.3wpa\_supplicant\_connect

1.调用wpa\_supplicant\_associate,请求associate

2. wpa\_supplicant\_associate,最终将wpas\_start\_assoc\_cb添加到radiowork

int wpa\_supplicant\_connect(struct wpa\_supplicant \*wpa\_s,

struct wpa\_bss \*selected,

struct wpa\_ssid \*ssid)

{

……

If(wpa\_s->reassociat[[3]](#footnote-3)e||…….)

{

if (wpa\_supplicant\_scard\_init(wpa\_s, ssid))

{

wpa\_supplicant\_req\_new\_scan(wpa\_s, 10, 0);

return 0;

}

wpa\_supplicant\_associate(wpa\_s, selected, ssid);

}

…….

Return 0;

}

void wpa\_supplicant\_associate(struct wpa\_supplicant \*wpa\_s,

struct wpa\_bss \*bss, struct wpa\_ssid \*ssid)

{

…….

if (radio\_work\_pending(wpa\_s, "connect")) {

return;

}

wpas\_abort\_ongoing\_scan(wpa\_s);

cwork = os\_zalloc(sizeof(\*cwork));

if (cwork == NULL)

return;

cwork->bss = bss;

cwork->ssid = ssid;

if (radio\_add\_work[[4]](#footnote-4)(wpa\_s, bss ? bss->freq : 0, "connect", 1,

wpas\_start\_assoc\_cb, cwork) < 0) {

os\_free(cwork);

}

}

#### 5.4wpas\_start\_assoc\_cb(struct wpa\_radio\_work \*work, int deinit)

1.取消扫描

2.清空之前的wpa IE

3.清空driver中的key设置

3.填充wlan driver发起关联请求的struct wpa\_driver\_associate\_params类型的对象

4.向driver发起associate请求

{

wpa\_s->connect\_work = work;

os\_memcpy(prev\_bssid, wpa\_s->bssid, ETH\_ALEN);

os\_memset(&params, 0, sizeof(params));

wpa\_s->reassociate = 0;

wpa\_s->eap\_expected\_failure = 0;

if (bss &&

(!wpas\_driver\_bss\_selection(wpa\_s) || wpas\_wps\_searching(wpa\_s)))

{

}//bss&&(!wpas\_driver\_bss\_selection(wpa\_s)||….)

else {???????????????????????????????

wpa\_msg(wpa\_s, MSG\_INFO, "Trying to associate with SSID '%s'",

wpa\_ssid\_txt(ssid->ssid, ssid->ssid\_len));

os\_memset(wpa\_s->pending\_bssid, 0, ETH\_ALEN);

}

……

wpa\_supplicant\_cancel\_scan(wpa\_s);

…..

////clear the possibly used WPA IE from the \* previous association. \*/

///清空上一次association时使用的WPA/RSN IE信息这些信息保存在wpa\_supplicant

对象的assoc\_wpa\_ie（类型为u8\*）对应的buffer中

wpa\_sm\_set\_assoc\_wpa\_ie(wpa\_s->wpa, NULL, 0);

……

if (bss && (wpa\_bss\_get\_vendor\_ie(bss, WPA\_IE\_VENDOR\_TYPE) ||

wpa\_bss\_get\_ie(bss, WLAN\_EID\_RSN)) &&

wpa\_key\_mgmt\_wpa(ssid->key\_mgmt)) {

int try\_opportunistic;

try\_opportunistic = (ssid->proactive\_key\_caching[[5]](#footnote-5) < 0 ?

wpa\_s->conf->okc :

ssid->proactive\_key\_caching) &&

(ssid->proto & WPA\_PROTO\_RSN);

if (pmksa\_cache\_set\_current[[6]](#footnote-6)(wpa\_s->wpa, NULL, bss->bssid,

ssid, try\_opportunistic) == 0)

eapol\_sm\_notify\_pmkid\_attempt(wpa\_s->eapol);

wpa\_ie\_len = sizeof(wpa\_ie);

if (wpa\_supplicant\_set\_suites[[7]](#footnote-7)(wpa\_s, bss, ssid,

wpa\_ie, &wpa\_ie\_len)) {

wpa\_msg(wpa\_s, MSG\_WARNING, "WPA: Failed to set WPA "

"key management and encryption suites");

wpas\_connect\_work\_done(wpa\_s);

return;

}

}

……

///清除wlan driver中的key设置

wpa\_clear\_keys(wpa\_s, bss ? bss->bssid : NULL);

use\_crypt = 1;

cipher\_pairwise = wpa\_s->pairwise\_cipher;

cipher\_group = wpa\_s->group\_cipher;

……..

wpa\_supplicant\_set\_state(wpa\_s, WPA\_ASSOCIATING);

if (bss) {

params.ssid = bss->ssid;

params.ssid\_len = bss->ssid\_len;

if (!wpas\_driver\_bss\_selection(wpa\_s) || ssid->bssid\_set) {

……

}

params.bssid\_hint = bss->bssid;

params.freq\_hint = bss->freq;

params.pbss = bss\_is\_pbss(bss);

}

……..

///填充参数

params.wpa\_ie = wpa\_ie;

params.wpa\_ie\_len = wpa\_ie\_len;

params.pairwise\_suite = cipher\_pairwise;

params.group\_suite = cipher\_group;

params.key\_mgmt\_suite = wpa\_s->key\_mgmt;

params.wpa\_proto = wpa\_s->wpa\_proto;

params.auth\_alg = algs;

params.mode = ssid->mode;

params.bg\_scan\_period = ssid->bg\_scan\_period;

for (i = 0; i < NUM\_WEP\_KEYS; i++) {

if (ssid->wep\_key\_len[i])

params.wep\_key[i] = ssid->wep\_key[i];

params.wep\_key\_len[i] = ssid->wep\_key\_len[i];

}

params.wep\_tx\_keyidx = ssid->wep\_tx\_keyidx;

…..

///设置是否丢球未加密数据包.use\_crypt=1

params.drop\_unencrypted = use\_crypt;

ret = wpa\_drv\_associate(wpa\_s, &params);

if (wpa\_s->key\_mgmt == WPA\_KEY\_MGMT\_WPA\_NONE)

{…..}….

Else

{

int timeout = 60;

if (assoc\_failed) {

/\* give IBSS a bit more time \*/

timeout = ssid->mode == WPAS\_MODE\_IBSS ? 10 : 5;

} else if (wpa\_s->conf->ap\_scan == 1) {

/\* give IBSS a bit more time \*/

timeout = ssid->mode == WPAS\_MODE\_IBSS ? 20 : 10;

}

///设置身份认证超时时间

wpa\_supplicant\_req\_auth\_timeout(wpa\_s, timeout, 0);

}

…….

old\_ssid = wpa\_s->current\_ssid;

wpa\_s->current\_ssid = ssid;

if (!wpas\_driver\_bss\_selection(wpa\_s) || ssid->bssid\_set)

wpa\_s->current\_bss = bss;

wpa\_supplicant\_rsn\_supp\_set\_config(wpa\_s, wpa\_s->current\_ssid);

wpa\_supplicant\_initiate\_eapol(wpa\_s);

if (old\_ssid != wpa\_s->current\_ssid)

wpas\_notify\_network\_changed(wpa\_s);

}

#### 5.5 wpa\_drv\_associate

static inline int wpa\_drv\_associate(struct wpa\_supplicant \*wpa\_s,

struct wpa\_driver\_associate\_params \*params)

{

if (wpa\_s->driver->associate) {

return wpa\_s->driver->associate(wpa\_s->drv\_priv, params);

}

return -1;

}

static int wpa\_driver\_nl80211\_associate(

void \*priv, struct wpa\_driver\_associate\_params \*params)

{

struct i802\_bss \*bss = priv;

struct wpa\_driver\_nl80211\_data \*drv = bss->drv;

int ret = -1;

struct nl\_msg \*msg;

nl80211\_unmask\_11b\_rates(bss);

if (params->mode == IEEE80211\_MODE\_AP)

return wpa\_driver\_nl80211\_ap(drv, params);

if (params->mode == IEEE80211\_MODE\_IBSS)

return wpa\_driver\_nl80211\_ibss(drv, params);

if (!(drv->capa.flags & WPA\_DRIVER\_FLAGS\_SME)) {

enum nl80211\_iftype nlmode = params->p2p ?

NL80211\_IFTYPE\_P2P\_CLIENT : NL80211\_IFTYPE\_STATION;

if (wpa\_driver\_nl80211\_set\_mode(priv, nlmode) < 0)

return -1;

return wpa\_driver\_nl80211\_connect(drv, params);

}

………

}

static int wpa\_driver\_nl80211\_connect(

struct wpa\_driver\_nl80211\_data \*drv,

struct wpa\_driver\_associate\_params \*params)

{

int ret;

if (params->bssid)

os\_memcpy(drv->auth\_attempt\_bssid, params->bssid, ETH\_ALEN);

else

os\_memset(drv->auth\_attempt\_bssid, 0, ETH\_ALEN);

ret = wpa\_driver\_nl80211\_try\_connect(drv, params);

if (ret == -EALREADY) {

wpa\_printf(MSG\_DEBUG, "nl80211: Explicitly "

"disconnecting before reassociation "

"attempt");

if (wpa\_driver\_nl80211\_disconnect(

drv, WLAN\_REASON\_PREV\_AUTH\_NOT\_VALID))

return -1;

ret = wpa\_driver\_nl80211\_try\_connect(drv, params);

}

return ret;

}

static int wpa\_driver\_nl80211\_try\_connect(

struct wpa\_driver\_nl80211\_data \*drv,

struct wpa\_driver\_associate\_params \*params)

{

msg = nl80211\_drv\_msg(drv, 0, NL80211\_CMD\_CONNECT);

////填充信息

ret = nl80211\_connect\_common(drv, params, msg);

……..

ret = nl80211\_set\_conn\_keys(params, msg);

ret = send\_and\_recv\_msgs(drv, msg, NULL, NULL);

}

### 6.处理reconnect,什么都没做

### 7.收到drv event (NL80211\_CMD\_CONNECT)

Do\_process\_drv\_event

{

case NL80211\_CMD\_CONNECT:

case NL80211\_CMD\_ROAM:

mlme\_event\_connect(drv, cmd,

tb[NL80211\_ATTR\_STATUS\_CODE],

tb[NL80211\_ATTR\_MAC],

tb[NL80211\_ATTR\_REQ\_IE],

tb[NL80211\_ATTR\_RESP\_IE],

tb[NL80211\_ATTR\_TIMED\_OUT],

NULL, NULL, NULL, NULL, NULL);

break;

}

mlme\_event\_connect 保存数据,通知wpa\_supplicant\_event处理

void wpa\_supplicant\_event(void \*ctx, enum wpa\_event\_type event,

union wpa\_event\_data \*data)

{

case EVENT\_ASSOC:

wpa\_supplicant\_event\_assoc(wpa\_s, data);

if (data && data->assoc\_info.authorized)

wpa\_supplicant\_event\_assoc\_auth(wpa\_s, data);

if (data) {

wpa\_msg(wpa\_s, MSG\_INFO,

WPA\_EVENT\_SUBNET\_STATUS\_UPDATE "status=%u",

data->assoc\_info.subnet\_status);

}

break;

}

static void wpa\_supplicant\_event\_assoc(struct wpa\_supplicant \*wpa\_s,

union wpa\_event\_data \*data)

{

u8 bssid[ETH\_ALEN];

int ft\_completed, already\_authorized;

int new\_bss = 0;

………….

eloop\_cancel\_timeout(wpas\_network\_reenabled, wpa\_s, NULL);

ft\_completed = wpa\_ft\_is\_completed(wpa\_s->wpa);

if (data && wpa\_supplicant\_event\_associ[[8]](#footnote-8)nfo(wpa\_s, data) < 0)

return;

if (wpa\_drv\_get\_bssid(wpa\_s, bssid) < 0) {

wpa\_dbg(wpa\_s, MSG\_ERROR, "Failed to get BSSID");

wpa\_supplicant\_deauthenticate(

wpa\_s, WLAN\_REASON\_DEAUTH\_LEAVING);

return;

}

wpa\_supplicant\_set\_state(wpa\_s, WPA\_ASSOCIATED);

if (os\_memcmp(bssid, wpa\_s->bssid, ETH\_ALEN) != 0) {

wpa\_dbg(wpa\_s, MSG\_DEBUG, "Associated to a new BSS: BSSID="

MACSTR, MAC2STR(bssid));

new\_bss = 1;

random\_add\_randomness(bssid, ETH\_ALEN);

os\_memcpy(wpa\_s->bssid, bssid, ETH\_ALEN);

os\_memset(wpa\_s->pending\_bssid, 0, ETH\_ALEN);

wpas\_notify\_bssid\_changed(wpa\_s);

if (wpa\_supplicant\_dynamic\_keys(wpa\_s) && !ft\_completed[[9]](#footnote-9)) {

wpa\_clear\_keys(wpa\_s, bssid);

}

if (wpa\_supplicant\_select\_config(wpa\_s) < 0) {

wpa\_supplicant\_deauthenticate(

wpa\_s, WLAN\_REASON\_DEAUTH\_LEAVING);

return;

}

}

#ifdef ANDROID

if (wpa\_s->conf->ap\_scan == 1) {

#else

if (wpa\_s->conf->ap\_scan == 1 &&

wpa\_s->drv\_flags & WPA\_DRIVER\_FLAGS\_BSS\_SELECTION) {

#endif

if (wpa\_supplicant\_assoc\_update\_ie(wpa\_s) < 0 && new\_bss)

wpa\_msg(wpa\_s, MSG\_WARNING,

"WPA/RSN IEs not updated");

}

wpas\_fst\_update\_mb\_assoc(wpa\_s, data);

#ifdef CONFIG\_SME

os\_memcpy(wpa\_s->sme.prev\_bssid, bssid, ETH\_ALEN);

wpa\_s->sme.prev\_bssid\_set = 1;

wpa\_s->sme.last\_unprot\_disconnect.sec = 0;

#endif /\* CONFIG\_SME \*/

wpa\_msg(wpa\_s, MSG\_INFO, "Associated with " MACSTR, MAC2STR(bssid));

if (wpa\_s->current\_ssid) {

/\* When using scanning (ap\_scan=1), SIM PC/SC interface can be

\* initialized before association, but for other modes,

\* initialize PC/SC here, if the current configuration needs

\* smartcard or SIM/USIM. \*/

wpa\_supplicant\_scard\_init(wpa\_s, wpa\_s->current\_ssid);

}

wpa\_sm\_notify\_assoc(wpa\_s->wpa, bssid);

#ifdef CONFIG\_WAPI

if (wpa\_s->key\_mgmt == WPA\_KEY\_MGMT\_WAPI\_PSK ||

wpa\_s->key\_mgmt == WPA\_KEY\_MGMT\_WAPI\_CERT){

wpa\_printf(MSG\_DEBUG, "[%s] WAPI AP Key Mgmt:%d ",

\_\_func\_\_, wpa\_s->key\_mgmt);

wapi\_supplicant\_event(wpa\_s, EVENT\_ASSOC, data);

} else {

#endif

if (wpa\_s->l2)

l2\_packet\_notify\_auth\_start(wpa\_s->l2);

already\_authorized = data && data->assoc\_info.authorized;

/\*

\* Set portEnabled first to FALSE in order to get EAP state machine out

\* of the SUCCESS state and eapSuccess cleared. Without this, EAPOL PAE

\* state machine may transit to AUTHENTICATING state based on obsolete

\* eapSuccess and then trigger BE\_AUTH to SUCCESS and PAE to

\* AUTHENTICATED without ever giving chance to EAP state machine to

\* reset the state.

\*/

if (!ft\_completed && !already\_authorized) {

eapol\_sm\_notify\_portEnabled(wpa\_s->eapol, FALSE);

eapol\_sm\_notify\_portValid(wpa\_s->eapol, FALSE);

}

if (wpa\_key\_mgmt\_wpa\_psk(wpa\_s->key\_mgmt) || ft\_completed ||

already\_authorized)

eapol\_sm\_notify\_eap\_success(wpa\_s->eapol, FALSE);

/\* 802.1X::portControl = Auto \*/

eapol\_sm\_notify\_portEnabled(wpa\_s->eapol, TRUE);

wpa\_s->eapol\_received = 0;

#ifdef CONFIG\_WAPI

}

#endif

…… else if (!ft\_completed) {

/\* Timeout for receiving the first EAPOL packet \*/

wpa\_supplicant\_req\_auth\_timeout(wpa\_s, 10, 0);

}

wpa\_supplicant\_cancel\_scan(wpa\_s);

…….

if (data) {

wmm\_ac\_notify\_assoc(wpa\_s, data->assoc\_info.resp\_ies,

data->assoc\_info.resp\_ies\_len,

&data->assoc\_info.wmm\_params);

if (wpa\_s->reassoc\_same\_bss)

wmm\_ac\_restore\_tspecs(wpa\_s);

}

}

### 8. EAPOL-Key交换流程分析

AP首先发送一个EAPOL帧给STA

l2\_packet\_receive收到eapol

由wpa\_supplicant\_rx\_eapol处理.

Wpa\_supplicant\_rx\_eapl第一次会设置认证超时任务,最后调用wpa\_sm\_rx\_eapol(wpa\_s->wpa, src\_addr, buf, len);处理收到的eapol数据.

wpa\_sm\_rx\_eapol会调用wpa\_supplicant\_process\_1\_of\_4()处理收到的数据

wpa\_supplicant\_process\_1\_of\_4()调用wpa\_supplicant\_send\_2\_of\_4()生成第二条eapol并发送.

AP收到第二条eapol消息,发送第三条eapol消息.

l2\_packet\_receive收到第三条消息调用wpa\_supplicant\_rx\_eapol处理,

wpa\_supplicant\_rx\_eapol最后调用wpa\_sm\_rx\_eapol()

wpa\_sm\_rx\_eapol会调用wpa\_supplicant\_process\_3\_of\_4 ()处理收到的数据

wpa\_supplicant\_process\_3\_of\_4 ()调用wpa\_supplicant\_send\_4\_of\_4 ()生成第二条eapol并发送.

安装key

然后将状态切换到WPA\_GROUP\_HANDSHAKE

调用wpa\_supplicant\_pairwise\_gtk()处理

wpa\_supplicant\_pairwise\_gtk()后调用wpa\_supplicant\_install\_gtk()函数安装GTK

wpa\_supplicant\_pairwise\_gtk()调用wpa\_supplicant\_key\_neg\_complete()将状态切到WPA\_COMPLETED.

最后调用wpa\_supplicant\_key\_neg\_complete()

wpa\_supplicant\_key\_neg\_complete()取消超时任务

设置WPAS的状态为WPA\_COMPLETED

连接动作完成

void wpa\_supplicant\_rx\_eapol(void \*ctx, const u8 \*src\_addr,

const u8 \*buf, size\_t len)

{

struct wpa\_supplicant \*wpa\_s = ctx;

wpa\_dbg(wpa\_s, MSG\_DEBUG, "RX EAPOL from " MACSTR, MAC2STR(src\_addr));

wpa\_hexdump(MSG\_MSGDUMP, "RX EAPOL", buf, len);

#ifdef CONFIG\_PEERKEY

if (wpa\_s->wpa\_state > WPA\_ASSOCIATED && wpa\_s->current\_ssid &&

wpa\_s->current\_ssid->peerkey &&

!(wpa\_s->drv\_flags & WPA\_DRIVER\_FLAGS\_4WAY\_HANDSHAKE) &&

wpa\_sm\_rx\_eapol\_peerkey(wpa\_s->wpa, src\_addr, buf, len) == 1) {

wpa\_dbg(wpa\_s, MSG\_DEBUG, "RSN: Processed PeerKey EAPOL-Key");

return;

}

#endif /\* CONFIG\_PEERKEY \*/

if (wpa\_s->wpa\_state < WPA\_ASSOCIATED ||

(wpa\_s->last\_eapol\_matches\_bssid &&

#ifdef CONFIG\_AP

!wpa\_s->ap\_iface &&

#endif /\* CONFIG\_AP \*/

os\_memcmp(src\_addr, wpa\_s->bssid, ETH\_ALEN) != 0)) {

/\*

\* There is possible race condition between receiving the

\* association event and the EAPOL frame since they are coming

\* through different paths from the driver. In order to avoid

\* issues in trying to process the EAPOL frame before receiving

\* association information, lets queue it for processing until

\* the association event is received. This may also be needed in

\* driver-based roaming case, so also use src\_addr != BSSID as a

\* trigger if we have previously confirmed that the

\* Authenticator uses BSSID as the src\_addr (which is not the

\* case with wired IEEE 802.1X).

\*/

wpa\_dbg(wpa\_s, MSG\_DEBUG, "Not associated - Delay processing "

"of received EAPOL frame (state=%s bssid=" MACSTR ")",

wpa\_supplicant\_state\_txt(wpa\_s->wpa\_state),

MAC2STR(wpa\_s->bssid));

wpabuf\_free(wpa\_s->pending\_eapol\_rx);

wpa\_s->pending\_eapol\_rx = wpabuf\_alloc\_copy(buf, len);

if (wpa\_s->pending\_eapol\_rx) {

os\_get\_reltime(&wpa\_s->pending\_eapol\_rx\_time);

os\_memcpy(wpa\_s->pending\_eapol\_rx\_src, src\_addr,

ETH\_ALEN);

}

return;

}

wpa\_s->last\_eapol\_matches\_bssid =

os\_memcmp(src\_addr, wpa\_s->bssid, ETH\_ALEN) == 0;

#ifdef CONFIG\_AP

if (wpa\_s->ap\_iface) {

wpa\_supplicant\_ap\_rx\_eapol(wpa\_s, src\_addr, buf, len);

return;

}

#endif /\* CONFIG\_AP \*/

if (wpa\_s->key\_mgmt == WPA\_KEY\_MGMT\_NONE) {

wpa\_dbg(wpa\_s, MSG\_DEBUG, "Ignored received EAPOL frame since "

"no key management is configured");

return;

}

if (wpa\_s->eapol\_received == 0 &&

(!(wpa\_s->drv\_flags & WPA\_DRIVER\_FLAGS\_4WAY\_HANDSHAKE) ||

!wpa\_key\_mgmt\_wpa\_psk(wpa\_s->key\_mgmt) ||

wpa\_s->wpa\_state != WPA\_COMPLETED) &&

(wpa\_s->current\_ssid == NULL ||

wpa\_s->current\_ssid->mode != IEEE80211\_MODE\_IBSS)) {

/\* Timeout for completing IEEE 802.1X and WPA authentication \*/

int timeout = 10;

if (wpa\_key\_mgmt\_wpa\_ieee8021x(wpa\_s->key\_mgmt) ||

wpa\_s->key\_mgmt == WPA\_KEY\_MGMT\_IEEE8021X\_NO\_WPA ||

wpa\_s->key\_mgmt == WPA\_KEY\_MGMT\_WPS) {

/\* Use longer timeout for IEEE 802.1X/EAP \*/

timeout = 70;

}

#ifdef CONFIG\_WPS

if (wpa\_s->current\_ssid && wpa\_s->current\_bss &&

(wpa\_s->current\_ssid->key\_mgmt & WPA\_KEY\_MGMT\_WPS) &&

eap\_is\_wps\_pin\_enrollee(&wpa\_s->current\_ssid->eap)) {

/\*

\* Use shorter timeout if going through WPS AP iteration

\* for PIN config method with an AP that does not

\* advertise Selected Registrar.

\*/

struct wpabuf \*wps\_ie;

wps\_ie = wpa\_bss\_get\_vendor\_ie\_multi(

wpa\_s->current\_bss, WPS\_IE\_VENDOR\_TYPE);

if (wps\_ie &&

!wps\_is\_addr\_authorized(wps\_ie, wpa\_s->own\_addr, 1))

timeout = 10;

wpabuf\_free(wps\_ie);

}

#endif /\* CONFIG\_WPS \*/

wpa\_supplicant\_req\_auth\_timeout(wpa\_s, timeout, 0);

}

wpa\_s->eapol\_received++;

if (wpa\_s->countermeasures) {

wpa\_msg(wpa\_s, MSG\_INFO, "WPA: Countermeasures - dropped "

"EAPOL packet");

return;

}

#ifdef CONFIG\_IBSS\_RSN

if (wpa\_s->current\_ssid &&

wpa\_s->current\_ssid->mode == WPAS\_MODE\_IBSS) {

ibss\_rsn\_rx\_eapol(wpa\_s->ibss\_rsn, src\_addr, buf, len);

return;

}

#endif /\* CONFIG\_IBSS\_RSN \*/

/\* Source address of the incoming EAPOL frame could be compared to the

\* current BSSID. However, it is possible that a centralized

\* Authenticator could be using another MAC address than the BSSID of

\* an AP, so just allow any address to be used for now. The replies are

\* still sent to the current BSSID (if available), though. \*/

os\_memcpy(wpa\_s->last\_eapol\_src, src\_addr, ETH\_ALEN);

if (!wpa\_key\_mgmt\_wpa\_psk(wpa\_s->key\_mgmt) &&

eapol\_sm\_rx\_eapol(wpa\_s->eapol, src\_addr, buf, len) > 0)

return;

wpa\_drv\_poll(wpa\_s);

if (!(wpa\_s->drv\_flags & WPA\_DRIVER\_FLAGS\_4WAY\_HANDSHAKE))

wpa\_sm\_rx\_eapol(wpa\_s->wpa, src\_addr, buf, len);

else if (wpa\_key\_mgmt\_wpa\_ieee8021x(wpa\_s->key\_mgmt)) {

/\*

\* Set portValid = TRUE here since we are going to skip 4-way

\* handshake processing which would normally set portValid. We

\* need this to allow the EAPOL state machines to be completed

\* without going through EAPOL-Key handshake.

\*/

eapol\_sm\_notify\_portValid(wpa\_s->eapol, TRUE);

}

}

int wpa\_sm\_rx\_eapol(struct wpa\_sm \*sm, const u8 \*src\_addr,

const u8 \*buf, size\_t len)

{

size\_t plen, data\_len, key\_data\_len;

const struct ieee802\_1x\_hdr \*hdr;

struct wpa\_eapol\_key \*key;

struct wpa\_eapol\_key\_192 \*key192;

u16 key\_info, ver;

u8 \*tmp = NULL;

int ret = -1;

struct wpa\_peerkey \*peerkey = NULL;

u8 \*key\_data;

size\_t mic\_len, keyhdrlen;

#ifdef CONFIG\_IEEE80211R

sm->ft\_completed = 0;

#endif /\* CONFIG\_IEEE80211R \*/

mic\_len = wpa\_mic\_len(sm->key\_mgmt);

keyhdrlen = mic\_len == 24 ? sizeof(\*key192) : sizeof(\*key);

if (len < sizeof(\*hdr) + keyhdrlen) {

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"WPA: EAPOL frame too short to be a WPA "

"EAPOL-Key (len %lu, expecting at least %lu)",

(unsigned long) len,

(unsigned long) sizeof(\*hdr) + keyhdrlen);

return 0;

}

hdr = (const struct ieee802\_1x\_hdr \*) buf;

plen = be\_to\_host16(hdr->length);

data\_len = plen + sizeof(\*hdr);

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"IEEE 802.1X RX: version=%d type=%d length=%lu",

hdr->version, hdr->type, (unsigned long) plen);

if (hdr->version < EAPOL\_VERSION) {

/\* TODO: backwards compatibility \*/

}

if (hdr->type != IEEE802\_1X\_TYPE\_EAPOL\_KEY) {

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"WPA: EAPOL frame (type %u) discarded, "

"not a Key frame", hdr->type);

ret = 0;

goto out;

}

wpa\_hexdump(MSG\_MSGDUMP, "WPA: RX EAPOL-Key", buf, len);

if (plen > len - sizeof(\*hdr) || plen < keyhdrlen) {

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"WPA: EAPOL frame payload size %lu "

"invalid (frame size %lu)",

(unsigned long) plen, (unsigned long) len);

ret = 0;

goto out;

}

if (data\_len < len) {

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"WPA: ignoring %lu bytes after the IEEE 802.1X data",

(unsigned long) len - data\_len);

}

/\*

\* Make a copy of the frame since we need to modify the buffer during

\* MAC validation and Key Data decryption.

\*/

tmp = os\_malloc(data\_len);

if (tmp == NULL)

goto out;

os\_memcpy(tmp, buf, data\_len);

key = (struct wpa\_eapol\_key \*) (tmp + sizeof(struct ieee802\_1x\_hdr));

key192 = (struct wpa\_eapol\_key\_192 \*)

(tmp + sizeof(struct ieee802\_1x\_hdr));

if (mic\_len == 24)

key\_data = (u8 \*) (key192 + 1);

else

key\_data = (u8 \*) (key + 1);

if (key->type != EAPOL\_KEY\_TYPE\_WPA && key->type != EAPOL\_KEY\_TYPE\_RSN)

{

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"WPA: EAPOL-Key type (%d) unknown, discarded",

key->type);

ret = 0;

goto out;

}

if (mic\_len == 24)

key\_data\_len = WPA\_GET\_BE16(key192->key\_data\_length);

else

key\_data\_len = WPA\_GET\_BE16(key->key\_data\_length);

wpa\_eapol\_key\_dump(sm, key, key\_data\_len, key192->key\_mic, mic\_len);

if (key\_data\_len > plen - keyhdrlen) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO, "WPA: Invalid EAPOL-Key "

"frame - key\_data overflow (%u > %u)",

(unsigned int) key\_data\_len,

(unsigned int) (plen - keyhdrlen));

goto out;

}

eapol\_sm\_notify\_lower\_layer\_success(sm->eapol, 0);

key\_info = WPA\_GET\_BE16(key->key\_info);

ver = key\_info & WPA\_KEY\_INFO\_TYPE\_MASK;

if (ver != WPA\_KEY\_INFO\_TYPE\_HMAC\_MD5\_RC4 &&

#if defined(CONFIG\_IEEE80211R) || defined(CONFIG\_IEEE80211W)

ver != WPA\_KEY\_INFO\_TYPE\_AES\_128\_CMAC &&

#endif /\* CONFIG\_IEEE80211R || CONFIG\_IEEE80211W \*/

ver != WPA\_KEY\_INFO\_TYPE\_HMAC\_SHA1\_AES &&

!wpa\_key\_mgmt\_suite\_b(sm->key\_mgmt) &&

sm->key\_mgmt != WPA\_KEY\_MGMT\_OSEN) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: Unsupported EAPOL-Key descriptor version %d",

ver);

goto out;

}

if (sm->key\_mgmt == WPA\_KEY\_MGMT\_OSEN &&

ver != WPA\_KEY\_INFO\_TYPE\_AKM\_DEFINED) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"OSEN: Unsupported EAPOL-Key descriptor version %d",

ver);

goto out;

}

if (wpa\_key\_mgmt\_suite\_b(sm->key\_mgmt) &&

ver != WPA\_KEY\_INFO\_TYPE\_AKM\_DEFINED) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"RSN: Unsupported EAPOL-Key descriptor version %d (expected AKM defined = 0)",

ver);

goto out;

}

#ifdef CONFIG\_IEEE80211R

if (wpa\_key\_mgmt\_ft(sm->key\_mgmt)) {

/\* IEEE 802.11r uses a new key\_info type (AES-128-CMAC). \*/

if (ver != WPA\_KEY\_INFO\_TYPE\_AES\_128\_CMAC) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"FT: AP did not use AES-128-CMAC");

goto out;

}

} else

#endif /\* CONFIG\_IEEE80211R \*/

#ifdef CONFIG\_IEEE80211W

if (wpa\_key\_mgmt\_sha256(sm->key\_mgmt)) {

if (ver != WPA\_KEY\_INFO\_TYPE\_AES\_128\_CMAC &&

sm->key\_mgmt != WPA\_KEY\_MGMT\_OSEN &&

!wpa\_key\_mgmt\_suite\_b(sm->key\_mgmt)) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: AP did not use the "

"negotiated AES-128-CMAC");

goto out;

}

} else

#endif /\* CONFIG\_IEEE80211W \*/

if (sm->pairwise\_cipher == WPA\_CIPHER\_CCMP &&

!wpa\_key\_mgmt\_suite\_b(sm->key\_mgmt) &&

ver != WPA\_KEY\_INFO\_TYPE\_HMAC\_SHA1\_AES) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: CCMP is used, but EAPOL-Key "

"descriptor version (%d) is not 2", ver);

if (sm->group\_cipher != WPA\_CIPHER\_CCMP &&

!(key\_info & WPA\_KEY\_INFO\_KEY\_TYPE)) {

/\* Earlier versions of IEEE 802.11i did not explicitly

\* require version 2 descriptor for all EAPOL-Key

\* packets, so allow group keys to use version 1 if

\* CCMP is not used for them. \*/

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: Backwards compatibility: allow invalid "

"version for non-CCMP group keys");

} else if (ver == WPA\_KEY\_INFO\_TYPE\_AES\_128\_CMAC) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: Interoperability workaround: allow incorrect (should have been HMAC-SHA1), but stronger (is AES-128-CMAC), descriptor version to be used");

} else

goto out;

} else if (sm->pairwise\_cipher == WPA\_CIPHER\_GCMP &&

!wpa\_key\_mgmt\_suite\_b(sm->key\_mgmt) &&

ver != WPA\_KEY\_INFO\_TYPE\_HMAC\_SHA1\_AES) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: GCMP is used, but EAPOL-Key "

"descriptor version (%d) is not 2", ver);

goto out;

}

#ifdef CONFIG\_PEERKEY

for (peerkey = sm->peerkey; peerkey; peerkey = peerkey->next) {

if (os\_memcmp(peerkey->addr, src\_addr, ETH\_ALEN) == 0)

break;

}

if (!(key\_info & WPA\_KEY\_INFO\_SMK\_MESSAGE) && peerkey) {

if (!peerkey->initiator && peerkey->replay\_counter\_set &&

os\_memcmp(key->replay\_counter, peerkey->replay\_counter,

WPA\_REPLAY\_COUNTER\_LEN) <= 0) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_WARNING,

"RSN: EAPOL-Key Replay Counter did not "

"increase (STK) - dropping packet");

goto out;

} else if (peerkey->initiator) {

u8 \_tmp[WPA\_REPLAY\_COUNTER\_LEN];

os\_memcpy(\_tmp, key->replay\_counter,

WPA\_REPLAY\_COUNTER\_LEN);

inc\_byte\_array(\_tmp, WPA\_REPLAY\_COUNTER\_LEN);

if (os\_memcmp(\_tmp, peerkey->replay\_counter,

WPA\_REPLAY\_COUNTER\_LEN) != 0) {

wpa\_dbg(sm->ctx->msg\_ctx, MSG\_DEBUG,

"RSN: EAPOL-Key Replay "

"Counter did not match (STK) - "

"dropping packet");

goto out;

}

}

}

if (peerkey && peerkey->initiator && (key\_info & WPA\_KEY\_INFO\_ACK)) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"RSN: Ack bit in key\_info from STK peer");

goto out;

}

#endif /\* CONFIG\_PEERKEY \*/

if (!peerkey && sm->rx\_replay\_counter\_set &&

os\_memcmp(key->replay\_counter, sm->rx\_replay\_counter,

WPA\_REPLAY\_COUNTER\_LEN) <= 0) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_WARNING,

"WPA: EAPOL-Key Replay Counter did not increase - "

"dropping packet");

goto out;

}

if (!(key\_info & (WPA\_KEY\_INFO\_ACK | WPA\_KEY\_INFO\_SMK\_MESSAGE))

#ifdef CONFIG\_PEERKEY

&& (peerkey == NULL || !peerkey->initiator)

#endif /\* CONFIG\_PEERKEY \*/

) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: No Ack bit in key\_info");

goto out;

}

if (key\_info & WPA\_KEY\_INFO\_REQUEST) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_INFO,

"WPA: EAPOL-Key with Request bit - dropped");

goto out;

}

if ((key\_info & WPA\_KEY\_INFO\_MIC) && !peerkey &&

wpa\_supplicant\_verify\_eapol\_key\_mic(sm, key192, ver, tmp, data\_len))

goto out;

#ifdef CONFIG\_PEERKEY

if ((key\_info & WPA\_KEY\_INFO\_MIC) && peerkey &&

peerkey\_verify\_eapol\_key\_mic(sm, peerkey, key192, ver, tmp,

data\_len))

goto out;

#endif /\* CONFIG\_PEERKEY \*/

if ((sm->proto == WPA\_PROTO\_RSN || sm->proto == WPA\_PROTO\_OSEN) &&

(key\_info & WPA\_KEY\_INFO\_ENCR\_KEY\_DATA)) {

if (wpa\_supplicant\_decrypt\_key\_data(sm, key, ver, key\_data,

&key\_data\_len))

goto out;

}

if (key\_info & WPA\_KEY\_INFO\_KEY\_TYPE) {

if (key\_info & WPA\_KEY\_INFO\_KEY\_INDEX\_MASK) {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_WARNING,

"WPA: Ignored EAPOL-Key (Pairwise) with "

"non-zero key index");

goto out;

}

if (peerkey) {

/\* PeerKey 4-Way Handshake \*/

peerkey\_rx\_eapol\_4way(sm, peerkey, key, key\_info, ver,

key\_data, key\_data\_len);

} else if (key\_info & WPA\_KEY\_INFO\_MIC) {

/\* 3/4 4-Way Handshake \*/

wpa\_supplicant\_process\_3\_of\_4(sm, key, ver, key\_data,

key\_data\_len);

} else {

/\* 1/4 4-Way Handshake \*/

wpa\_supplicant\_process\_1\_of\_4(sm, src\_addr, key,

ver, key\_data,

key\_data\_len);

}

} else if (key\_info & WPA\_KEY\_INFO\_SMK\_MESSAGE) {

/\* PeerKey SMK Handshake \*/

peerkey\_rx\_eapol\_smk(sm, src\_addr, key, key\_data\_len, key\_info,

ver);

} else {

if (key\_info & WPA\_KEY\_INFO\_MIC) {

/\* 1/2 Group Key Handshake \*/

wpa\_supplicant\_process\_1\_of\_2(sm, src\_addr, key,

key\_data, key\_data\_len,

ver);

} else {

wpa\_msg(sm->ctx->msg\_ctx, MSG\_WARNING,

"WPA: EAPOL-Key (Group) without Mic bit - "

"dropped");

}

}

ret = 1;

out:

bin\_clear\_free(tmp, data\_len);

return ret;

}

### 9.wifiMonitor收到CTRL-EVENT-CONNECTED

wifiMonitor收到ctrl-event-connected向wifiStateMachine发送NETWORK\_CONNECTION\_EVENT

### 10.wifiStateMachine收到NETWORK\_CONNECTION\_EVENT并处理

1.wifiStateMachine当前处于disconnectedState,由ConnectModeState处理 ,将状态切换到obtainingIpState

2. obtainingIpState的enter向ipmanager发送cmdstart

3. 处理cmd\_start,,将状态切到StartedState.StartedState的Enter,调用startIPv4()向dhcpclient发送cmd\_start\_dhcp

4. 由StoppedState处理CMD\_START\_DHCP,,将状态机切到WaitBeforeStartedState

waitBeforeStartState向ipManager发送CMD\_PRE\_DHCP\_ACTION

5.ipManager处理CMD\_PRE\_DHCP\_ACTION,向WifiStateMachine发送DhcpClient.CMD\_PRE\_DHCP\_ACTION

6.wifiStateMachine处于obtainingIpState,交给L2ConnectedState处理,处理函数是

handlePreDhcpSetup();处理完该消息向wifiStateMachine发送DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE

7. wifiStateMachine处理该消息像ipmanager发送EVENT\_PRE\_DHCP\_ACTION\_COMPLETE

8.Ipmanager处理EVENT\_PRE\_DHCP\_ACTION\_COMPLETE

向dhcpClient发送DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE,

dhcpClient处理CMD\_PRE\_DHCP\_ACTION\_COMPLETE,WaitBeforeStartedState处理,将状态切机到DhcpInitState

9. DhcpInitState的enter将发送CMD\_KICK给自己,并处理该消息,发送dhcp discover

10.收到offer报文并处理

ReceiveThread发送cmd\_received\_packet

最终由dhcpInitState的receivePacket处理,将状态机切到DhcpRequestingState

DhcpRequestingState的enter,,发送dhcp request报文

11. 收到ack并处理

ReceiveThread发送cmd\_received\_packet

由DhcpRequestingState的receivePacket处理,向IpManager发送post\_dhcp\_action,最后将状态机切到ConfiguringInterfaceState

12. IpManager处理CMD\_POST\_DHCP\_ACTION,取消定时任务,并向wifiStateMachine发送一系列消息

13. ConfiguringInterfaceState的Enter函数向ipManager发送CMD\_CONFIGURE\_LINKADDRESS

IpManager的StartedState处理该消息,最后向dhcpclient发送DhcpClient.EVENT\_LINKADDRESS\_CONFIGURED

14. Dhcpclient处于ConfiguringInterfaceState处理DhcpClient.EVENT\_LINKADDRESS\_CONFIGURED,将状态机切到mDhcpBoundState

WifiStateMachine收到消息并处理,最后将状态机切到connectedstate

case WifiMonitor.NETWORK\_CONNECTION\_EVENT:

if (DBG) log("Network connection established");

mLastNetworkId = message.arg1;

mLastBssid = (String) message.obj;

mWifiInfo.setBSSID(mLastBssid);

mWifiInfo.setNetworkId(mLastNetworkId);

mWifiQualifiedNetworkSelector

.enableBssidForQualityNetworkSelection(mLastBssid, true);

sendNetworkStateChangeBroadcast(mLastBssid);

transitionTo(mObtainingIpState);

break;

obtainingIpstate的enter

class ObtainingIpState extends State {

@Override

public void enter() {

linkDebouncing = false;

setNetworkDetailedState(DetailedState.OBTAINING\_IPADDR);

clearCurrentConfigBSSID("ObtainingIpAddress");

stopIpManager(); mIpManager.setHttpProxy(mWifiConfigManager.getProxyProperties(mLastNetworkId));

if (!TextUtils.isEmpty(mTcpBufferSizes)) {

mIpManager.setTcpBufferSizes(mTcpBufferSizes);

}

if (!mWifiConfigManager.isUsingStaticIp(mLastNetworkId)) {

final IpManager.ProvisioningConfiguration prov =

mIpManager.buildProvisioningConfiguration()

.withPreDhcpAction()

.withApfCapabilities(mWifiNative.getApfCapabilities())

.build();

mIpManager.startProvisioning(prov); //向ipmanager发送cmdstart

obtainingIpWatchdogCount++;

logd("Start Dhcp Watchdog " + obtainingIpWatchdogCount);

// Get Link layer stats so as we get fresh tx packet counters

getWifiLinkLayerStats(true);

sendMessageDelayed(obtainMessage(CMD\_OBTAINING\_IP\_ADDRESS\_WATCHDOG\_TIMER,

obtainingIpWatchdogCount,0), BTAINING\_IP\_ADDRESS\_GUARD\_TIMER\_MSEC);

} else {

………

}

public void startProvisioning(ProvisioningConfiguration req) {

getNetworkInterface();

mCallback.setNeighborDiscoveryOffload(true);

sendMessage(CMD\_START, new ProvisioningConfiguration(req));

}

#### 10.1IpManger处于StoppedState

处理cmd\_start,,将状态切到StartedState.

StartedState的Enter,调用startIPv4()向dhcpclient发送cmd\_start\_dhcp

public void enter() {

mStartTimeMillis = SystemClock.elapsedRealtime();

mApfFilter = ApfFilter.maybeCreate(mConfiguration.mApfCapabilities, mNetworkInterface,

mCallback, mMulticastFiltering);

if (mApfFilter == null) {

mCallback.setFallbackMulticastFilter(mMulticastFiltering);

}

if (mConfiguration.mEnableIPv6) {

startIPv6();

}

if (mConfiguration.mUsingIpReachabilityMonitor) {

mIpReachabilityMonitor = new IpReachabilityMonitor(

mContext,

mInterfaceName,

new IpReachabilityMonitor.Callback() {

@Override

public void notifyLost(InetAddress ip, String logMsg) {

mCallback.onReachabilityLost(logMsg);

}

});

}

if (mConfiguration.mEnableIPv4) {

if (!startIPv4()) {

transitionTo(mStoppingState);

}

}

}

private boolean startIPv4() {

// If we have a StaticIpConfiguration attempt to apply it and

// handle the result accordingly.

if (mConfiguration.mStaticIpConfig != null) {

……….

} else {

// Start DHCPv4.

mDhcpClient = DhcpClient.makeDhcpClient(mContext, IpManager.this, mInterfaceName);

mDhcpClient.registerForPreDhcpNotification();

mDhcpClient.sendMessage(DhcpClient.CMD\_START\_DHCP);

if (mConfiguration.mProvisioningTimeoutMs > 0) {

final long alarmTime = SystemClock.elapsedRealtime() +

mConfiguration.mProvisioningTimeoutMs;

mProvisioningTimeoutAlarm.schedule(alarmTime);

}

}

return true;

}

#### 10.2dhcpClient处理cmd\_start\_Dhcp

dhcpClient处于stoppedState

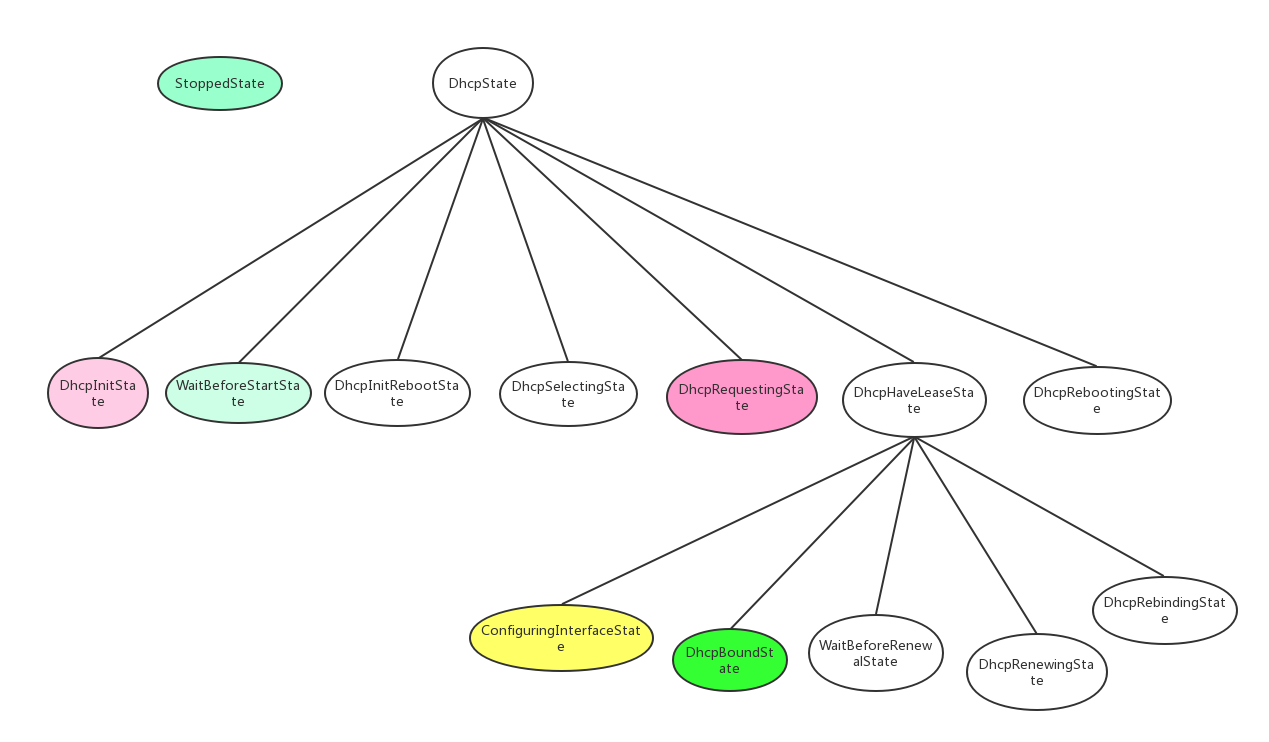


图10-2-1 Dhcpclient状态机

由StoppedState处理CMD\_START\_DHCP,,将状态机切到WaitBeforeStartedState

case CMD\_START\_DHCP:

if (mRegisteredForPreDhcpNotification) {

transitionTo(mWaitBeforeStartState);

}

waitBeforeStartState向ipManager发送CMD\_PRE\_DHCP\_ACTION

#### 10.3ipManager处理CMD\_PRE\_DHCP\_ACTION

ipManager处理CMD\_PRE\_DHCP\_ACTION,向WifiStateMachine发送DhcpClient.CMD\_PRE\_DHCP\_ACTION

IpMangaer处理

case DhcpClient.CMD\_PRE\_DHCP\_ACTION:

if (mConfiguration.mRequestedPreDhcpActionMs > 0) {

ensureDhcpAction();

} else {

sendMessage(EVENT\_PRE\_DHCP\_ACTION\_COMPLETE);

}

break;

private void ensureDhcpAction() {

if (!mDhcpActionInFlight) {

mCallback.onPreDhcpAction();//wifiStateMachine的onPreDhcpAction

mDhcpActionInFlight = true;

final long alarmTime = SystemClock.elapsedRealtime() +

mConfiguration.mRequestedPreDhcpActionMs;

mDhcpActionTimeoutAlarm.schedule(alarmTime);

}

}

public void onPreDhcpAction() {

sendMessage(DhcpClient.CMD\_PRE\_DHCP\_ACTION);

}

#### 10.4WifiStateMachine处理DhcpClient.CMD\_PRE\_DHCP\_ACTION

wifiStateMachine处于obtainingIpState,交给L2ConnectedState处理

handlePreDhcpSetup();向wifiStateMachine发送DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE,wifiStateMachine处理该消息像ipmanager发送EVENT\_PRE\_DHCP\_ACTION\_COMPLETE

1.L2connectedstate处理DhcpClient.CMD\_PRE\_DHCP\_ACTION

case DhcpClient.CMD\_PRE\_DHCP\_ACTION:

handlePreDhcpSetup();

break;

void handlePreDhcpSetup() {

// Disable the coexistence mode

mWifiNative.setBluetoothCoexistenceMode(

mWifiNative.BLUETOOTH\_COEXISTENCE\_MODE\_DISABLED);

setSuspendOptimizationsNative(SUSPEND\_DUE\_TO\_DHCP, false);

mWifiNative.setPowerSave(false);

// Update link layer stats

getWifiLinkLayerStats(false);

/\* P2p discovery breaks dhcp, shut it down in order to get through this \*/

Message msg = new Message();

msg.what = WifiP2pServiceImpl.BLOCK\_DISCOVERY;

msg.arg1 = WifiP2pServiceImpl.ENABLED;

msg.arg2 = DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE;

msg.obj = WifiStateMachine.this;

mWifiP2pChannel.sendMessage(msg);

}

2.l2connectedstate处理DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE;

case DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE:

mIpManager.completedPreDhcpAction();

public void completedPreDhcpAction() {

sendMessage(EVENT\_PRE\_DHCP\_ACTION\_COMPLETE);

}

#### 10.5Ipmanager处理EVENT\_PRE\_DHCP\_ACTION\_COMPLETE

向dhcpClient发送DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE,dhcpClient将状态机切到dhcpInitstate并发送dhcpDiscover

case EVENT\_PRE\_DHCP\_ACTION\_COMPLETE:

if (mDhcpClient != null) {

mDhcpClient.sendMessage(DhcpClient.CMD\_PRE\_DHCP\_ACTION\_COMPLETE);

}

break;

dhcpClient处理CMD\_PRE\_DHCP\_ACTION\_COMPLETE,WaitBeforeStartedState处理,将状态切机到DhcpInitState

case CMD\_PRE\_DHCP\_ACTION\_COMPLETE:

transitionTo(mOtherState);//otherState在构造函数中设置为DhcpInitState了

return HANDLED;

DhcpInitState的enter将发送CMD\_KICK给自己,并处理该消息,发送dhcp discover

DhcpInitState处理CMD\_KICK

case CMD\_KICK:

sendPacket();

scheduleKick();

return HANDLED;

protected boolean sendPacket() {

return sendDiscoverPacket();//发送DHCP Discover

}

private boolean sendDiscoverPacket() {

ByteBuffer packet = DhcpPacket.buildDiscoverPacket(

DhcpPacket.ENCAP\_L2, mTransactionId, getSecs(), mHwAddr,

DO\_UNICAST, REQUESTED\_PARAMS);

return transmitPacket(packet, "DHCPDISCOVER", DhcpPacket.ENCAP\_L2, INADDR\_BROADCAST);

}

private boolean transmitPacket(ByteBuffer buf, String description, int encap, Inet4Address to) {

try {

if (encap == DhcpPacket.ENCAP\_L2)

{

if (DBG) Log.d(TAG, "Broadcasting " + description);

Os.sendto(mPacketSock, buf.array(), 0, buf.limit(), 0, mInterfaceBroadcastAddr);

}

else if (encap == DhcpPacket.ENCAP\_BOOTP && to.equals(INADDR\_BROADCAST))

{…..}

else {….}

} catch(ErrnoException|IOException e) {

Log.e(TAG, "Can't send packet: ", e);

return false;

}

return true;

}

#### 10.6收到offer报文并处理

ReceiveThread发送cmd\_received\_packet

最终由dhcpInitState的receivePacket处理,将状态机切到DhcpRequestingState

protected void receivePacket(DhcpPacket packet) {

if (!isValidPacket(packet)) return;

if (!(packet instanceof DhcpOfferPacket)) return;

mOffer = packet.toDhcpResults();

if (mOffer != null) {

Log.d(TAG, "Got pending lease: " + mOffer);

transitionTo(mDhcpRequestingState);

}

}

DhcpRequestingState的enter,,发送dhcp request报文

public void enter() {

super.enter();

initTimer();

maybeInitTimeout();

sendMessage(CMD\_KICK);

}

protected boolean sendPacket() {

return sendRequestPacket(

INADDR\_ANY, // ciaddr

(Inet4Address) mOffer.ipAddress.getAddress(), // DHCP\_REQUESTED\_IP

(Inet4Address) mOffer.serverAddress, // DHCP\_SERVER\_IDENTIFIER

INADDR\_BROADCAST); // packet destination address

}

private boolean sendRequestPacket(

Inet4Address clientAddress, Inet4Address requestedAddress,

Inet4Address serverAddress, Inet4Address to) {

// TODO: should we use the transaction ID from the server?

final int encap = INADDR\_ANY.equals(clientAddress)

? DhcpPacket.ENCAP\_L2 : DhcpPacket.ENCAP\_BOOTP;

ByteBuffer packet = DhcpPacket.buildRequestPacket(

encap, mTransactionId, getSecs(), clientAddress,

DO\_UNICAST, mHwAddr, requestedAddress,

serverAddress, REQUESTED\_PARAMS, null);

String serverStr = (serverAddress != null) ? serverAddress.getHostAddress() : null;

String description = "DHCPREQUEST ciaddr=" + clientAddress.getHostAddress() +

" request=" + requestedAddress.getHostAddress() +

" serverid=" + serverStr;

return transmitPacket(packet, description, encap, to);

}

#### 10.7收到ack并处理

ReceiveThread发送cmd\_received\_packet

最终由DhcpRequestingState的receivePacket处理,向IpManager发送post\_dhcp\_action,最后将状态机切到ConfiguringInterfaceState

protected void receivePacket(DhcpPacket packet) {

if (!isValidPacket(packet)) return;

if ((packet instanceof DhcpAckPacket)) {

DhcpResults results = packet.toDhcpResults();

if (results != null) {

setDhcpLeaseExpiry(packet);

acceptDhcpResults(results, "Confirmed");//向IpManager发送CMD\_POST\_DHCP\_ACTION

transitionTo(mConfiguringInterfaceState);

}

} else if (packet instanceof DhcpNakPacket) {

// TODO: Wait a while before returning into INIT state.

Log.d(TAG, "Received NAK, returning to INIT");

mOffer = null;

transitionTo(mDhcpInitState);

}

}

IpManager处理CMD\_POST\_DHCP\_ACTION

case DhcpClient.CMD\_POST\_DHCP\_ACTION:

stopDhcpAction();

switch (msg.arg1) {

case DhcpClient.DHCP\_SUCCESS:

handleIPv4Success((DhcpResults) msg.obj);

break;

case DhcpClient.DHCP\_FAILURE:

handleIPv4Failure();

break;

default:

Log.e(mTag, "Unknown CMD\_POST\_DHCP\_ACTION status:" + msg.arg1);

}

break;

private void stopDhcpAction() {

mDhcpActionTimeoutAlarm.cancel();

if (mDhcpActionInFlight) {

mCallback.onPostDhcpAction();

mDhcpActionInFlight = false;

}

}

public void onPostDhcpAction() {

sendMessage(DhcpClient.CMD\_POST\_DHCP\_ACTION);

}

private void handleIPv4Success(DhcpResults dhcpResults) {

mDhcpResults = new DhcpResults(dhcpResults);

final LinkProperties newLp = assembleLinkProperties();

final ProvisioningChange delta = setLinkProperties(newLp);

if (VDBG) {

Log.d(mTag, "onNewDhcpResults(" + Objects.toString(dhcpResults) + ")");

}

mCallback.onNewDhcpResults(dhcpResults);

dispatchCallback(delta, newLp);

}

public void onNewDhcpResults(DhcpResults dhcpResults) {

if (dhcpResults != null) {

sendMessage(CMD\_IPV4\_PROVISIONING\_SUCCESS, dhcpResults);

} else {

sendMessage(CMD\_IPV4\_PROVISIONING\_FAILURE); mWifiLastResortWatchdog.noteConnectionFailureAndTriggerIfNeeded(getTargetSsid(),

mTargetRoamBSSID,

WifiLastResortWatchdog.FAILURE\_CODE\_DHCP);

}

}

private void dispatchCallback(ProvisioningChange delta, LinkProperties newLp) {

switch (delta) {

case GAINED\_PROVISIONING:

if (VDBG) { Log.d(mTag, "onProvisioningSuccess()"); }

recordMetric(IpManagerEvent.PROVISIONING\_OK);

mCallback.onProvisioningSuccess(newLp);

break;

….

}

}

public void onProvisioningSuccess(LinkProperties newLp) {

sendMessage(CMD\_UPDATE\_LINKPROPERTIES, newLp);

sendMessage(CMD\_IP\_CONFIGURATION\_SUCCESSFUL);

}

wifiStateMachine消息队列CMD\_POST\_DHCP\_ACTION , CMD\_IPV4\_PROVISIONING\_SUCCESS , CMD\_UPDATE\_LINKPROPERTIES , CMD\_IP\_CONFIGURATION\_SUCCESSFUL

ConfiguringInterfaceState的Enter函数向ipManager发送CMD\_CONFIGURE\_LINKADDRESS

IpManager的StartedState处理该消息,最后向dhcpclient发送DhcpClient.EVENT\_LINKADDRESS\_CONFIGURED

case DhcpClient.CMD\_CONFIGURE\_LINKADDRESS: {

final LinkAddress ipAddress = (LinkAddress) msg.obj;

if (setIPv4Address(ipAddress)) {

mDhcpClient.sendMessage(DhcpClient.EVENT\_LINKADDRESS\_CONFIGURED);

} else {

……..

}

break;

}

private boolean setIPv4Address(LinkAddress address) {

final InterfaceConfiguration ifcg = new InterfaceConfiguration();

ifcg.setLinkAddress(address);

try {

mNwService.setInterfaceConfig(mInterfaceName, ifcg);

if (VDBG) Log.d(mTag, "IPv4 configuration succeeded");

} catch (IllegalStateException | RemoteException e) {

Log.e(mTag, "IPv4 configuration failed: ", e);

return false;

}

return true;

}

netlinkTracket.callback的update发送EVENT\_NETLINK\_LINKPROPERTIES\_CHANGED给ipmanager(?????????????????????????????)

Dhcpclient处于ConfiguringInterfaceState处理DhcpClient.EVENT\_LINKADDRESS\_CONFIGURED,将状态机切到mDhcpBoundState

case EVENT\_LINKADDRESS\_CONFIGURED:

transitionTo(mDhcpBoundState);

return HANDLED;

#### 10.8WifiStateMachine处理消息

CMD\_POST\_DHCP\_ACTION,CMD\_IPV4\_PROVISIONING\_SUCCESS , CMD\_UPDATE\_LINKPROPERTIES , CMD\_IP\_CONFIGURATION\_SUCCESSFUL

WifiStateMachine处于disconnectedState,

处理CMD\_POST\_DHCP\_ACTION,主要设置省电模式、挂起模式、恢复一些设置

处理CMD\_IPV4\_PROVISIONING\_SUCCESS，主要保存地址

defaultState处理CMD\_UPDATE\_LINKPROPERTIES，更新属性

处理CMD\_IP\_CONFIGURATION\_SUCCESSFUL,,最后将状态机切到connectedState

### 附: WifiStateMachine状态机变换过程

WifiStateMachine: transitionTo: destState=InitialState

WifiStateMachine: transitionTo: destState=SupplicantStartingState

WifiStateMachine: transitionTo: destState=DriverStartedState

WifiStateMachine: transitionTo: destState=DisconnectedState

WifiStateMachine: transitionTo: destState=DisconnectedState

WifiStateMachine: transitionTo: destState=ObtainingIpState

WifiStateMachine: transitionTo: destState=ConnectedState

1. NULL 0,这样才会执行fast\_associate [↑](#footnote-ref-1)
2. Returned 1 [↑](#footnote-ref-2)
3. =1,后面不用看 [↑](#footnote-ref-3)
4. Returns: 0 on success, -1 on failure [↑](#footnote-ref-4)
5. 该值是0,所以try\_oppotunistic=0 [↑](#footnote-ref-5)
6. 失败,, (从pmksa缓存中取出current\_ssid对应的pmkid cache项) [↑](#footnote-ref-6)
7. 生成一个用于关联请求的IE信息, 包括：group\_cipher、pairwise\_cipher、key\_mgmt。信息的选择需要考虑AP的情况 [↑](#footnote-ref-7)
8. 主要是更新RSN/WPA IE信息 [↑](#footnote-ref-8)
9. ft\_complete=0; [↑](#footnote-ref-9)