COAP: A Software-Defined Approach for Managing Residential Wireless Gateways

1. OVERVIEW

Wireless gateways (e.g., Access Points, wireless Set-Top Boxes) act as the primary mode of internet access in most residential network deployments today. A diverse set of WiFi-capable devices access Internet-based services through these gateways, e.g., laptops, tablets and other handhelds, game controllers (XBox, Wii), media streaming devices (Apple TV, Google TV, Roku), and many more. Given its central role in these home networks, the performance and experience of users at homes depend centrally on efficient and dynamic configuration of these gateways.

Using SDNs for residential Wireless Gateways. Most enterprise WiFi solutions today (including some recent SDN-style efforts [7]) adopt a centralized approach (including vendor-specific cloud based solutions [2]) for managing a set of homogeneous Access Points in a uniform and coordinated manner. We believe that a SDN based approach (using an open API) is important in home environments where each wireless neighborhood has a diverse set of these wireless gateways. Unlike enterprise environments, homogeneity in such environments is likely hard to achieve.

In our proposed service, called COAP (Coordination framework for Open APs), participating wireless gateways (e.g., Access Points) are configured to securely connect to a cloud-based controller (Figure 2). The controller provides all necessary management services that can be operated by a third-party (potentially distinct from the individual ISPs). In general, it is desirable that all nearby APs use the same controller service for the management function. In the context of large apartment buildings, we envision that the apartment management contract with a single controller service and all residents are asked to utilize the designated controller service within the building. This service would be no different than many other utilities distributed to residents, e.g., water, electricity, etc., which is arranged by the apartment management. Individual residents can also pick different controller services to realize many of our proposed benefits. *However, some advanced features, e.g., interference management and mitigation, are better served if neighboring gateways participate through the same service.*

The COAP framework extends OpenFlow to provide wireless management capabilities which is complementary to applications that manage residential broadband networks. Furthermore, with the growing demands on in-home wireless networks, especially with the dominance and growth in usage of high bandwidth media streaming, the need for coordination between neighbors will grow.

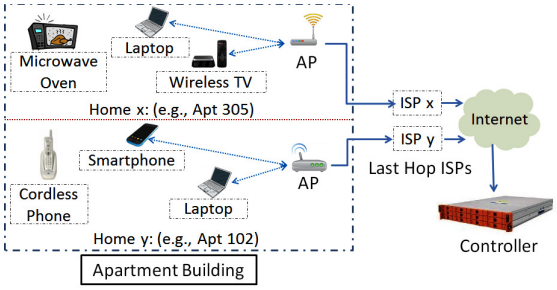

Figure 1: An example of COAP deployment within a residential apartment building consisting of COAP capable home wireless gateways (e.g., Access Points) and a cloud based COAP controller.

2. FRAMEWORK COMPONENTS AND IMPLEMENTATION

To participate in the COAP framework, home gateways need to expose a vendor-neutral open API to communicate with the COAP controller. In this model (Figure 2), gateways need to implement three different modules that expose a different set of functions (related to configuration, diagnostics and statistics collection) to the controller.

2.1 Framework Components

COAP uses OpenFlow to enable SDN style management of residential APs. Following are the main components of the COAP framework.

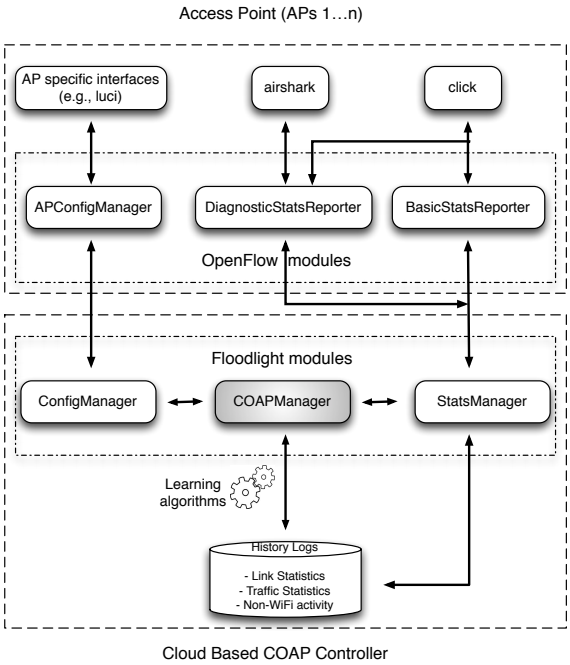

Figure 2: Components of the COAP framework. The different modules implemented by the home gateways (e.g., Access Points) and the controller are shown.

Controller. The COAP controller is implemented over the Java based open source SDN controller, Floodlight and currently runs on a standard linux server for our deployment. Floodlight allows developers to implement use one of existing modules or implement their own modules for this framework. All the COAP controller modules (StatsManager, COAPManager and ConfigManager) are implemented as modules within Floodlight. The controller uses OpenFlow with COAP-specific protocol extensions to collect data from the gateways as well as apply the configuration updates.

Protocol extensions for OpenFlow. The OpenFlow communication protocol currently consists of capabilities to exchange switch related statistics (e.g., statistics per switch port, flow, queue etc.). We augment this feature to also exchange different COAP related wireless specific statistics (e.g., airtime usage, link performance, non-WiFi activity). To transmit wireless configuration updates to gateways (e.g., switch channel, throttle airtime), we extend the OpenFlow protocol to use a mechanism analogous to the one used to send switch configuration updates (e.g., adding a flow-related rule). We discuss the protocol details in §3.

Wireless Gateways. In our current implementation using OpenWrt based WiFi gateways, we use the open-source click [1] framework to implement the WiFi and non-WiFi statistics gathering capabilities of the *BasicStatsReporter* and *Diagnostic-StatsReporter* modules. Airshark [5] provides the non-WiFi device detection capabilities using commodity WiFi cards. The OpenFlow module interfaces with Airshark and click as well as communicates with the SDN controller.

We have instrumented the ath9k wireless driver to support APIs related to airtime management. For example, to throttle the airtime access of a gateway, we disable the transmit queue (using the AR_Q_TXD register) to block packet transmissions for the required duration. Across vendors, the underlying implementation of this feature can be driver specific and transparent to the COAP API.

The *APConfigManager* module interfaces with the OpenWrt configuration tool (*luci*) to perform AP level configurations (e.g., channel, transmit power and traffic shaping). This interface can easily be modified for other platforms.

2.2 COAP API

Table 1 describes the different capabilities implemented within the COAP modules. They allow the OpenFlow based COAP controller to learn about the wireless activity as well as provides a set of key "hooks" to configure and manage the APs. Following is a brief overview of the different modules implemented at the COAP APs and controller.

APConfigManager. It allows the controller to configure basic AP parameters such as the operating channel and transmit power levels. Beyond this basic control, the AP provides an API to remotely manage its airtime access. For example, by slotting time into fine grained intervals (e.g., 10 ms periods), the controller can manage an AP's airtime access to reduce channel contention for important flows and mitigate receiver-side interference. The *ConfigManager* module at the controller communicates these

Function	Description
APConfigManager: Module to receive configuration commands from the controller and execute them at the AP	
SetParameters(channel, power)	Configures the AP to a particular channel and/or transmit power.
SetAirtimeAccess(slotDuration, transmitBitmap)	Manage airtime access of APs by throttling/slotting transmissions.
BasicStatsReporter: Module to report aggregate wireless statistics to the controller	
GetNeighborInfo()	Scans all WiFi channels for neighboring APs' beacons, gets MAC address (hashed) and RSSI.
GetAirtimeUtilization()	Get the current channel's airtime utilization (0 - 100%) over the most recent time epoch.
GetClientInfo()	Get information about associated clients: e.g., MAC address (hashed), device type.
GetLocalLinkStatistics()	Get packet transmission statistics per local link: e.g., signal strength and total packets sent, received, retried.
GetTrafficInfo()	Get statistics about current traffic: e.g., source id, type, packet count, bytes sent.
DiagnosticStatsReporter: Module to report more detailed wireless statistics for diagnosis purposes	
GetNonWifiDevices()	Get neighboring non-WiFi activity (using Airshark [5]): e.g., device type, duration etc.
GetPacketSummaries()	Get fine grained MAC layer packet transmission reports summaries [3] for overheard links. Each packet summary contains: timestamp, packet length, PHY rate, retry bit and RSSI.
GetSyncBeacons()	Get overheard beacon information (receive. timestamp, MAC sequence number, hashed MAC id) on the same WiFi channel for time synchronization.

Table 1: Main functions implemented by the COAP APs. Most functions are fairly simple to implement. We are planning to release the API specification and reference implementation: http://research.cs.wisc.edu/wings/projects/coap/.

configuration updates from the controller to the APs (e.g., channel switch, transmit power, airtime management settings).

BasicStatsReporter and DiagnosticStatsReporter. This module's goal is to collect different statistics about the local wireless as well as neighboring wireless activity. These statistics are gathered by the controller (*StatsManager* module) from the individual APs as well as logged into a database. Similarly, the *DiagnosticStatsReporter* module is implemented at the APs to collect additional debugging information such as neighboring non-WiFi activity and fine grained statistics about link activity (e.g., to determine interference [3, 6]).

COAPManager. All server modules and tasks are managed by the *COAPManager* module, through which it manages and configures the connected APs. In the COAP framework, the implementation of the COAPManager is specific to service providers which allow them to implement custom policies based on the requirements. The server also maintains logs about previous wireless activity (WiFi and non-WiFi) to learn and predict wireless usage characteristics near each AP's location. This *context* related information can be useful for pre-emptive configuration of home APs to mitigate wireless problems, such as interference and channel congestion.

```
enum ofp_stats_types {
  OFPST_DESC,
  OFPST_FLOW,
  OFPST_AGGREGATE,
  OFPST_TABLE,
  OFPST_PORT,
  OFPST_QUEUE,

  .
  .
  .

  // Added to support COAP messages
  OFPST_APINFO,       /* AP information */
  OFPST_UTIL,         /* Airtime Utilization */
  OFPST_STATION,      /* Station statistics */
  OFPST_SYNC_BEACON,  /* Overheard beacon information */
  OFPST_NONWIFI,      /* Observed non-WiFi devices */
  OFPST_BEACON,       /* Beacon statistics */
  OFPST_PASSIVE,      /* Overheard WiFi activity */
  OFPST_TRAFFICINFO,  /* Traffic type information */
  // End

  OFPST_VENDOR = 0xffff
};
```

Figure 3: Adding COAP related types to *ofp_stats_types* for receiving various wireless related statistics from the COAP APs.

3. OPENFLOW PROTOCOL MESSAGES

```
enum ofp_type {
  /* Immutable messages. */
  OFPT_HELLO,               /* Symmetric message */
  OFPT_ERROR,               /* Symmetric message */
  OFPT_ECHO_REQUEST,        /* Symmetric message */
  OFPT_ECHO_REPLY,          /* Symmetric message */
  OFPT_VENDOR,              /* Symmetric message */

  .
  .
  .

  // For COAP: Added to manage wireless configuration at the APs
  OFPT_SET_WIRELESS_CONFIG  /* Controller/switch message */
};
```

Figure 4: Adding the *OFPT_SET_WIRELESS_CONFIG* message type to receive wireless configuration updates from the controller.

In this section, we provide details about the extensions to the OpenFlow spec to add COAP related features. We have used this implementation to understand wireless performance in dense residential deployments [3] and employ strategies to improve performance [4].

3.1 Message Types

1. To collect COAP based wireless statistics (Table 1), new types are added to the *enum ofp_stats_types*. Figure 3 shows these new types. In §3.2, we further discuss the details of the structures used to collect these statistics from the COAP APs.
2. To transmit wireless configuration updates from the controller to the COAP APs, a new message type (*OFPST_SET_WIRELESS_CONFIG*) is added to *ofp_type* (Figure 4). For example, to set an AP's channel, the COAP controller uses the this type to send a channel update message to the AP.

3.2 COAP related statistics

The section describes the OpenFlow structures to collect COAP related statistics described in Table 1.

1. The *ofp_apinfo_stats* structure (figure 5) provides information about each wireless NIC present on the Access Point.

```
/* AP information statistics. */
struct ofp_apinfo_stats {
  char apmacid[MACID_LENGTH];     /* AP MAC ID */
  uint32_t timestamp;             /* Unix timestamp */

  uint8_t nic_count;              /* NIC number */
  uint8_t is_2GHz_supported;      /* 2 GHz support */
  uint8_t is_5GHz_supported;      /* 5 GHz support */

  uint8_t is_2GHz_ht20_support;   /* HT20 on 2GHz? */
  uint8_t is_2GHz_ht40_support;   /* HT40 on 2GHz? */
  uint8_t is_5GHz_ht20_support;   /* HT20 on 5GHz? */
  uint8_t is_5GHz_ht40_support;   /* HT40 on 5GHz? */

  uint8_t pad;                    /* Align to 32 bits */
};
OFP_ASSERT(sizeof(struct ofp_apinfo_stats) == 32);
```

Figure 5: OpenFlow structure used by a COAP AP to report airtime utilization information to the controller.

2. The *ofp_util_stats* structure (figure 6) is used to collect parameters related to airtime utilization on the AP's current channel.
3. The *ofp_beacon_stats* structure (figure 7) is used to obtain information about neighboring APs located in the vicinity of the COAPAP. The specific AP performs a scan on WiFi supported WiFi channel to detect neighboring APs' beacons.
4. The *ofp_station_stats* structure (figure 8) reports a COAP AP's download packet transmission statistics to a local WiFi client. The client is identified by a unique ID (*client_macid*) which can either be the client MAC address or a hashed version of the MAC address for privacy purposes. The *retry_string* is a formatted string containing the packets transmitted and retried by the AP at each individual PHY rate.
5. The *ofp_syncbeacon_stats* structure (figure 9) reports beacons from other APs overheard by a COAP AP. The *timestamp* field records a 32 bit timestamp for when the beacon was observed. This data is collected from nearby COAP APs by the COAP controller to obtain their relative clock offsets. The clock offsets are used to synchronize the COAP APs' clock for the *SetAirtimeAccess* API (Table 1).

```
/* Airtime utilization statistics. */
struct ofp_util_stats {
  uint16_t type;          /* util or util hop */
  short util_val;         /* Utilization value. */
  uint32_t active;        /* Active period (in seconds) */
  uint32_t busy;          /* Busy period (in seconds) */
  uint32_t receive;       /* Receive period (in seconds) */
  uint32_t xmit;          /* Transmit period (in seconds) */
  uint32_t channel;       /* WiFi Channel */
  uint32_t timestamp;     /* Unix timestamp */
  int noise_floor;        /* Noise floor (in -dBm) */
};
OFP_ASSERT(sizeof(struct ofp_util_stats) == 32);
```

Figure 6: OpenFlow structure used by a COAP AP to report airtime utilization information to the controller.

```
/* Neighboring AP information */
struct ofp_beacon_stats {
  char apmac[MACID_LENGTH]; /* AP MAC ID */
  uint32_t timestamp;       /* Unix timestamp */
  int rssi;                 /* Signal strength */
  uint16_t channel;         /* Channel */
  uint8_t pad[2];           /* Align to 32-bits. */
};
OFP_ASSERT(sizeof(struct ofp_beacon_stats) == 32);
```

Figure 7: OpenFlow structure used by a COAP AP to report neighboring AP information (observed through beacons) to the controller.

6. The *ofp_nonwifi_stats* structure (figure 10) provides information about nearby non-WiFi activity (e.g., cordless phones, microwave ovens) observed by COAP APs. The information can be helpful to build applications that detect and mitigate non-WiFi interference using commodity WiFi APs. In our implementation, we use Airshark [5] to provide this information.
7. The *ofp_client_stats* structure (figure 11) provides client related metadata such the mac_id (mac address or hashed mac address), host_name and optional (device_type). This information can used by the COAP controller to determine the client "context". Such information can be potentially used for predicting future client activity based on prior history (e.g., watching long running video streams) and perform adaptation at neighboring APs to reduce channel contention.
8. The *ofp_passive_stats* structure (figure 12) is used to report external WiFi activity to the controller. Each instance provides aggregate packet transmission statistics for a single external WiFi link (represented by the *sender, receiver* fields). This is useful for knowing the WiFi activity of APs, which are not a part of the COAP AP framework.
9. The *ofp_trafficinfo_stats* structure (figure 13) provides information about the traffic characteristics to the controller as a formatted string. The controller can leverage this information to predict future traffic characteristics and perform adaptations at nearby COAP APs to load balance the traffic across different channels.
10. The *ofp_diagnostic_stats* structure (figure 14) provides a formatted string containing packet level summaries for diagnostic purposes; such as detecting the presence of hidden terminal interference. The per-packet information only consists of wireless transmission related parameters (e.g., PHY rates, MAC timestamp, packet retries, length etc.). During periods of poor WiFi performance, the COAP controller can collect this information to detect the causes of high interference scenarios to apply the required mitigation strategies.

```
/* Per-client packet transmission summary */
struct ofp_station_stats {
  char client_macid[MACID_LENGTH];    /* Client MAC ID */
  char retry_string[RATESTR_LENGTH];  /* PHY rate stats */

  uint32_t packet_count;              /* Packets sent */
  uint32_t packet_retries;            /* Packets retried */

  int rssi;                           /* RSSI */
  // uint32_t channel;                /* WiFi channel */
  uint32_t timestamp;                 /* Unix timestamp */
};
OFP_ASSERT(sizeof(struct ofp_station_stats) == 196);
```

Figure 8: OpenFlow structure used by a COAP AP to report per-station download packet transmission statistics to the controller.

```
/* Beacon information - For time synchronization */
struct ofp_syncbeacon_stats {
  char apmac[MACID_LENGTH]; /* Access Point MAC ID */
  uint32_t timestamp;       /* Timestamp value */
  uint16_t seq;             /* Sequence number.*/
  uint16_t channel;         /* Channel */
};
OFP_ASSERT(sizeof(struct ofp_syncbeacon_stats) == 28);
```

Figure 9: OpenFlow structure used by a COAP AP to report timestamped overheard beacons to the controller. The controller used this information for inter-AP time synchronization.

```
/* Non-WiFi device activity */
struct ofp_nonwifi_stats {
  uint32_t timestamp;    /* Unix timestamp for the record */
  uint32_t start_ts;     /* Unix timestamp when
                         activity is first detected */
  uint32_t end_ts;       /* Unix timestamp when
                         activity is last detected */

  uint32_t duration_sec; /* Activity duration in seconds */
  int rssi;              /* Observed signal strength */

  uint16_t start_bin;    /* Starting WiFi subcarrier (0-55) */
  uint16_t center_bin;   /* Peak RSSI subcarrier */
  uint16_t end_bin;      /* Ending WiFi subcarrier */

  uint16_t device_type;  /* Device type (e.g., FHSS phone) */

  uint32_t channel;      /* WiFi channel with activity */
};
OFP_ASSERT(sizeof(struct ofp_nonwifi_stats) == 32);
```

Figure 10: OpenFlow structure used by a COAP AP to report non-WiFi device activity to the controller.

```
/* Meta data about the associated clients */
struct ofp_client_stats {
  char mac_id[MACID_LENGTH];   /* Client MAC ID */
  char host_name[NAME_LENGTH]; /* Client's hostname */
  uint32_t timestamp;           /* Unix timestamp */
  long apip;                    /* Local IP Address */
  uint32_t dev_type;            /* Device type */
};
OFP_ASSERT(sizeof(struct ofp_client_stats) == 72);
```

Figure 11: OpenFlow structure used by a COAP AP to report client metadata to the controller.

```
/* Passively collected neighboring WiFi link statistics */
struct ofp_passive_stats {
  char sender[MACID_LENGTH];         /* Sender's MAC ID */
  char receiver[MACID_LENGTH];       /* Receiver MAC ID */
  char retry_string[RATESTR_LENGTH]; /* PHY rate stats */

  uint16_t average_packet_length; /* Average packet length */
  uint16_t average_rate_times10;  /* Average PHY rate (x10) */

  int rssi;                       /* Signal strength */
  uint32_t packet_count;          /* Packets transmitted */
  uint32_t packet_retries;        /* Packets retried */
  uint32_t channel;               /* WiFi channel */
  uint32_t timestamp;             /* Unix timestamp */
};
OFP_ASSERT(sizeof(struct ofp_passive_stats) == 224);
```

Figure 12: OpenFlow structure used by a COAP AP to report observed external WiFi link statistics to the controller.

```
  /* Report statistics about the traffic type */
  struct ofp_trafficinfo_stats {
      char trafficinfo_string[TRAFFICINFO_STRING_LENGTH];
  };
  OFP_ASSERT(sizeof(struct ofp_trafficinfo_stats) == 1024);
```

Figure 13: OpenFlow structure used by a COAP AP to report traffic type statistics to the controller.

```
/* Report statistics about the traffic type */
struct ofp_diagnostic_stats {
    char diagnosticstats_string[DIAGNOSTICSTATS_STRING_LENGTH];
};
OFP_ASSERT(sizeof(struct ofp_diagnostic_stats) == 30720);
```

Figure 14: OpenFlow structure used by a COAP AP to report packet level summaries to the controller.

4. REFERENCES

[1] Click modular router. http://www.read.cs.ucla.edu/click/click.

[2] Meraki. Enterprise cloud management. http://www.meraki.com/products/wireless/enterprise-cloud-management.

[3] A. Patro, S. Govindan, and S. Banerjee. Observing Home Wireless Experience Through WiFi APs. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13.

[4] A. Patro, S. Govindan, and S. Banerjee. Outsourcing Home AP Management to the Cloud through an Open API. Open Networking Summit 2013.

[5] S. Rayanchu, A. Patro, and S. Banerjee. Airshark: Detecting non-WiFi RF devices using commodity WiFi hardware. IMC '11.

[6] V. Shrivastava, S. Rayanchu, S. Banerjee, and K. Papagiannaki. PIE in the sky: online passive interference estimation for enterprise WLANs. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11.

[7] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao. Towards programmable enterprise WLANs with Odin. HotSDN '12.