Dyson: An Architecture for Extensible Wireless LANs

Rohan Murty[†], Jitendra Padhye, Alec Wolman, Matt Welsh[†]
Microsoft Research, [†]Harvard University

Abstract

Dyson is a new software architecture for building customizable WLANs. While research in wireless networks has made great strides, these advancements have not seen the light of day in real WLAN deployments. One of the key reasons is that today's WLANs are not architected to embrace change. For example, system administrators cannot fine-tune the association policy for their particular environment: an administrator may know certain nodes in certain locations interfere with each other and cause a severe degradation in throughput, and hence, such associations must be avoided in the particular deployment. Dyson defines a set of APIs that allow clients and APs to send pertinent information such as radio channel conditions to a central controller. The central controller processes this information, to form a global view of the network. This global view, combined with historical information about spatial and temporal usage patterns, allows the central controller enact a rich set of policies to control the network's behavior. Dyson provides a Python-based scripting API that allows the central controller's policies to be extended for site-specific customizations and new optimizations that leverage historical knowledge. We have built a prototype implementation of Dyson, which currently runs on a 28-node testbed distributed across one floor of a typical academic building. Using this testbed, we examine various aspects of the architecture in detail, and demonstrate the ease of implementing a wide range of policies. Using Dyson, we demonstrate optimizing associations, handling VoIP clients, reserving airtime for specific users, and optimizing handoffs for mobile clients.

1 Introduction

Wireless networks are struggling to keep up with the demands of new applications, such as media streaming, voice over IP, and the increasing use of mobile devices, such as Wi-Fi enabled smartphones. Researchers have proposed numerous ways to cope with these changing demands, including new approaches to association and handoff control [22], channel allocation strategies [20, 17], and centralized packet transmission scheduling [26]. However, deploying these innovations in real wireless LANs remains a significant challenge. Enterprises wishing to roll out new applications, services, or policies in a wireless LAN are faced with ossified standards and a wide variety of software, device driver, and hardware implementations of these standards by many different vendors. Compounding this problem is the fact that existing WLAN standards generally do not allow for much customization. In this paper, we argue that it is time to rethink the architecture of wireless networks from the ground up, to enable greater observability, control, and extensibility to meet future needs.

Today, WLAN vendors offer few knobs to customize network operation. A case in point is the Microsoft campus enterprise wireless network, which uses access points supplied a single vendor. These access points are managed by a central controller, which attempts to dynamically tune the assignments for channel selection and transmit power to improve performance. Shortly after these new APs were deployed, the WLAN administrators realized that the transmit power control algorithm was not suitable for our campus. Because the algorithm was geared towards avoiding interference, many APs reduced their transmit power to such an extent that it left large holes in coverage. The controller offered no knobs to allow administrators to customize the power assignment algorithm; the only option was to disable it entirely.

In the above example, the lack of flexibility does not arise from the 802.11 standard, which is in fact quite flexible in many respects: it imposes no specific policies on association, channel assignment, or power control. The problem is that there is no agreed-upon framework to control these knobs. Moreover, there are no explicit mechanisms for stations to coordinate with each other to observe the state of the network, requiring nodes to take a purely local "every station for itself" point-of-view. This local viewpoint then encourages vendors to hard-code important algorithms into device drivers and firmware that affect WLAN performance, such as those that control AP associations, PHY data rates, and transmission power.

In this paper, we present a new WLAN architecture, called *Dyson*, that is designed to enable extensive customization and control over many aspects of network operation and performance. The Dyson architecture is designed to support *global network observation*, *deep control*, and *extensibility* to meet future needs. In Dyson, both clients and access points coordinate with the network infrastructure to provide detailed measurements on location, radio channel conditions, connectivity, and observed performance. Measurements are stored in a persistent database, allowing the infrastructure to adapt its

behavior based on historical knowledge of network state. Dyson defines a set of APIs that allow clients and APs to share pertinent performance information such as packet loss rates and radio channel conditions. Dyson also defines a control interface, supported by both clients and APs, that permits the infrastructure to manage many aspects of their operation, including associations, channel selection, PHY rate, and transmission throttling. The central controller can enact a rich set of policies to control the WLAN behavior. These polices are built, customized and extended using a Python-based scripting API. Even though Dyson benefits when all clients in the WLAN support the control interface, it can still work with a mix of legacy and controllable clients.

Dyson's design hinges on the use of a centralized network controller, a common feature in recent research [22, 26, 11] and commercial [1, 2] WLANs for enterprises. None of the previous centralized WLAN systems have considered flexibility and evolvability as their primary design requirement. Dyson focuses on providing a rich set of APIs that allow the network's operation to be extended to embrace new application demands and site-specific customizations. Dyson's Python programming interface makes it easy to develop new policies and experiment with a wide range of behaviors. To demonstrate Dyson's flexibilty, we have implemented a range of policies for optimizing associations, handling VoIP clients, and reserving airtime for specific users.

This paper makes the following contributions. First, Dyson is the first wireless LAN architecture that directly addresses the need for extensibility and evolvability, leveraging both centralized control and client-side instrumentation to enable a wide range of new policies to be layered on the existing network. Second, Dyson enables more efficient use of radio spectrum by taking measurements gathered from both clients and APs into account. Third, we have implemented Dyson on a 28-node testbed distributed over one floor of an office building, which we use to evaluate the system in detail using a range of policies. We demonstrate how WLAN behavior can be easily customized via Dyson's policies to provide better performance overall.

2 Dyson Architecture

The Dyson network architecture, shown in Figure 2, consists of a number of wireless *clients*, *access points* (APs), and a single *central controller* (CC). As described below, both APs and Dyson-enabled clients report measurements to the infrastructure, which are used to construct a dynamic *network map* representing the state of the network. Measurements are also logged to a database for historical analysis, and static information on AP location and MAC addresses are stored in a separate AP database. Extensibility is enabled through *policies* that are used to trigger network configuration changes via *commands* delivered by the central controller to APs and clients.

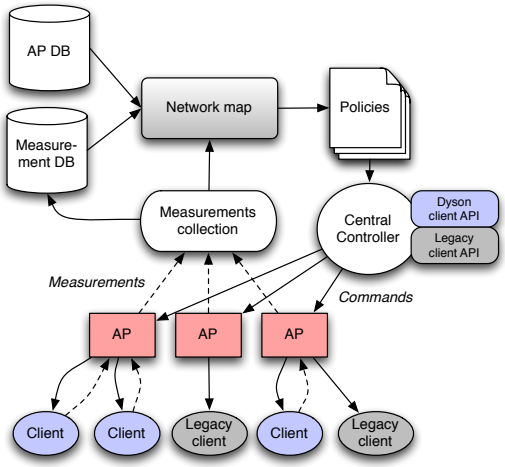


Figure 1: **The Dyson network architecture.**

Dyson builds upon existing 802.11 standards, including CSMA MAC and the format of the data and management frames. As a result, Dyson can be implemented entirely using existing 802.11-compatible hardware. The key difference between Dyson and existing enterprise WLANs is the manner in which network management and control is performed. The Dyson architecture requires APs to be Dyson-aware. Dyson-aware clients support enhanced functionality for measurement collection and control, as described below. Legacy 802.11 clients can be supported by Dyson, although with reduced functionality. Note that Dyson-enabled clients remain compatible with legacy 802.11 networks.

The use of a central controller in enterprise WLANs is widespread.[1] For example, in Aruba [1] networks, the CC is responsible for assigning radio channels and transmission power levels to individual APs based on global observation of the network traffic. Dyson significantly augments this design by extending both observation and control to the wireless clients as well as the APs. Dyson clients are responsible for collecting periodic *measurements* of channel and traffic conditions and reporting them to the CC, as well as responding to *commands* from the CC that control many aspects of transmission parameters, as described below.

A key question that arises in this regime is how much control clients should yield to the infrastructure. At one extreme, the CC could control clients at a very fine-grained level, for example, by dictating individual packet transmission timings [26]. However, this design would require substantial control overhead, and would fail to re-

[1]Note that the CC need not be physically centralized, as this functionality can be replicated across multiple physical hosts for reliability and scalability.

spond rapidly to local changes in channel conditions (e.g., interference) at the client. In Dyson, we opt to affect control at a higher level, via channel allocations, client-AP associations and throttling. Although cruder than packet-level control, this design strikes a balance between the overhead for command issue and the ability of the network to drive towards more efficient configurations.

One implication of this design is that we assume that Dyson clients are willing participants in the system, and are capable of accurately and truthfully responding to measurement requests and commands. There is, of course, the potential that malicious or buggy clients could misbehave and degrade network performance. However, we argue that the degree of trust that Dyson places in clients is not substantially greater than that in conventional 802.11 networks, in which it must be assumed that clients correctly obey the protocol. We assume that Dyson clients are authenticated using 802.1x.

The power of the CC is derived from its global knowledge of the state of the network and ability to control both APs and clients at fine granularity. It also maintains a database to store received measurements, permitting long-term historical analysis of network performance.

A key benefit in Dyson is the ability to collect *client-side* measurements, providing the CC with greater visibility and control over the network state. Client-side information can be used to resolve sources of ambiguity that would arise with AP-only observations. Examples include detection of hidden terminals, awareness of mutual connectivity between APs and clients, and mapping channel airtime utilization. While client participation has been explored by several previous systems [11, 6], Dyson provides a flexible framework in which a wide range of policies can be specified programmatically.

2.1 Measurement collection

In Dyson, both clients and APs are responsible for collecting passive measurements on the state of the network, reporting measurements to the CC, and responding to commands issued by the CC to modify local parameters. As described above, the granularity of measurements and commands is chosen to avoid high overheads for client/CC interactions, but still yield adequate control over client behavior by the infrastructure.

Measurement collection in Dyson supports network-wide optimizations based on both AP and client-side knowledge of the network state. This provides the CC with global information on various factors that affect client performance, such as traffic patterns, interference, hidden terminals, and congestion. This approach obviates the need for a separate wireless monitoring infrastructure [13, 7].

Each client and AP in the system records a set of statistics, summarized in Table 2.1. For each received packet, a set of counters are incremented to track the total number of packets, total packet size, total airtime utilization, and other measures. Dividing counters by the number of received packets can be used to calculate mean values over a measurement window. Clients maintain a single set of these counters, whereas the AP maintains these counters on a per-associated-client basis, allowing measurements to be collected for each separate uplink. In addition to the these statistics, nodes also record the mean airtime utilization (reported by the radio hardware) of the radio channel.

Measurement	Description
`numPackets`	*Number of pkts received*
`totalBytes`	*Total bytes received*
`totalRSSI`	*Total RSSI of received pkts*
`connectivity[]`	*List of tuples* $\langle srcmac, numPkts, totalRSSI \rangle$
`packetsPerPhyRate[]`	*One counter for each PHY rate*
`totalAirtime`	*Airtime used by packets (size × PHY rate)*
`numTxFailures`	*Number of Tx failures*
`numRetransmissions`	*Number of ARQ retransmissions*
`airtimeUtil`	*Channel airtime utilization*

Table 1: **Measurements collected by Dyson nodes.**

APs periodically query their associated clients to collect their measurements, after which clients reset their counters. The AP then pushes the collected client measurements, as well as its own, to the CC. The AP's measurement collection period can be adjusted by the CC to tradeoff reporting latency and measurement traffic overhead. Our measurements in Section 4.8 show that for moderate-sized networks, this overhead is less than 1%.

2.2 Network map

The central controller uses collected measurements to maintain a *network map* representing the global state of the Dyson network. The network map is the key data structure accessed by Dyson's policies (Section 4) in order to drive reconfiguration. The network map is updated each time new measurements are pushed to the CC by an AP. Policies can read the complete network map and push new information into the network map. This allows individual policies to augment the global state maintained by the CC, as well as enabling policies to be composed.

The map consists of several components:

Node location: A table of the physical location of each AP and client in the system, indexed by MAC address. AP locations are static, whereas client locations are computed using the algorithm described in [12]. This information can be used for determining the physical location of network hotspots, and by policies that consider client mobility.

Connectivity: A directed connectivity graph is main-

`SetRate(`r`)`	Set PHY rate
`SetChannel(`c`)`	Set channel
`SetTxLevel(`t`)`	Set transmission power level
`SetCCAThresh(`t`)`	Set CCA threshold
`SetPriority(`p`)`	Set 802.11e priority
`Throttle(`r`)`	Throttle outgoing traffic at the specified rate r
`Handoff` $(c, ap, chan)$	Handoff client c to AP ap on channel $chan$
`AcceptClient` (c)	Associate AP with client c
`EjectClient` (c)	Disassociate client c

Table 2: **The Dyson command API. Commands in bold are applicable to APs only.**

tained, where vertices represent nodes (clients or APs) and edges represent the ability of one node to overhear packets of another node. For each unique MAC address that a node overhears during a measurement interval, the mean RSSI value of packets from that MAC address are reported to the CC. The connectivity graph contains a directed edge for each pair of MAC addresses. While clients are only capable of reporting links on their current channel, APs can use a secondary radio to perform background scanning and report observed connectivity on every channel. An edge is removed from the graph if no packets are observed on the link for 30 seconds. The connectivity graph is used in client-AP associations, detecting hidden terminals, and managing handoffs.

Airtime utilization: Each node measures the airtime utilization of the radio channel in its vicinity. The network map includes a hash table mapping a node's MAC address and channel number to its airtime utilization estimate. This information can be used by a wide range of policies to detect congestion, balance uplink and downlink fairness, and optimize client/AP associations. APs can measure airtime on every channel using the secondary scanning radio.

Historical measurements: Collected measurements are also stored in a persistent database, permitting policies to make use of historical information when making decisions about network reconfiguration. As an example, a policy may wish to consider the historical interference pattern between two APs, or variance in the network congestion at different hours of the day, when driving network reconfigurations.

The network map serves primarily as input to the various policies for driving network configurations. However, it can also serve an auxiliary role to assist a network administrator in understanding AP coverage and sources of performance degradation. For example, visualizing the airtime utilization graph as well as the associated client and AP locations can provide real-time information on network hotspots.

2.3 Central controller

The central controller is responsible for managing the entire Dyson network based on collected measurements from clients and APs. Its job is to apply administrator-defined *policies* to the current network map, and issue *commands* to set parameters of clients and APs according to the policy decisions.

The Dyson command API is shown in Table 2. These commands are intended to provide a rich set of knobs for controlling the network's operation while limiting overheads for command issue. Commands set parameters such as the transmission power level, CCA threshold, 802.11e priority levels, and PHY data rate. The `Handoff`, `AcceptClient`, and `EjectClient` commands control client-AP associations, as described in the next section. Note that clients do not decide themselves which AP to associate with; this is under the control of the Dyson infrastructure.

The CC sends commands to APs directly. Commands to clients are relayed via the AP that the client is currently associated with; in this way the client need not be aware of the CC's identity, and the CC's functionality can be decentralized. Commands are exchanged using MAC-layer control messages which are ACKed by the receiving node. For AP-client commands, ARQ is used to ensure commands are delivered reliably.

Support for legacy clients: Dyson can support legacy 802.11 clients without the extensions described above. Of course, this implies reduced functionality as it is not possible to directly obtain client-side measurements, nor control many aspects of client operation. The CC is able to control client-AP associations for legacy clients, giving the infrastructure control over at least which APs and channels those clients occupy. AP-side measurements can account for any associated legacy clients allowing the system to have visibility into the impact of legacy client traffic. Dyson policies use a reduced control API (that only contains the relevant calls) to interact with legacy clients.

2.4 Policy Engine

Dyson's architecture is designed to support extensibility, composability, and separation of concerns, in order to tune network performance as well as impose site- and client-specific policies. Each policy is encapsulated in a software module that runs on the CC, takes the network map as input, and issues commands to APs and clients as output. As described above, policies can also update and augment the network map itself.

Dyson has a predefined set of policy modules providing commonly-used functionality, but it is possible for new policies to be implemented and loaded into the central controller as needed. Policies are implemented in Python

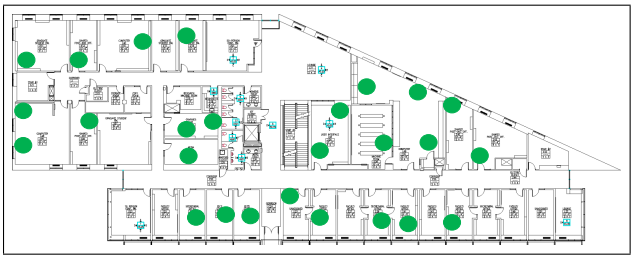


Figure 2: **Dyson testbed deployment**

and are relatively easy to write, as we will show below. This approach enables network designers to update the policies used by a Dyson network installation over time in response to new demands or shifting priorities. We also envision third parties developing new policies for Dyson that can be readily plugged into an existing deployment.

In our current design, policy composition and dependencies must be handled manually by policy designers. There is nothing to prevent two policies from "competing" (say, by issuing conflicting commands in response to the same event in the network); each policy should clearly document its own behavior to avoid unexpected results.

Each policy runs as a separate thread on the CC and is responsible for its own scheduling. Typically, a policy will run with some predefined period, but a policy can also trigger execution on some condition being met (for example, an update to some element in the network map). Standard thread synchronization primitives can be used to implement more sophisticated cross-policy interactions.

In Section 4, we demonstrate a set of policies that highlight different aspects of Dyson's global network visibility and deep control over both APs and clients.

3 Implementation and Testbed

We have implemented a prototype of the Dyson architecture using the ALIX 2c2 single-board computer (500 MHz AMD Geode processor with 256 MB DRAM) running FreeBSD 7, coupled with dual CM 9 Atheros-based 802.11a/b/g radios. Each node can act as either a Dyson client or an AP; only APs make use of the second radio for collecting channel utilization measurements.

We have deployed a testbed of 28 nodes across one floor of an academic office building, as shown in Figure 2. Each node is connected to an Ethernet network for control. The central controller is implemented on a separate machine running FreeBSD with 2 GB of RAM. All experiments presented in this paper use 802.11a to avoid interference with existing 802.11b/g networks in the building.

To support Dyson, we modified the FreeBSD Atheros driver to add support for statistics collection and the Dyson command API, as well as to disable local rate adaptation. Each node runs a Python-based daemon that exposes the Dyson measurements and command API via an XML-RPC interface, and communicates with the modified Atheros driver through *ioctl* calls. The central controller is also implemented in Python; policies are loaded as Python modules at startup time.

The commands listed in Table 2 were implemented via modifications to the Atheros driver. Most of the commands (such as `SetTxLevel`, `SetChannel`, and so forth) simply set driver parameters. `Handoff` informs a client to switch channels and associate with the specified AP. This eliminates the need for scanning, provided the destination AP is still on the specified channel. The `Throttle` command makes use of *dummynet*, a FreeBSD traffic shaping tool, to limit the rate of outgoing traffic. `Throttle` simply sets the dummynet outgoing bandwidth limit on the radio interface to the specific rate.

4 Policies and Evaluation

The primary goal of this section is to demonstrate that the *extensibility* afforded by the Dyson architecture is both desirable and feasible. To do so, we focus on five interrelated issues.

First, we show that the Dyson architecture enables interesting, non-trivial customizations of WLAN deployment that either improve performance, or enable new features. This is our key contribution.

Second, we show that the customizations are easy to realize. Unless Dyson makes it easy to customize WLANs, the fact that it enables interesting customizations is of little practical value. Also, demonstrating the ease of customization reaffirms our programming model, and validates our contention that our chosen API provides the right level of control for our purposes.

To address these two issues, we demonstrate a set of four policies that customize WLAN deployments in a variety of ways. We show how these policies can be realized via simple Python scripts, and illustrate how our APIs provide the right level of abstraction to achieve this.

Third, we discuss how multiple policies co-exist within the Dyson framework. We will show how Dyson allows different policies to be run in different parts of the network. Dyson requires policies to document their behavior and it lets the system administrator decide which polices can run safely together.

Fourth, we show that Dyson can operate at a sufficiently large scale to be of practical use. We demonstrate this via large-scale experiments with one of our policies, and also by careful micro-benchmarking of several aspects of the Dyson architecture.

Fifth, we show although Dyson can operate without client-side modifications, using Dyson-enabled clients significantly improves performance, and enables features

```
# Input: client MAC, list of (AP MAC, RSSI) for
#   each received probe request
# Output: client MAC, AP with highest
#   available capacity
def (clientmac, heard_list):
  global ap_list, ap_list_lock, ratemap
  best_ap = None
  max_ac = -1

  # Compute available capacity for each AP
  # Pick AP with the highest value
  for (apmac, rssi) in heard_list:
    ap_list_lock.acquire()
    data_rate = ratemap.get_rate(rssi)
    airtime = ap_list[apmac].airtime
    avail_capacity = data_rate * (1.0 - airtime)
    if avail_capacity > max_ac:
      max_ac = avail_capacity
      best_ap = ap_list[apmac]
    ap_list_lock.release()

  # Assign channel if no clients already
  if (best_ap.channel == -1):
    best_ap.assign_channel()
  # Associate client
  best_ap.AcceptClient(clientmac)

def run(self):
  global pending_associations
  global pending_associations_lock

  while (True):
    pending_associations_lock.acquire()
    map(compute_ac, pending_associations)
    pending_associations = []
    pending_associations_lock.release()
    time.sleep(5)
```

Figure 3: **The Dyson capacity-aware association policy.**

that would not otherwise be possible. This is essential because client modifications are generally considered to be disruptive and expensive. We illustrate this by running one of the policies both with and without client modifications.

4.1 Customizing associations

We begin with a simple demonstration of Dyson's extensibility mechanisms at work. In Figure 3, we present a policy that associates clients with access points based on the estimated *channel capacity* at each AP. This policy is similar to the one proposed in DenseAP [22], but rather than being implemented as a complete, standalone system, using Dyson we implement it in approximately 40 lines of Python code.

The key idea is to use information on airtime utilization and an estimate of the feasible PHY rates to determine the best AP with which to associate a given client. The policy runs every 5 seconds. On each iteration, it scans over a list of probe requests received from clients. A given probe request may have been overheard by multiple APs. For each AP, the available channel capacity is computed, which is the product of the estimated PHY rate at which the client and AP will communicate, and the inverse of the AP's measured airtime utilization. The PHY rate is determined using a *rate map* that maps the RSSI of the received

```
  # Compute available capacity for each AP
  # Pick AP with the highest value. But ensure
  # this association is not an exception
  for (apmac, rssi) in heard_list:
    ap_list_lock.acquire()
    data_rate = ratemap.get_rate(rssi)
    airtime = ap_list[apmac].airtime
    avail_capacity = data_rate * (1.0 - airtime)

    #Check if this association is prohibited
    # (Code not shown ... )
    if is_exception(clientmac, apmac):
       if avail_capacity > max_ac:
         max_ac = avail_capacity
         best_ap = ap_list[apmac]
    ap_list_lock.release()
```

Figure 4: **A snippet of modifications necessary to the capacity-aware association to account for interference.**

probe request to the maximum feasible PHY rate for that client/AP pair. The rate map computation is performed separately and is not shown in the code in Figure 3.

The AP with the maximum available capacity is selected as the one that the client should associate with. If the AP currently has no clients, a channel is assigned to it, and the AP is then instructed to accept the client's probe request, by sending a probe response. This policy is used as the default association policy in Dyson and is used by the subsequent policies unless otherwise specified. We have performed experiments to confirm that its performance is similar to DenseAP's association scheme [22]. *The key takeaway from this example is the ease and conciseness of writing a Dyson policy whose functionality mimics that of a system proposed earlier.*

4.2 Interference-aware association policy

The association policy described in the previous section does not explicitly take interference into account. Recent research [26] has shown the benefits of explicitly accounting for interference between clients. A system administrator may wish to utilize such knowledge to improve associations in the WLAN. Prior systems [26, 22] do not permit such rapid changes to the WLAN.

However, due to the flexibility of Dyson, as seen in Figure 4, we can easily modify the basic policy shown in Figure 3 to account for interference. The change to the policy is minor because it checks if a particular client associating with an AP is part of the same exception.

A simple case of interference is when two clients, associated with different APs, can hear each other. The Dyson central controller can easily detect such cases (see Figure 5, based on information reported by clients and can take remedial action if necessary. For example, it can change the channel of one of the APs.

The interference-aware association policy periodically scans the global connectivity graph and detects cases in which two APs and two clients form an interference relationship similar to that in Figure 5. The policy changes the

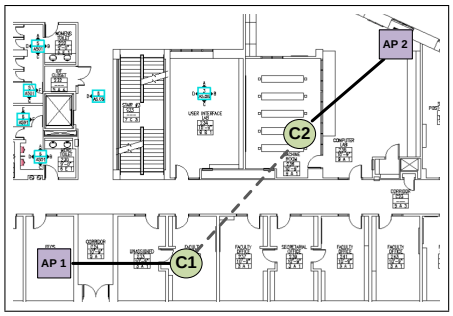


Figure 5: **Interference example. The two clients determine they interfere with each other, despite being associated with different APs.**

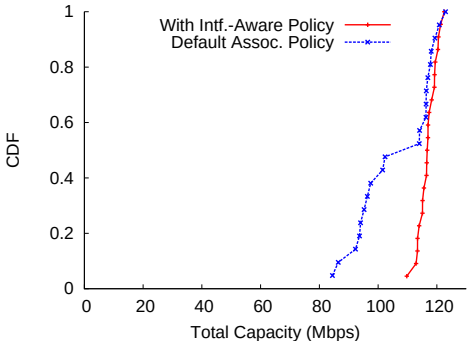


Figure 6: **Impact of interference mitigation policy on throughput of nodes across the entire floor across 20 separate runs.**

channel of the particular AP (and its clients) with fewer associated clients. The affected nodes are informed of the channel switch directly via a command, thereby avoiding the overhead of re-discovery and re-association if the policy were to simply change the channel of the AP. Note that this simple greedy algorithm might induce a new interference condition elsewhere in the network, necessitating another channel switch. To avoid oscillations, we do not change an AP's channel more than once every 10 minutes.

To demonstrate the impact of this policy, we ran an experiment with 5 of the testbed nodes acting as APs and 16 nodes acting as clients. The APs and clients were distributed roughly evenly across the testbed. The clients generated uplink traffic using greedy TCP flows for 5 minutes. We first ran the experiment using the association policy described in Figure 3, and then repeated the experiment using the modification described in Figure 4. We repeated the entire experiment 20 times.

For each run, we obtained the total capacity of the network by summing up the total throughput achieved via each AP. The results, as seen in Figure 6 show that the interference aware association policy significantly improves the capacity of the network.

There are two key takeaways from this example. First, we show how interesting performance optimizations can be enabled in Dyson with just a few lines of Python code. Second, we show the usefulness of modifying clients: without detailed measurements from the clients, it would not have been possible to identify interfering pairs.

```
#Returns clients whose airtime reservations
#have not been met
def cull_special_clients(clients):
  resv_clients = []
  resv_count = 0
  for c in clients:
    if is_reserved(c)
      resv_count = resv_count + 1
      if c.get_airtime() < c.res_airtime:
        resv_clients[] = c
  return (resv_clients, resv_count)

#Throttle other clients as necessary
def throttle (ap, c,f):
  for client in ap.clients:
    if client.mac != c.mac and
       !is_reserved(c):

       #throttle/de-throttle by f%
       throttle(client.mac, f)

#Returns residual airtime at ap
def get_residual_at(ap):
# (Code not shown ... )

def run(self):
  global ap_list, ap_list_lock
  global client_list, client_list_lock
  while (True):
    ap_list_lock.acquire()
    for ap in apmap:
     residual_at = get_residual_at(ap)
     (res_clients, resv_count) =
           cull_special_clients(ap.clients,
                                residual_at)

     if len(l) > 0:
      #For each special client, throttle other
      #associated APs until targets are met
      for c in res_clients: throttle(ap, c, F)
     elif resv_count > 0:
      #needs of special client are met
      #de-throttle other clients
      for c in res_clients: throttle(ap, c, -F)
    ap_list_lock.release()
```

Figure 7: **The air-time reservation policy**

A more complex version of this policy can take historical knowledge of the network into account. For example, once an interference pattern between locations is determined, the system can proactively assign APs and clients in those locations to different channels.

4.3 User-specific airtime reservation

We now demonstrate that the Dyson architecture can enable new functionality that is not available in traditional WLAN systems, namely, reserving airtime for a specific user or group of users. Note, while some Wi-Fi networks do enable 802.11e for prioritization, 11e lacks the ability to reserve a certain fraction of airtime for a given station.

The network designer can easily accomplish this task with Dyson using the policy shown in Figure 7. A high-priority client c_h is identified by its MAC address. For all other clients $\{c_1, c_2, ...c_k\}$ associated with the same AP, the residual airtime $R = 1 - \sum_i ATU(c_i)$ is computed.

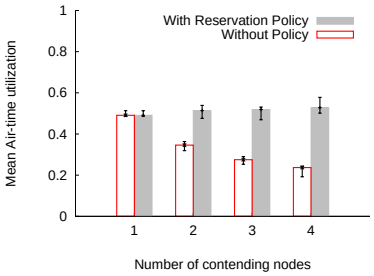
If R is less than the target airtime for c_h, the policy iterates through the list of low-priority clients, and *throttles* each of their transmission rates by a fraction f of their current throughput. This is performed using the Dyson `Throttle` command, shown in Table 2. Throttling is performed periodically until the residual airtime exceeds the target. On the other hand, if there are special users and their needs are being met, the policy then attempts to de-throttle other clients which may have been throttled.

This approach makes no assumptions about the nature of client traffic, and simply "searches" for the throttle setpoints that yield adequate airtime to the high-priority client. It is also conservative in the sense that clients are throttled equally, without regards to their load. A straightforward enhancement would throttle higher-load clients first. Note that when c_h disassociates with the AP, the low-priority clients are unthrottled; likewise, when a client moves to another AP is throttle is released. Multiple high-priority clients can also be supported on a single AP as long as their airtime targets do not exceed 100%; in that case, each high-priority client receives a weighted proportional share of the airtime. Note that this policy requires the ability to control clients directly, and hence is not possible to implement without client modifications.

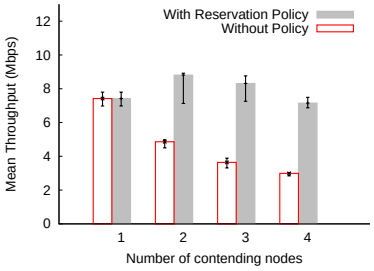
We demonstrate this policy using the following experiment. The setup consists of four APs and 11 clients. One of the clients is given an airtime reservation of 50%. For this experiment, we manually set the APs to different channels, and associate one non-privileged client with AP1, two non-privileged clients with AP2, and so on. The privileged client is nomadic. It associates with each of the four APs in turn for 10 minutes each. All clients download data as fast as they can using *iperf* UDP flows. We first perform the experiment without any reservation policy, and then repeat it after reserving 50% of the airtime for the privileged user. We repeat the entire experiment 10 times for statistical significance.

The impact of the policy is shown in Figure 4.3. In the absence of the reservation policy, the fraction of airtime received by the privileged user drops as the number of non-privileged clients increases. However, when the reservation policy is in force, the privileged user always receives the 50% reserved fraction of the airtime.

As a side note, remember that providing *guaranteed* airtime does not translate to guaranteed throughput, because of the variability in radio link quality of the link between the privileged client and each of the APs. The throughput received by the privileged client in the previous experiment is shown in Figure 8(b). Even though the privileged client receives a fixed amount of airtime, the throughput it achieves varies for different APs. We can easily modify the policy described above to ensure that the reserved airtime varies in inverse proportion to the quality of the channel seen by the privileged user, to ensure that



(a) Impact on airtime



(b) Impact on throughput

Figure 8: **Impact of airtime reservation policy on the airtime and throughput received by a single privileged user competing with several other clients. Error bars represent 10th and 90th percentiles.**

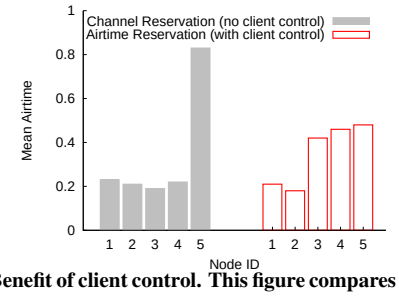
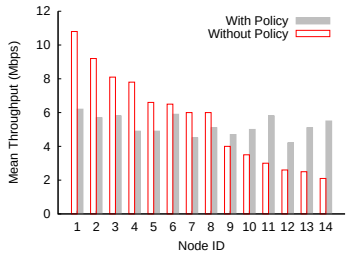


Figure 9: **Benefit of client control. This figure compares two different policies: one that exercises control over clients (airtime reservation) to one that only controls APs (channel reservation) only. Node 5 is the special user guaranteed at least 50% of the airtime. Client-side throttling leads to a more fair distribution of airtime across clients than reserving channels alone.**

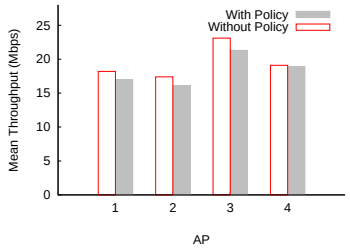
the bulk transfers done by the user receive certain guaranteed target throughput. Though we have implemented and tested this policy, we elide details due to lack of space.

There are two key takeaways from this example. First, we show that Dyson can enable novel functionality, that is not otherwise available. Second, this example also demonstrates that certain kinds of functionality can only be enabled via client modifications.

Benefits of client-side control: Throttling airtime usage at the client gives Dyson direct control over client behavior. The question that arises is whether such client control is strictly necessary. As an alternative, consider a policy that does not assume any client extensions. As opposed to reserving airtime, the policy reserves an entire channel on a given AP for special users, requiring changes to client-AP associations in order to avoid inter-



(a) Throughput for various nodes with/without load balancing



(b) Total throughput at each AP with/without load balancing

Figure 10: **Large-scale load balancing experiment.**

ference between special users and regular users. We have implemented such a policy and compared it against the airtime reservation policy. The setup consists of two APs and four clients (two associated with each AP) performing uplink TCP traffic. A fifth client arrives, one of which is guaranteed 50% of the airtime. In the channel reservation policy, clients from one AP are moved to the other, which has four clients contending. The fifth user is given exclusive access to the vacated AP. With the airtime reservation policy, the fifth user associates with one of the APs where the other clients are throttled (the clients at the other AP are left untouched). The results are shown in Figure 4.3, which shows that the channel reservation policy leads to a less fair distribution of airtime than the airtime reservation policy. This experiment demonstrates the added value of control over both clients and the infrastructure. Note, another possible approach here would have been to enable 802.11e priority queues. However, our intention is to demonstrate the ease with which different kinds of policies, each offering a different degree of control, can be implemented and deployed in Dyson.

4.4 Uplink/downlink load balancing

In this section, we show how Dyson can be used to correct a basic flaw in the 802.11 architecture, called the upload-download anomaly [24]. The 802.11 MAC ensures that each node with pending packets gets equal opportunity to access the channel. Consider a WLAN with ten clients associated with a single AP. Nine of these clients download data from the network, while one client uploads data to the network. Because the AP and the upload client are the only two nodes that have pending data to send, they share the airtime equally. The upload client gets to transmit roughly half the time, while the nine download clients *together* share the remaining time!

Although traditionally the majority of the traffic in WLANs has been download traffic [4], this pattern is expected to change as WLANs become more popular as access networks. As the above example shows, only a few upload clients are needed to cause significant unfairness.

In a Dyson-enabled network, we can easily address this problem via the following simple policy. The policy attempts to balance the total volume of uplink and downlink traffic handled by an AP. For each AP, associated clients are classified as either predominantly upload or download, based on the ratio of their throughput in each direction. We then compute the ratio of the mean throughput for upload and download clients. If the ratio exceeds a specified threshold, it suggests that upload clients are dominant and that rebalancing is required for this AP.

As a simple approach, the policy throttles upload clients in an attempt to bring the upload/download ratio closer to 1. Upload clients are ordered by decreasing uplink throughput, and the "heaviest" upload client is throttled to 50% of its current throughput. The policy then sleeps for 10 sec and re-evaluates the upload/download ratio, iteratively throttling the highest-throughput upload client until the ratio between the mean upload and mean download throughput at the AP falls to less than 1.5.

This policy relies on client cooperation to solve the problem. This is not strictly necessary, but other remedies are more disruptive. For example, the policy could attempt to modify client-AP associations to balance the number of download clients across APs. However, such a policy is not always guaranteed to achieve the correct distribution of clients required. The client-throttling approach described above is much simpler to implement.

To demonstrate this policy, we performed an experiment with 4 APs and 15 client nodes. Client-AP associations were determined using the capacity-aware association policy (Section 4.1). Note that different APs have a different number of associated clients. Each AP is assigned to a different channel by the policy.

One client associated with each AP generates upload traffic, while others generate download traffic. We ran the experiment twice, first without the uplink/downlink load balancing policy running, and then with the policy enabled. Figure 10(a) shows the distribution of the throughput obtained by each of the clients with and without the policy running. There is a clear bandwidth inequity in the default case, but the policy produces a much more balanced distribution of network capacity to each client.

Of course, achieving fairness is often at odds with maximizing overall network capacity. Figure 10(b) shows the aggregate throughput at each AP before and after the pol-

icy was enabled. As the figure shows, there is a slight dip in overall bandwidth usage at each AP: 5.7% on average.

This example illustrates that a simple Dyson policy can correct problems inherent in the 802.11 architecture by using feedback from the client and by exercising control over clients.

4.5 VoIP-aware handoffs

As a another example of Dyson's ability to enable network-wide optimizations, we present an example policy that assigns VoIP clients to a different set of APs than other clients, to increase overall VoIP call capacity and avoid bulk transfers from impacting VoIP call quality. This policy assumes that clients have been classified as VoIP or non-VoIP clients, for example, based on the client's MAC address (e.g., for WiFi VoIP handsets). Due to lack of space, we omit the python code for this policy.

For each VoIP client that is assigned to a non-VoIP AP, the policy identifies a new VoIP-specific AP with which to associate. For each VoIP AP that the client can potentially connect to (based on the connectivity graph), the available capacity metric is computed, as described earlier. The client is simply handed off to the VoIP AP with the highest available capacity.

Although there are more sophisticated techniques to improve VoIP capacity in WiFi networks [29], this policy is simply intended to demonstrate Dyson's interfaces and programmability. This simple policy can be extended in various ways. For example, the assignment of APs as VoIP or non-VoIP (which is currently static) can be performed in a dynamic fashion based on VoIP call load. Likewise, the number of VoIP clients assigned to each AP could be taken into consideration. We elide the details due to lack of space.

We carry out the following experiment. We configured two nodes near each other as APs, and another four nodes as clients. The capacity-aware association policy described in Section 4.1 was used, resulting in two clients being associated with each AP. The APs were assigned to different channels by the association policy.

Two clients, on separate APs, initiated a bidirectional VoIP flow while the other two clients began large saturating download traffic using *iperf*. The VoIP flows each use a standard g729 VoIP codec that generates 50-byte packets at a rate of 31.2 Kbps.

The bulk flows adversely affect the VoIP flows in terms of introducing increased packet jitter, which causes the quality of the VoIP call to degrade. A common requirement for VoIP calls is that jitter should be no greater than 2ms [3]. Figure 4.5 shows that with the default configuration, up to 2.17 ms of jitter is induced by the bulk flows on each VoIP call. Note, this is done with a few clients.

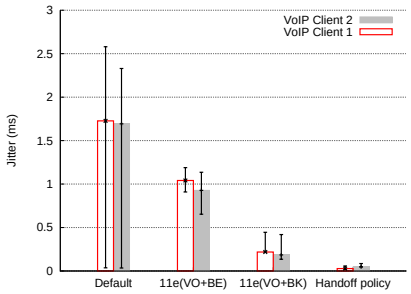


Figure 11: **Effect of 802.11e prioritization and VoIP-aware handoffs on VoIP jitter.** ***This is an experiment with two VoIP clients competing with two bulk-download clients, with two APs on different channels. Using the* default *policy, one VoIP client and one bulk client are assigned to each AP. The* 11e(VO+BE) *policy uses 802.11e prioritization, assigning bulk clients to the* best effort *queue.* 11e(VO+BK) *assigns bulk clients to the* background *queue.***

Next, we enabled the VoIP handoff policy, which migrates VoIP clients to one of the APs and the bulk flows to the other. As Figure 4.5 shows, this substantially reduces the jitter to a mean of 0.02 ms. This also causes the bulk transfers to share the channel on a single AP, causing their throughputs to degrade; prior to migration, each bulk flow obtained 24 Mbps of throughput. After migration, each bulk flow degrades to 12 Mbps. This is an explicit tradeoff between providing good service to VoIP clients versus the (arguably less severe) impact on bulk flows.

As an alternative, we also experimented with using 802.11e priority levels, with a simple policy that uses the `SetPriority` command. We set up one experiment in which the VoIP clients were configured to use the 802.11 *voice* priority and the bulk clients to use the 802.11e *best effort* priority, while maintaining the original AP associations. Another experiment uses the 802.11e *background* priority, which is lower than best-effort. As the figure shows, 802.11e priorities do mitigate some of the jitter effects, but do not operate as well as the handoff policy. Each bulk client received 24 Mbps of throughput using the best-effort priority, and 18 MBps using the background priority. In general, it will not always be possible to cleanly separate VoIP clients from others in the network, so in general a combination of migration (where possible) and 802.11e priority levels is likely to be the most effective solution.

Note this policy could have been implemented in prior systems, such as SMARTA [6], MDG [11], or DenseAP [22]. Note, however, that this is easy to do so in Dyson via the exposed API. The *key takeaway from this example is the platform Dyson provides to develop and deploy such polices very quickly.*

4.6 Running multiple policies together

So far, we have demonstrated each policy in isolation. However, a network administrator will often want to run

Figure 12: **The floor map for our testbed, which was divided into four regions. Each region is configured to run a different set of policies.**

multiple policies simultaneously and compose their behavior. For example, different types of traffic may need to be given different priorities in different parts of the building, or at different times of day.

Dyson supports running multiple policies on different spatial regions of the network, and varying the set of policies that are active over time. Furthermore, Dyson can track client locations, and ensure that appropriate polices are applied depending on the client's location.

To illustrate the use of multiple policies, we ran the following experiment on our testbed. We divide our floor into four regions as shown in Figure 4.6. Regions 1, 3, and 4 do not overlap with each other, whereas region 2 overlaps with 1 and 3. For each region, we configure Dyson to run the following policies.

Region 1: We reserve one of the APs (and one of the channels) for VoIP traffic and enable the VoIP sifting policy described in Section 4.5. This ensures we dedicate resources to VoIP clients in this area.

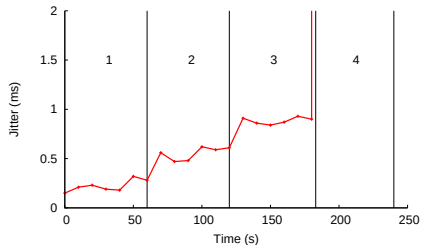
Region 2: In this region, we run the interference-aware association policy described in Section 4.2.

Region 3: In this region, we reserve 80% of the airtime for a set of users. For example, the administrator may want to deploy such a policy in a region where faculty offices are located, giving faculty members preferential treatment.

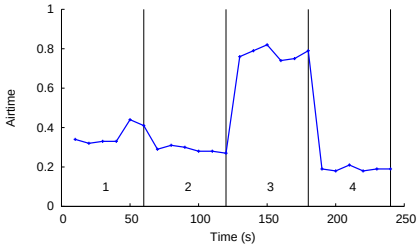
Region 4: In this region, we disable all VoIP calls using a policy that dissociates any client that is transmitting VoIP traffic.

Each region contains at least two APs, and each AP has anywhere between 2–4 clients associated with them performing variable bit rate UDP traffic. In addition, there are two nomadic users, whose behavior we monitor. User 1 is a VoIP client who starts walking in region 1, 2, 3, and finally ends in 4. User 2 is another nomadic user who is performing bulk TCP transfers. This user is a faculty member who is guaranteed by the policy in Region 3 to get 80% of the airtime. Each user spends approximately 60 seconds in each location.

The jitter for User 1 and airtime for User 2 are shown in Figure 4.6. As the figure shows, the VoIP user experiences significantly less jitter when she is in Region 1, compared to other regions, because an AP and a channel are reserved for VoIP calls in this region. Furthermore, when she enters Region 4, her service is cut off. We also



(a) Nomadic VoIP User



(b) Nomadic Prioritized User

Figure 13: **Time series graphs for the two different nomadic users as they walk through the various regions.**

see that User 2 gets her reserved airtime only when she is in Region 3, as expected.

The key takeaway from this example is that Dyson can successfully run multiple policies in a single network and apply policies in a location-specific manner.

This example does not address how multiple polices may interact with each other. In our current implementation, we rely on the network administrator to determine if policies may have adverse interactions or may simply cancel out each other's decisions. It is assumed that policies themselves are well-documented and that the administrator can reason about the possible interactions between multiple policies running on the same parts of the network. In our future work, we plan to build tools to help administrators detect and resolve such conflicts.

4.7 Other Policies

Due to a lack of space, we are unable to present results from other policies we have implemented and experimented with, in the Dyson system. One such policiy is a one that reduces the cumulative handoffs for certain mobility paths. Dyson can use historical knowledge of client mobility patterns to optimize AP handoffs. Since mobile handoffs are expensive and can lead to temporary connectivity loss, it is important to avoid redundant or poorly-chosen handoffs. The key idea is to predict the next AP a client will encounter while roaming, in order to avoid handing off to a different AP that will quickly go out of range. This is possible, since in many workplaces, users are more likely to travel along certain paths than others. We built and deployed this policy on our floorwide testbed and we found, on average it halves the number of handoffs

Stats interval (s)	10th	Median	90th
1	3.5%	4.3%	9.3%
5	0.8%	3.9%	6.1%
10	0.0%	3.5%	6.2%

Table 3: **CPU utilization at an AP with eight clients measured over a period of 10 minutes, for various statistics reporting intervals.**

for well traversed paths.

Along the lines of the airtime reservation policy, we have also implemented a bandwidth reservation policy whereby we guarantee a certain bandwidth to a user, and we throttle other clients nearby until we reach the desired target. There are host of other policies that are variants of the policies described in this paper, that we are currently working in the Dyson system.

4.8 Microbenchmarks

In this section, we study the overheads imposed by the Dyson architecture on clients and APs and the amount of control traffic generated by Dyson nodes. We will also study the performance penalty caused by client handoffs, as Dyson uses this mechanism often.

AP and client overheads: We examine the CPU utilization of the AP. Recall that we use an ALIX 2c2 with a 500 MHz AMD Geode, and the Dyson software is implemented in Python. We configured an AP with eight clients and varied the intervals at which the AP reported statistics to the CC. As seen in Table 3 the median utilization is still low. Since clients are periodically sending statistics to the AP, we also measured the CPU utilization at a client over a period of ten minutes. We found the modified Dyson drivers added negligible overhead in terms of CPU and memory utilization ($< 1\%$).

Traffic overhead for measurments collection: We also measured the traffic sent by clients and APs to the CC. The AP's measurement collection period can be adjusted by the CC to tradeoff reporting latency and measurement traffic overhead. As an estimate of this overhead, each client measurement packet requires at most 850 bytes, including MAC headers. At the lowest OFDM PHY rate of 6 Mbps, this requires 1184μs to transmit (accounting for MAC and framing overheads). Therefore, an AP with 20 clients will require less than 1% of the radio channel for statistics collection. The AP sends all client statistics as well its own statistics to the CC. We measured the traffic sent by an AP with six clients to the CC. With a statistics reporting interval of 5 sec, the AP generates 1638 bytes/sec in traffic to the CC, which includes overheads induced by the use of XML-RPC. This is a small fraction of the backhaul wired network capacity.

Handoff overhead: We measured the time taken for a MAC-layer handoff of a client from one AP to another. We configured two APs (on different channels) and a single client, which was initially associated with AP1. The CC then issued a `Handoff` command to migrate the client to AP2. AP1 receives this command and relays it to the client who quickly switches channels and associations. This process also includes informing AP2 to permit the new association.

Step	Time(ms)
Handoff command executed	0
Message reception (at client)	0.120
Channel change	5.6
Authentication	0.159
Association	0.359
Total	6.238

Table 4: **Handoff overhead in Dyson.**

The MAC-layer handoff overhead includes the time for the command transmission to the client, the time for the client to switch channels (between to 5 to 7 ms on the Atheros chipset), and the client's reassociation with the new AP. The end-to-end delay experienced by an application may be longer, for example, due to the settling time of the spanning-tree algorithm on the wired backbone.

The results are shown in Table 4, which shows that a MAC-layer handoff requires approximately 6.2 ms in our current prototype. This process can be further optimized, as demonstrated in [25]. Also, the use of protocols such as IAPP (Inter-Access Point Protocol) at a higher layer, in which APs cache packets during a handoff and forward them to the destination AP, can mitigate the packet loss incurred during a handoff. We have not yet implemented this approach in Dyson.

Central controller scalability: Dyson uses a central controller to control the APs and the clients. This raises questions about the scalability and fault tolerance of our architecture. If necessary, fault tolerance can be achieved using standard techniques such as primary-backup. We believe that scalability is not a concern, since the processing done at the central controller is not CPU or memory intensive for most polices that we envision. In all our experiments, the load on the central controller was negligible. However, it may be the case that certain policies or certain deployments may require extensive processing capabilities at the central controller. In such a case, it may be possible to use multiple machines to act as central controllers and use standard load balancing techniques to prevent any one machine from becoming a bottleneck. We plan to study these issues in detail as part of our future work.

5 Related Work

Dyson is complementary to a broad class of prior work on improving the performance and scalability of wireless networks through new techniques at the MAC and PHY layers [28, 9]. Our focus is on the higher-level aspects of network management that can be obtained through global observation and deep control.

Dyson is inspired by the same vision that inspired projects such as OpenFlow [19] and 4D [14], where significant intelligence resides in a central controller. The central controller makes use of global knowledge to make network-wide decisions. We note that the Dyson architecture is quite compatible with the overall OpenFlow design. We are currently investigating whether some parts of Dyson functionality (especially AP controls) can be recast in the OpenFlow model.

Several commercial systems use some form of global knowledge or a central controller for managing WLAN deployments. Aruba [1] uses central controller to do network-wide channel and power management to mitigate interference, while Meru [2] uses a central controller to speed up handoffs for mobile clients. Detailed information on how these systems work is difficult to come by - the marketing literature does not reveal much. However, commercial vendors are hampered by the need to maintain backwards compatibility with existing 802.11 networks. To the best of our knowledge, no commercial system includes a client component.

Research systems such as DenseAP [22] and DIRAC [32] also propose a centralized architecture. However, both systems explicitly assume that no special software can be run on clients, and thus are limited in what they can accomplish. Centaur [26] does use some form of client modifications, along with centralized control. However, Centaur has a narrow goal: to avoid hidden and exposed terminal issues. Dyson is a much more general system. In fact, in Section 4.2, we have shown how Dyson can find and avoid certain kinds of interference and hidden terminal problems.

Several research systems use a limited form of client cooperation. In MDG [11], clients get information from APs via special fields in the Beacon packets, and the client driver uses this information to make various decisions (e.g. associations). However, the specified interface is quite limited, and is more akin to the one proposed in the 802.11k standard [15]. Similarly, [21] uses feedback from clients to enable use of partially overlapping channels, [6] uses client-cooperation via micro-probing [5] to construct a conflict graphs [23] of the network.

The Dyson architecture, on the other hand, provides a general-purpose API for managing clients and APs, and can be viewed as a generalized version of these systems.

Systems such as SoftRepeater [8] and CMAP [30] specifically focus on client cooperation to improve WLAN performance. In SoftRepeater, clients with good connections relay packets for poorly-connected clients. Similar functionality can be implemented as a policy in the Dyson framework. In CMAP, clients collaborate to build an interference map of the network, which is used to schedule transmissions. Dyson's network map is a generalized version of CMAP's interference graph.

Another interesting design point is explored in [27]. The idea is to use bare-bones APs with analog-to-digital converters such that they are oblivious to the PHY/MAC layers being used at the client. As a result, all intelligence in the network is pushed to the clients. The Dyson approach is practical, and can be deployed with off-the-shelf 802.11 hardware.

Outside of the networking space, many systems have explored the use of extensibility via add-on modules with a well-defined programmatic interface. SPIN [10] and Exokernel [16] are classic examples of opening up the operating system interface to permit greater flexibility and application-specific control. Likewise, Lance [31] provides a policy module interface to customize data collection from a wireless sensor network.

6 Discussion and Future Work

Our prototype of Dyson has shed light on several directions for future work. First, our current design assumes that Dyson-enabled clients will be able to provide periodic measurement reports regardless of their power state. Power-constrained clients such as mobile phones routinely turn off their Wi-Fi interfaces (power save mode), and hence may not always be able to collect or report these measurements. This raises the question of what the impact of intermittent measurements collection will have on efficacy of Dyson policies. If the density of non-power-constrained clients (e.g. laptops on people's desks) is sufficiently high, good measurements can still be collected. Alternatively, a separate monitoring system like DAIR [7] can be used. In some cases, the design of polices itself will have to change to deal with partial information. We are exploring these alternatives further.

We have designed Dyson primarily for enterprise networks, where clients are under the control of a central IT department and do not need incentives for running the measurement software. We have also not considered the impact of malicious users reporting false measurements or not responding to commands. These concerns are addressed partially by the fact that in most enterprise networks, WLAN users are explicitly authenticated using protocols such as 802.1x. Another interesting possibility is to identify malicious users by comparing measurement reports from different clients [18].

In the current Dyson prototype, clients perform only passive measurements. This was done for the sake of simplicity. We plan to explore the possibility of asking clients to perform *active* measurements, e.g., asking a client to transmit a series of probe packets to measure loss rate more accurately. Concerns about overhead and battery drain will likely limit how often such active measurements are carried out. In the same vein, one may also ask certain clients to relay packets for other clients [8]. We have not

considered such possibilities in the current prototype.

Finally, we note that while it is easy to write new Dyson policies, it does require some expert knowledge, especially to avoid unwanted interactions between polices that run simultaneously. We do not expect an average system administrator to have the requisite skill set. We believe if Dyson is deployed in a widespread manner, a new class of experts in programmable network management will arise who will write and distribute pre-packaged policies.

7 Conclusions

We have presented Dyson, a new architecture for extensible wireless LANs. Dyson provides an extensible network architecture that evolves with new challenges and application demands. Dyson's programmable policy framework makes it easy to customize the network's operation for site-specific needs and new services. The framework also makes it easy to store historical information about network performance, and leverage it to fine-tune network parameters. By "opening up" clients for measurements collection and control, Dyson breaks down the traditional barrier between the infrastructure and its clients, offering substantial benefits for network management.

We demonstrated how Dyson can support a wide range of policies for managing associations, specialized traffic classes (such as VoIP), mitigating interference and airtime reservations for specific users. We demonstrated the benefits of these policies using our 28-node testbed.

References

[1] Enterprise solutions from aruba networks, http://www.arubanetworks.com/solutions/enterprise.php.

[2] Meru networks - virtual cell, http://www.merunetworks.com/pdf/whitepapers/.

[3] A reference guide to all things voip, http://www.voip-info.org/wiki/view/qos.

[4] AHMED, N., BANERJEE, S., KESHAV, S., MISHRA, A., PAPAGIANNAKI, K., AND SHRIVASTAVA, V. Interference Mitigation in Wireless LANs using Speculative Scheduling . In *MobiCom* (2007).

[5] AHMED, N., ISMAIL, U., KESHAV, S., AND PAPAGIANNAKI, D. Online Estimation of RF Interference. In *CoNEXT* (2008).

[6] AHMED, N., AND KESHAV, S. SMARTA: A Self-Managing Architecture for Thin Access Points. In *CoNEXT* (2006).

[7] BAHL, P., CHANDRA, R., PADHYE, J., RAVINDRANATH, L., SINGH, M., WOLMAN, A., AND ZILL, B. Enhancing the Security of Corporate Wi-Fi Networks Using DAIR. In *MobiSys* (2006).

[8] BAHL, V., CHANDRA, R., LEE, P., MISRA, V., PADHYE, J., RUBENSTEIN, D., AND YU, Y. Opportunistic Use of Client Repeaters to Improve Performance of WLANs. In *CoNext* (2008).

[9] BEJERANO, Y., AND BHATIA, R. S. MiFi: a framework for fairness and QoS assurance in current IEEE 802.11 Networks with Multiple Access Points. In *Infocom* (2004).

[10] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proc. the 15th SOSP (SOSP-15)* (1995).

[11] BROUSTIS, I., PAPAGIANNAKI, K., KRISHNAMURTHY, S. V., FALOUTSOS, M., AND MHATRE, V. MDG: Measurement-driven Guidelines for 802.11 WLAN Design. In *MobiCom* (2007).

[12] CHANDRA, R., PADHYE, J., WOLMAN, A., AND ZILL, B. A Location-based Management System for Enterprise Wireless LANs. In *NSDI* (2007).

[13] CHENG, Y.-C., AFANASYEV, M., VERKAIK, P., BENKO, P., CHIANG, J., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Automated Cross-Layer Diagnosis of Enterprise Wireless Networks. In *SIGCOMM* (2007).

[14] GREENBERG, A., HJALMTYSSON, G., MALTZ, D., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. In *SIGCOMM CCR* (2005).

[15] IEEE. *IEEE 802.11k-2008 — Amendment 1: Radio Resource Measurement of Wireless LANs.* June 2008.

[16] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *Proc. the 16th SOSP (SOSP '97)* (October 1997).

[17] KO, B.-J., MISRA, V., PADHYE, J., AND RUBENSTEIN, D. Distributed channel assignment in multi-radio 802.11 mesh networks. In *WCNC* (2007).

[18] MAHAJAN, R., RODRIG, M., WETHERALL, D., AND ZAHORJAN, J. Sustaining Cooperation in Multi-Hop Wireless Networks. In *Proc. NSDI* (2005).

[19] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks.

[20] MISHRA, A., BRIK, V., BANERJEE, S., SRINIVASAN, A., AND ARBAUGH, W. A Client-driven Approach for Channel Management in Wireless LANs. In *Infocom* (2006).

[21] MISHRA, A., SHRIVASTAVA, V., BANERJEE, S., AND ARBAUGH, W. Partially-overlapped Channels not considered harmful. In *ACM Sigmetrics* (2006).

[22] MURTY, R., PADHYE, J., CHANDRA, R., WOLMAN, A., AND ZILL, B. Designing High-Performance Enterprise Wireless Networks. In *NSDI* (San Francisco, CA, April 2008).

[23] PADHYE, J., AGARWAL, S., PADMANABHAN, V., QIU, L., RAO, A., AND ZILL, B. Estimation of Link Interference in Static Multi-hop Wireless Networks. In *IMC* (2005).

[24] PILOSOF, S., RAMJEE, R., RAZ, D., SHAVITT, Y., , AND SINHA, P. Understanding TCP fairness over Wireless LAN. In *INFOCOM* (2003).

[25] SHARMA, A., AND BELDING, E. M. FreeMAC: Framework for Multi-Channel MAC Development on 802.11 Hardware. In *ACM SIGCOMM PRESTO* (2008).

[26] SHRIVASTAVA, V., AHMED, N., RAYANCHU, S., BANERJEE, S., KESHAV, S., PAPAGIANNAKI, K., AND MISHRA, A. CENTAUR: Realizing the Full Potential of Centralized WLANs through a Hybrid Data Path. In *MOBICOM* (2009).

[27] SINGH, S. Challenges: Wide-Area wireless NETworks (WANETs). In *MOBICOM* (2008).

[28] VASAN, A., RAMJEE, R., AND WOO, T. ECHOS - Enhanced Capacity 802.11 Hotspots. In *Infocom* (2005).

[29] VERKAIK, P., AGARWAL, Y., GUPTA, R., AND SNOEREN, A. C. SoftSpeak: Making VoIP Play Fair in Existing 802.11 Deployments. In *NSDI* (2009).

[30] VUTUKURU, M., JAMIESON, K., AND BALAKRISHNAN, H. Harnessing Exposed Terminals in Wireless Networks. In *NSDI* (2008).

[31] WERNER-ALLEN, G., DAWSON-HAGGERTY, S., AND WELSH, M. Lance: Optimizing high-resolution signal collection in wireless sensor networks. In *Proc. Sensys* (2008).

[32] ZERFOS, P., ZHONG, G., CHENG, J., LUO, H., LU, S., AND L, J. J.-R. DIRAC: a software-based wireless router system. In *MOBICOM* (2003).