FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput

SRIKANTH KANDULA
MIT
KANDULA@MIT.EDU

KATE CHING-JU LIN
NTU/MIT
KATE@CSAIL.MIT.EDU

TURAL BADIRKHANLI
MIT
TURALB@MIT.EDU

DINA KATABI
MIT
DK@MIT.EDU

Abstract– It is increasingly common that computers in residential and hotspot scenarios see multiple access points (APs). These APs often provide high speed wireless connectivity but access the Internet via independent, relatively low-speed DSL or cable modem links. Ideally, a client would simultaneously use all accessible APs and obtain the sum of their backhaul bandwidth. Past work can connect to multiple APs, but can neither aggregate AP backhaul bandwidth nor can it maintain concurrent TCPs across them.

This paper introduces FatVAP, an 802.11 driver that aggregates the bandwidth available at accessible APs and also balances their loads. FatVAP has three key features. First, it chooses the APs that are worth connecting to and connects with each AP just long enough to collect its available bandwidth. Second, it ensures fast switching between APs without losing queued packets, and hence is the only driver that can sustain concurrent high throughput TCP connections across multiple APs. Third, it works with unmodified APs and is transparent to applications and the rest of the network stack. We experiment with FatVAP both in our lab and hotspots and residential deployments. Our results show that, in today's deployments, FatVAP immediately delivers to the end user a median throughput gain of 2.6x, and reduces the median response time by 2.8x.

1 INTRODUCTION

Today, WiFi users often see many access points (APs), multiple of which are open [10], or accessible at a small charge [9]. The current 802.11 connectivity model, which limits a user to a single AP, cannot exploit this phenomenon and, as a result, misses two opportunities.

- It does not allow a client to harness unused bandwidth at multiple APs to maximize its throughput. Users in hotspots and residential scenarios typically suffer low throughput, despite the abundance of high-speed APs. This is because these high-speed APs access the Internet via low capacity (e.g., 1Mb/s or less) DSL or cable modem links. Since the current connectivity model ties a user to a single AP, a user's throughput at home or in a hotspot is limited by the capacity of a single DSL line, even when there are plenty of high-speed APs with underutilized DSL links.
- It does not facilitate load balancing across APs. WiFi users tend to gather in a few locations (e.g., a conference room, or next to the window in a cafe). The current 802.11 connectivity model maps all of these users to a single AP, making them compete for the same limited resource, even when a nearby AP hardly has any users [12, 24]. Furthermore, the mapping is relatively static and does not change with AP load.

Ideally, one would like a connectivity model that approximates a *fat virtual AP*, whose backhaul capacity is the sum of the access capacities of nearby APs. Users then compete fairly for this fat AP, limited only by security restrictions. A fat AP design benefits users because it enables them to harness unused bandwidth at accessible APs to maximize their throughput. It also benefits AP owners because load from users in a campus, office, or hotel is balanced across all nearby APs, reducing the need to install more APs.

It might seem that the right strategy to obtain a fat virtual AP would be to greedily connect to every AP. However, using all APs may not be appropriate because of the overhead of switching between APs. In fact, if we have to ensure that TCP connections simultaneously active across multiple APs do not suffer timeouts, it might be impossible to switch among all the APs. Also, all APs are not equal. Some may have low load, others may have better backhaul capacities or higher wireless rates (802.11a/g vs. 802.11b). So, a client has to ascertain how valuable an AP is and spend more time at APs that it is likely to get more bandwidth from, i.e., the client has to divide its time among APs so as to maximize its throughput. Further, the efficiency of any system that switches between APs on short time scales crucially depends on keeping the switching overhead as low as possible. We need a system architecture that not only shifts quickly between APs, but also ensures that no in-flight packets are lost in the process.

While prior work virtualizes a wireless card allowing it to connect to multiple APs, card virtualization alone cannot approximate a fat virtual AP. Past work uses this virtualization to bridge a WLAN with an ad-hoc network [6, 13], or debug wireless connectivity [11], but

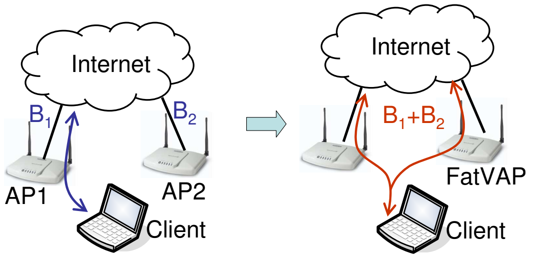


Figure 1: An example scenario where a client can potentially obtain the sum of the backhaul bandwidth available at the two APs.

can neither aggregate AP backhaul bandwidth nor balance their load. This is because it cannot tell which APs are worth connecting to and for how long. Further, it has a large switching overhead of 30-600ms [7, 13] and hence cannot be used for switching at short time-scales on the order of 100 ms, which is required for high-throughout TCP connections across these APs.

This paper introduces FatVAP, an 802.11 driver design that enables a client to aggregate the bandwidth available at accessible APs and balance load across them. FatVAP approximates the concept of a fat virtual AP given the physical restrictions on the resources. To do so, FatVAP periodically measures both the wireless and end-to-end bandwidths available at each AP. It uses this information as well as an estimate of the switching overhead to connect to each AP for just enough time to collect its available bandwidth and toggle only those APs that maximize user throughput.

The FatVAP driver has the following key features.

- It has an AP scheduler that chooses how to distribute the client's time across APs so as to maximize throughput.
- It ensures fast switching between APs (about 3 ms) without losing queued packets, and hence is the only driver that can sustain concurrent high throughput TCP connections across multiple APs.
- It works with existing setups, i.e., single 802.11 card, unmodified APs, and is transparent to applications and the rest of the network stack.

FatVAP leverages today's deployment scenarios to provide immediate improvements to end users without any modification to infrastructure or protocols. It does not need fancy radios, access to the firmware, or changes to the 802.11 MAC. FatVAP has been implemented in the MadWifi driver [4], and works in conjunction with autorate algorithms, carrier-sense, CTS-to-self protection, and all other features in the publicly released driver.

Experimental evaluation of our FatVAP prototype in a testbed and actual hotspot deployments shows that:

- In today's residential and Hotspot deployments (in Cambridge/Somerville MA), FatVAP immediately delivers to the end user a median throughput gain of 2.6x, and reduces the median response time by 2.8x.
- FatVAP is effective at harnessing unused bandwidth from nearby APs. For example, with 3 APs bottlenecked at their backhaul links, FatVAP's throughput is 3x larger than an unmodified MadWifi driver.
- FatVAP effectively balances AP loads. Further, it adapts to changes in the available bandwidth at an AP and re-balances load with no perceivable delay.
- FatVAP coexists peacefully. At each AP, FatVAP competes with unmodified clients as fairly as an unmodified MadWifi driver and is sometimes fairer as FatVAP will move to an alternate if the AP gets congested. Further, FatVAP clients compete fairly among themselves.

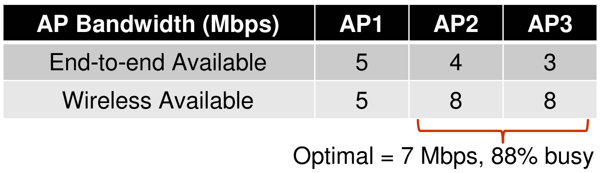
2 Motivating Examples

Not all access points are equal. An 802.11 client might have a low loss-rate to one access point; another access point might be less congested; yet another may have a high capacity link to the Internet or support higher data rates (802.11g rather than 802.11b). How should an 802.11 client choose which access points to connect to and what fraction of its time to stay connected to each AP?

To better understand the tradeoffs in switching APs, let us look at a few simple examples. Consider the scenario in Fig. 1, where the wireless client is in the range of 2 open APs. Assume the APs operate on orthogonal 802.11 channels. For each AP, let the *wireless available bandwidth*, w, be the rate at which the client communicates with the AP over the wireless link, and the *end-to-end available bandwidth*, e, be the client's end-to-end data rate when connected to that AP. Note that these values do not refer to link capacities but the throughput achieved by the client and in particular subsume link losses, driver's rate selection and competition from other clients at the AP. Note also that the end-to-end bandwidth is always bounded by the wireless available bandwidth, i.e., $e \leq w$. How should the client in Fig. 1 divide its time between connecting to AP1 and AP2? The answer to this question depends on a few factors.

First, consider a scenario in which the bottlenecks to both APs are the wireless links (i.e., $w = e$ at both APs). In this case, there is no point toggling between APs. If the client spends any time at the AP with lower available wireless bandwidth, it will have to send at a lower rate for that period, which reduces the client's overall throughput. Hence, when the wireless link is the bottleneck, the client should stick to the best AP and avoid AP switching.

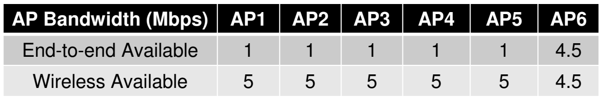
Now assume that the bottlenecks are the APs' access links (i.e., $w > e$ for both APs). As a concrete example, say that the client can achieve 5 Mb/s over either wireless link, i.e., $w_1 = w_2 = 5$ Mb/s, but the client's end-to-end available bandwidth across either AP is only 2 Mb/s, i.e., $e_1 = e_2 = 2$ Mb/s. If the client picks one of the two



AP Bandwidth (Mbps)	AP1	AP2	AP3
End-to-end Available	5	4	3
Wireless Available	5	8	8

Optimal = 7 Mbps, 88% busy

Figure 2: Choosing APs greedily based on higher end-to-end available bandwidth is not optimal.



AP Bandwidth (Mbps)	AP1	AP2	AP3	AP4	AP5	AP6
End-to-end Available	1	1	1	1	1	4.5
Wireless Available	5	5	5	5	5	4.5

Figure 3: Choosing APs greedily based on higher wireless available bandwidth is not practical because it doesn't account for switching overheads.

APs and sticks to it, as is the case with current drivers, its throughput will be limited to 2 Mb/s. We observe however that the client need not spend 100% of its time at an AP to obtain its end-to-end available bandwidth. It is sufficient to connect to each AP for $\frac{2}{5}$ of the client's time. While connected, the client sends (and receives) its data at 5 Mb/s, i.e., according to its wireless available bandwidth. The AP drains the client's data upstream (or receives new data for the client) at the lower rate of 2 Mb/s, which is the end-to-end bandwidth available to our client. Until the AP drains the previous burst (or gets new data for the client), there is no point for the client to stay connected to the AP. As long as the client spends more than $\frac{2}{5}$ of its time on each AP, it can achieve the sum of their end-to-end rates, i.e., in our example it achieves 4 Mb/s.

Thus, to obtain the bandwidth available at an AP, a client should connect to it for at least a fraction $f_i = \frac{e}{w}$ of its time. This means that when the wireless link is the bottleneck at an AP, i.e., $w = e$, a client needs to spend 100% of its time at that AP in order to collect its available bandwidth. Otherwise, the client can use its spare time to get then unused bandwidth at other APs. But since the sum of the f_i's across all APs can exceed 1, a client will need to select a subset of the available APs. So, which APs does a client pick?

One may think of making greedy decisions. In particular, the client can order the APs according to their end-to-end available bandwidth, and greedily add APs to its schedule until the sum of the fractions f_i's reaches 1– i.e., 100% of the client's time is used up. Such a scheduler however is suboptimal. Fig. 2 shows a counter example, where AP1 has the highest end-to-end rate of 5Mb/s, yet picking AP1 means that the client has to spend $\frac{e_i}{w_i} = \frac{5}{5} = 100\%$ of its time at AP1 leaving no time to connect to other APs. The optimal scheduler here picks $\{$AP2, AP3$\}$ and achieves 7 Mb/s throughput; the client spends $\frac{e_i}{w_i} = \frac{4}{8} = 50\%$ of its time at AP2 and $\frac{3}{8} = 38\%$ at AP3 for a total of 88% of busy time. The remaining 12% of time can compensate for the switching overhead and increase robustness to inaccurate estimates of AP bandwidth.

In practice, one also cannot pick APs greedily based on their wireless available bandwidth. Consider the example in Fig. 3. One may think that the client should toggle between AP1, AP2, AP3, AP4, and AP5, spending 20% of its time on each AP. This would have been true if switching APs takes no time. In practice, switching between APs incurs a delay to reset the hardware to a different channel, to flush packets within the driver, etc., and this overhead adds up over the number of APs switched. Consider again the scenario in Fig. 3. Let the switching delay be 5 ms, then each time it toggles between 5 APs, the client wastes 25 ms of overhead. This switching overhead cannot be amortized away by switching infrequently between APs. To ensure that TCP connections via an AP do not time out, the client needs to serve each AP frequently, say once every 100ms. With a duty cycle of 100ms, and a switching overhead of 25ms a client has only 75% of its time left for useful work. Dividing this over the five APs results in a throughput of 5 × .75=3.25 Mb/s, which is worse than sticking to AP6 for 100% of the time, and obtaining 4.5 Mb/s.

In §3.1, we formalize and solve a scheduling problem that maximizes client throughput given practical constraints on switching overhead and the switching duty cycle.

3 FatVAP

FatVAP is an 802.11 driver design that aggregates the bandwidth available at nearby APs and load balances traffic across them. We implemented FatVAP as a modification to the MadWifi driver [4]. FatVAP incorporates the following three components:

- An AP scheduler that maximizes client throughput;
- A load balancer that maps traffic to APs according to their available bandwidth;
- An AP switching mechanism that is fast, loss-free, and transparent to both the APs and the host network stack.

At a high level, FatVAP works as follows. FatVAP scans the various channels searching for available access-points (APs). It probes these APs to estimate their wireless and end-to-end available bandwidths. FatVAP's scheduler decides which APs are worth connecting to and for how long in order to maximize client throughput. FatVAP then toggles connections to APs in accordance to the decision made by the scheduler. When switching away from an AP, FatVAP informs the AP that the client is entering the power-save mode. This ensures that the AP buffers the client's incoming packets, while it is away collecting traffic from another AP. Transparent to user's applications, FatVAP pins flows to APs in a way that balances their loads. FatVAP continually estimates the end-to-end and wireless available bandwidths at each

AP by passively monitoring ongoing traffic, and adapts to changes in available bandwidth by re-computing the best switching schedule.

3.1 The AP Scheduler

The scheduler chooses which APs to toggle between to maximize client throughput, while taking into account the bandwidth available at the APs and the switching overhead.

We formalize the scheduling problem as follows. The scheduler is given a set of accessible APs. It assigns to each AP a value and a cost. The value of connecting to a particular AP is its contribution to client throughput. If f_i is the fraction of time spent at AP$_i$, and w_i is AP$_i$'s wireless available bandwidth, then the value of connecting to AP$_i$ is:

$$value_i = f_i \times w_i. \tag{1}$$

Note that as discussed in §2, a client can obtain no more than the end-to-end available bandwidth at AP$_i$, e_i, and thus need not connect to AP$_i$ for more than $\frac{e_i}{w_i}$ of its time. Hence,

$$0 \leq f_i \leq \frac{e_i}{w_i} \Rightarrow value_i \leq e_i. \tag{2}$$

The cost of an AP is equal to the time that a client has to spend on it to collect its value. The cost also involves a setup delay to pick up in-flight packets and re-tune the card to a new channel. Note that the setup delay is incurred only when the scheduler spends a non-zero amount of time at AP$_i$. Hence, the cost of AP$_i$ is:

$$cost_i = f_i \times D + \lceil f_i \rceil \times s, \tag{3}$$

where D is the scheduler's duty cycle, i.e., the total time to toggle between all scheduled APs, s is the switching setup delay, and $\lceil f_i \rceil$ is the ceiling function, which is one if $f_i > 0$ and zero otherwise.

The objective of the scheduler is to maximize client throughput. The scheduler, however, cannot have too large a duty cycle. If it did, the delay can hamper the TCP connections, increasing their RTTs, causing poor throughput and potential time-outs. The objective of the scheduler is to pick the f_i's to maximize the switching value subject to two constraints: the cost in time must be no more than the chosen duty cycle, D, and the fraction of time at an AP has to be positive and no more than $\frac{e_i}{w_i}$, i.e.,

$$\max_{f_i} \quad \sum_i f_i w_i \tag{4}$$

$$\text{s.}t. \quad \sum_i (f_i D + \lceil f_i \rceil s) \leq D \tag{5}$$

$$0 \leq f_i \leq \frac{e_i}{w_i}, \ \forall i. \tag{6}$$

How do we solve this optimization? In fact, the optimization problem in Eqs. 4-6 is similar to the known knapsack problem [3]. Given a set of items, each with a value and a weight, we would like to pack a knapsack so as to maximize the total value subject to a constraint on the total weight. Our items (the APs) have both fractional weights (costs) $f_i \times D$ and zero-one weights $\lceil f_i \rceil \times s$. The knapsack problem is typically solved using dynamic programming. The formulation of this dynamic programming solution is well-known and can be used for our problem [3].

A few points are worth noting.

- FatVAP's solution based on dynamic programming is efficient and stays within practical bounds. Even with 5 APs, our implementation on a 2GHz x86 machine solves the optimization in 21 microseconds (see §4.2).
- So far we have assumed that we know both the wireless and end-to-end bandwidths of all accessible APs. FatVAP estimates these values passively (§3.1.1, §3.1.2).
- The scheduler takes AP load into account. Both the wireless and end-to-end bandwidths refer to the rate obtained by the client as it competes with other clients.
- It is important to include the switching overhead, s, in the optimization. This variable accounts for various overheads such as switching the hardware, changing the driver's state, and waiting for in-flight packets. It also ensures that the scheduler shies away from switching APs whenever a tie exists, or when switching does not yield a throughput increase. FatVAP continuously measures the switching delay and updates s if the delay changes (we show microbenchmarks in §4.2).
- Our default choice for duty cycle is $D = 100$ ms. This value is long enough to enable the scheduler to toggle a handful of APs and small enough to ensure that the RTTs of the TCP flows stay in a reasonable range [19].

3.1.1 Measuring Wireless Available Bandwidth

The wireless available bandwidth is the rate at which the client and AP communicate over the wireless link. If the client is the only contender for the medium, the wireless available bandwidth is the throughput of the wireless link. If other clients are contending for the medium, it reduces to the client's competitive share of the wireless throughput after factoring in the effect of auto-rate. Here, we describe how to estimate the wireless available bandwidth from client to the AP, i.e., on the uplink. One can have separate estimates for uplink and downlink. However, in our experience the throughput gain from this improved accuracy is small in comparison to the extra complexity.

How does a client estimate the uplink wireless available bandwidth? The client can estimate it by measuring the time between when a packet reaches the head of the transmit queue and when the packet is acked by the AP. This is the time taken to deliver one packet, td, given contention for the medium, autorate, retransmissions, etc. We estimate the available wireless bandwidth by dividing the packet's size in bytes, B, by its delivery time td. The client takes an exponentially weighted average over these measurements to smooth out variability, while adapting to changes in load and link quality.

Next, we explain how we measure the delivery time td. Note that the delivery time of packet j is:

$$td_j = ta_j - tq_j, \tag{7}$$

where tq_j is the time when packet j reaches the head of the transmit queue, and ta_j is the time when packet j is acked. It is easy to get ta_j because the Hardware Abstraction Layer (HAL) timestamps each transmitted packet with the time it was acked. Note that the HAL does raise a tx interrupt to tell the driver to clean up the resources of transmitted packets but it does this only after many packets have been transmitted. Hence, the time when the tx interrupt is raised is a poor estimate of ta_j.

Obtaining tq_j, however, is more complex. The driver hands the packet to the HAL, which queues it for transmission. The driver does not know when the packet reaches the head of the transmission queue. Further, we do not have access to the HAL source, so we cannot modify it to export the necessary information.[1] We work around this issue as follows. We make the driver timestamp packets just before it hands them to the HAL. Suppose the timestamp of packet j as it is pushed to the HAL is th_j, we can then estimate tq_j as follows:

$$tq_j = \max(th_j, ta_{j-1}) \tag{8}$$

The intuition underlying Eq. 8 is simple; either the HAL's queue is empty and thus packet j reaches the head of the queue soon after it is handed to the HAL, i.e., at time th_j, or the queue has some previous packets, in which case packet j reaches the head of the queue only when the HAL is done with delivering packet $j-1$, i.e., at time ta_{j-1}.

Two practical complications exist however. First, the timer in the HAL has millisecond accuracy. As a result, the estimate of the delivery time td in Eq. 7 will be equally coarse, and mostly either 0 ms or 1 ms. To deal with this coarse resolution, we need to aggregate over a large number of measurements. In particular, FatVAP produces a measurement of the wireless available throughput at AP$_i$ by taking an average over a window of T seconds (by default $T = 2$s), as follows:

$$w_i = \frac{\sum_{j \in T} B_j}{\sum_{j \in T} td_j}. \tag{9}$$

The scheduler continuously updates its estimate by using an exponentially weighted average over the samples in Eq. 9.

A second practical complication occurs because both the driver's timer and the HAL's timer are typically synchronized with the time at the AP. This synchronization happens with every beacon received from the AP. But as FatVAP switches APs, the timers may resynchronize with a different AP. This is fine in general as both timers are always synchronized with respect to the same AP. The problem, however, is that some of the packets in the transmit queue may have old timestamps taken with respect to the previous AP. To deal with this issue, the FatVAP driver remembers the id of the last packet that was pushed into the HAL. When resynchronization occurs (i.e., the beacon is received), it knows that packets with ids smaller than or equal to the last pushed packet have inaccurate timestamps and should not contribute to the average in Eq. 9.

Finally, we note that FatVAP's estimation of available bandwidth is mostly passive and leverages transmitted data packets. FatVAP uses probes only during initialization, because at that point the client has no traffic traversing the AP. FatVAP also occasionally probes the unused APs (i.e., APs not picked by the scheduler) to check that their available bandwidth has not changed.

3.1.2 Measuring End-to-End Available Bandwidth

The available end-to-end bandwidth via an AP is the average throughput that a client obtains when using the AP to access the Internet.[2] The available end-to-end bandwidth is lower when there are more contenders causing a FatVAP client to avoid congested APs in favor of a balanced load.

How do we measure an AP's end-to-end available bandwidth? The naive approach would count all bytes received from the AP in a certain time window and divide the count by the window size. The problem, however, is that no packets might be received either because the host has not demanded any, or the sending server is idle. To avoid underestimating the available bandwidth, FatVAP guesses which of the inter-packet gaps are caused by idleness and removes those gaps. The algorithm is fairly simple. It ignores packet gaps larger than one second. It also ignores gaps between small packets, which are mostly AP beacons and TCP acks, and focuses on the spacing between pairs of large packets. After ignoring packet pairs that include small packets and those that are spaced by excessively long intervals, FatVAP computes an estimate

[1] An open source project named OpenHAL allows access to the HAL but is too inefficient to be used in practice.

[2] Note that our definition of available end-to-end bandwidth is not the typical value [17, 26] that is computed between a source-destination pair, but is an average over all paths through the AP.

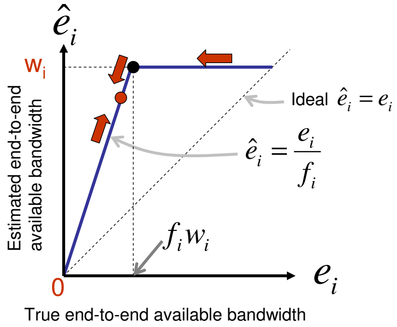


Figure 4: The estimate of end-to-end available bandwidth is different from the true value because the AP buffers data when the client is not listening and buffered data drains at the wireless available bandwidth. FatVAP corrects for this by spending slightly longer than necessary at each AP, i.e., operating at the red dot rather than the black dot.

of the end-to-end available bandwidth at AP$_i$ as:

$$\widehat{e_i} = \frac{\sum B_j}{\sum g_j}, \tag{10}$$

where B_j is the size of the second packet in the j^{th} pair, and g_j is the gap separating the two packets, and the sum is taken over a time window of $T = 2$ seconds.

One subtlety remains however. When a client reconnects to an AP, the AP first drains out all packets that it buffered when the client was away. These packets go out at the wireless available bandwidth w_i. Once the buffer is drained out, the remaining data arrives at the end-to-end available bandwidth e_i. Since the client receives a portion of its data at the wireless available bandwidth and $w_i \geq e_i$, simply counting how quickly the bytes are received, as in Eq. 10, over-estimates the end-to-end available bandwidth.

Fig. 4 plots how the estimate of end-to-end available bandwidth $\widehat{e_i}$ relates to the true value e_i. There are two distinct phases. In one phase, the estimate is equal to w_i, which is shown by the flat part of the solid blue line. This phase corresponds to connecting to AP$_i$ for less time than needed to collect all buffered data, i.e., $f_i < \frac{e_i}{w_i}$. Since the buffered data drains at w_i, the estimate will be $\widehat{e_i} = w_i$. In the other phase, the estimate is systematically inflated by $\frac{1}{f_i}$, as shown by the tilted part of the solid blue line. This phase corresponds to connecting to AP$_i$ for more time than needed to collect all buffered data, i.e., $f_i > \frac{e_i}{w_i}$. The derivation for this inflation is in Appendix A. Here, we note the ramifications.

Inflated estimates of the end-to-end available bandwidth make the ideal operating point unstable. A client would ideally operate at the black dot in Fig. 4, where it connects to AP$_i$ for exactly $f_i^* = \frac{e_i}{w_i}$ of its time. But, if the client does so, the estimate $\widehat{e_i}$ will be $\widehat{e_i} = \frac{e_i}{f_i^*} = w_i$. In this case, the client cannot figure out the amount of inflation in e_i and compensate for it because the true end-to-end available bandwidth can be any value corresponding to the flat thick blue line in Fig. 4. Even worse, if the actual end-to-end available bandwidth were to increase, say because a contending client shuts off, the client cannot observe this change, because its estimate will still be w_i.

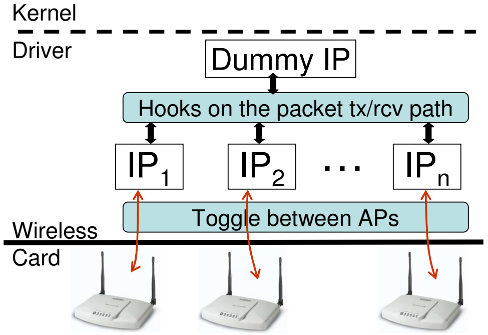
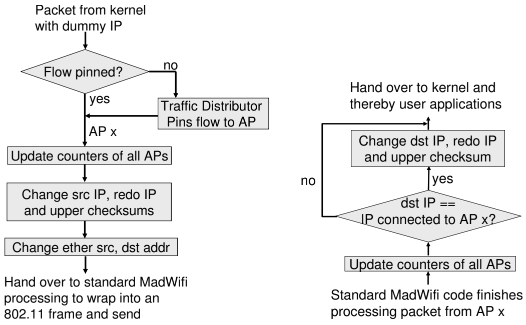


Figure 5: FatVAP's reverse NAT architecture.


Packet from kernel with dummy IP
Flow pinned?
no
yes
Traffic Distributor Pins flow to AP
AP x
Update counters of all APs
Change src IP, redo IP and upper checksums
Change ether src, dst addr
Hand over to standard MadWifi processing to wrap into an 802.11 frame and send
Hand over to kernel and thereby user applications
Change dst IP, redo IP and upper checksum
yes
no
dst IP == IP connected to AP x?
Update counters of all APs
Standard MadWifi code finishes processing packet from AP x

(a) Send Side

(b) Receive Side

Figure 6: Getting packets to flow over multiple interfaces.

To fix this, FatVAP clients operate at the red dot, i.e., they spend slightly longer than necessary at each AP in order to obtain an accurate estimate of the end-to-end bandwidth. Specifically, if $\widehat{e_i} \approx w_i$, FatVAP knows that it is operating near or beyond the black dot and thus slightly increases f_i to go back to the red dot. The red arrows in the figure show how a FatVAP client gradually adapts its f_i to bring it closer to the desired range. As long as f_i is larger than the optimal value, we can compensate for the inflation knowing that $e_i = f_i \widehat{e_i}$, i.e., Eq. 10 can be re-written as:

$$e_i = f_i \frac{\sum B_j}{\sum g_j}. \tag{11}$$

3.2 Load Balancing Traffic Across APs

The scheduler in §3.1 gives an opportunity to obtain the sum of available bandwidth at all APs, but to fulfill that opportunity, the FatVAP driver should map traffic to APs appropriately. There are two parts to mapping traffic: a load balancer that splits traffic among the APs, and a reverse-NAT that ensures traffic goes through the desired APs.

3.2.1 The Load Balancer

The load balancer assigns traffic to APs proportionally to the end-to-end bandwidth obtainable from an AP.

Thus, the traffic ratio assigned to each AP, r_i, is:

$$r_i = \frac{f_i w_i}{\sum_j f_j w_j}, \tag{12}$$

where f_i is the fraction of time that the client connects to AP_i and $f_i w_i$ is the value of AP_i (see Eqs. 1, 2).

When splitting traffic, the first question is whether the traffic allocation unit should be a packet, a flow, or a destination? FatVAP allocates traffic to APs on a flow-by-flow basis. A flow is identified by its destination IP address and its ports. FatVAP records the flow-to-AP mapping in a hash-table. When a new flow arrives, FatVAP decides which AP to assign this flow to and records the assignment in the hash table. Subsequent packets in the flow are simply sent through the AP recorded in the hash table.

Our decision to pin flows to APs is driven by practical considerations. First, it is both cumbersome and inefficient to divide traffic at a granularity smaller than a flow. Different APs usually use different DHCP servers and accept traffic only when the client uses the IP address provided by the AP's DHCP server. This means that in the general case, a flow cannot be split across APs. Further, splitting a TCP flow across multiple paths often reorders the flow's packets hurting TCP performance [25]. Second, a host often has many concurrent flows, making it easy to load balance traffic while pinning flows to APs. Even a single application can generate many flows. For example, browsers open parallel connections to quickly fetch the objects in a web page (e.g., images, scripts) [18], and file-sharing applications like BitTorrent open concurrent connections to peers.

But, how do we assign flows to APs to satisfy the ratios in Eq.12? The direct approach assigns a new flow to the ith AP with a random probability r_i. Random assignment works when the flows have similar sizes. But flows vary significantly in their sizes and rates [15, 22, 25]. To deal with this issue, FatVAP maintains per-AP token counters, C, that reflect the deficit of each AP, i.e., how far the number of bytes mapped to an AP is from its desired allocation. For every packet, FatVAP increments all counters proportionally to the APs' ratios in Eq. 12. The counter of the AP that the packet was sent/received on is decremented by the packet size B. Hence, every window of T_c seconds (default is $T = 60$s) we compute:

$$C_i = \begin{cases} C_i + r_i \times B - B & \text{Packet is mapped to AP}_i \\ C_i + r_i \times B & \text{Otherwise.} \end{cases} \tag{13}$$

It is easy to see that APs with more traffic than their fair share have negative counters and those with less than their fair share have positive counter values. When a new flow arrives, FatVAP assigns the flow to the AP with the most positive counters and decreases that AP's counters by a constant amount F (default $10,000$) to accommodate for TCP's slow ramp-up. Additionally, we decay all counters every $T_c = 60$s to forget biases that occurred a long time ago.

3.2.2 The Reverse-NAT

How do we ensure that packets in a particular flow are sent and received through the AP that the load balancer assigns the flow to? If we simply present the kernel with multiple interfaces, one interface per AP like prior work [13], the kernel would send all flows through one AP. This is because the kernel maps flows to interfaces according to routing information, not load. When all APs have valid routes, the kernel simply picks the default interface.

To address this issue, FatVAP uses a reverse NAT as a shim between the APs and the kernel, as shown in Fig. 5. Given a single physical wireless card, the FatVAP driver exposes just one interface with a dummy IP address to the kernel. To the rest of the MadWifi driver, however, FatVAP pretends that the single card is multiple interfaces. Each of the interfaces is associated to a different AP, using a different IP address. Transparent to the host kernel, FatVAP resets the addresses in a packet so that the packet can go through its assigned AP.

On the send side, and as shown in Fig. 6, FatVAP modifies packets just as they enter the driver from the kernel. If the flow is not already pinned to an AP, FatVAP uses the load balancing algorithm above to pin this new flow to an AP. FatVAP then replaces the source IP address in the packet with the IP address of the interface that is associated with the AP. Of course, this means that the IP checksum has to be re-done. Rather than recompute the checksum of the entire packet, FatVAP uses the fact that the checksum is a linear code over the bytes in the packet. So analogous to [14], the checksum is recomputed by subtracting some $f(\text{the dummy IP address})$ and adding $f(\text{assigned interface's IP})$. Similarly, transport layer checksums, e.g., TCP and UDP checksums, need to be redone as these protocols use the IP header in their checksum computation. After this, FatVAP hands over the packet to standard MadWifi processing, as if this were a packet the kernel wants to transmit out of the assigned interface.

On the receive side, FatVAP modifies packets after standard MadWifi processing, just before they are handed up to the kernel. If the packet is not a broadcast packet, FatVAP replaces the IP address of the actual interface the packet was received on with the dummy IP of the interface the kernel is expecting the packets on. Checksums are re-done as on the send side, and the packet is handed off to the kernel.

3.3 Fast, Loss-Free, and Transparent AP Switching

To maximize user throughput, FatVAP has to toggle between APs according to the scheduler in §3.1 while simultaneously maintaining TCP flows through multiple APs (see §3.2). Switching APs requires switching the HAL and potentially resetting the wireless channel. It also requires managing queued packets and updating the driver's state. These tasks take time. For example, the Microsoft virtual WiFi project virtualizes an 802.11 card, allowing it to switch from one AP to another. But this switching takes 30-600 ms [7] mostly because a new driver module needs to be initialized when switching to a new AP. Though successful in its objective of bridging wireless networks, the design of Virtual WiFi is not sufficient to aggregate AP bandwidth. FatVAP needs to support fast AP switching, i.e., a few milliseconds, otherwise the switching overhead may preclude most of the benefits. Further, switching should not cause packet loss. If the card or the AP loses packets in the process, switching will hurt TCP traffic [25]. Finally, most of the switching problems would be easily solved if one can modify both APs and clients. Such a design, however, will not be useful in today's 802.11 deployments.

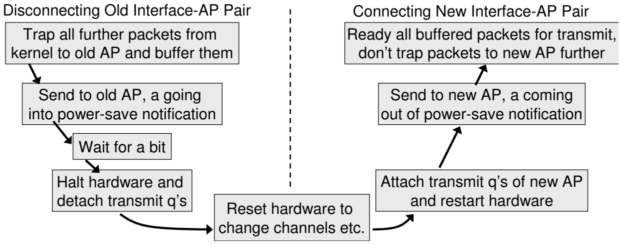
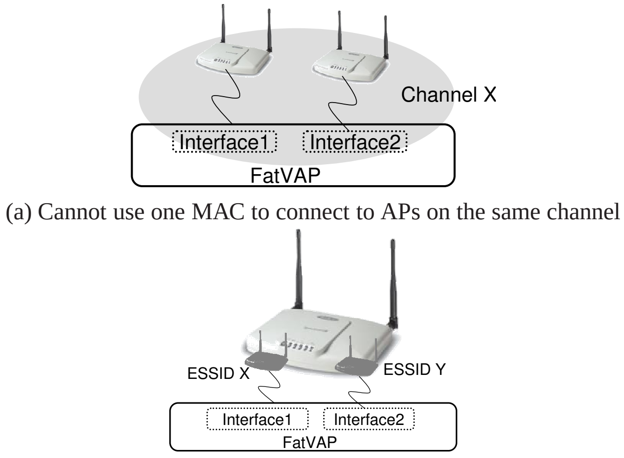

Disconnecting Old Interface-AP Pair
Connecting New Interface-AP Pair
Trap all further packets from kernel to old AP and buffer them
Send to old AP, a going into power-save notification
Wait for a bit
Halt hardware and detach transmit q's
Reset hardware to change channels etc.
Attach transmit q's of new AP and restart hardware
Send to new AP, a coming out of power-save notification
Ready all buffered packets for transmit, don't trap packets to new AP further

Figure 7: FatVAP's approach to switching between interfaces



(a) Cannot use one MAC to connect to APs on the same channel

(b) No benefit in connecting to multiple light-weight APs

Figure 8: Challenges in transparently connecting to multiple APs.

3.3.1 Fast and Loss-Free Switching

The basic technique that enables a card to toggle between APs is simple and is currently used by the MadWiFi [4] driver to background scan for better APs and others. Before a client switches away from an AP, it tells the AP that it is going to power save mode. This causes the AP to buffer the client's packets for the duration of its absence. When the client switches again to the AP, it sends the AP a frame to inform the AP of its return, and the AP then, forwards the buffered packets.

So, how do we leverage this idea for quickly switching APs without losing packets? Two fundamental issues need to be solved. First, when switching APs, what does one do with packets inside the driver destined for the old AP? An AP switching system that sits outside the driver, like MultiNet [13] has no choice but to wait until all packets queued in the driver are drained, which could take a while. Systems that switch infrequently, such as MadWifi that does so to scan in the background, drop all the queued packets. To make AP switching fast and loss-free, FatVAP pushes the switching procedure to the driver, where it maintains multiple transmit queues, one for each interface. Switching APs simply means detaching the old AP's queue and reattaching the new AP's queue. This makes switching a roughly constant time operation and avoids dropping packets. It should be noted that packets are pushed to the transmit queue by the driver and read by the HAL. Thus, FatVAP still needs to wait to resolve the state of the head of the queue. This is, however, a much shorter wait (a few milliseconds) with negligible impact on TCP and the scheduler.

Second, how do we maintain multiple 802.11 state machines simultaneously within a single driver? Connecting with an AP means maintaining an 802.11 state machine. For example, in 802.11, a client transitions from INIT to SCAN to AUTH to ASSOC before reaching RUN, where it can forward data through the AP. It is crucial to handle state transitions correctly because otherwise no communication may be possible. For example, if an association request from one interface to its AP is sent out when another interface is connected to its AP, perhaps on a different channel, the association will fail preventing further communication. To maintain multiple state machines simultaneously, FatVAP adds hooks to MadWifi's 802.11 state-machine implementation. These hooks trap all state transitions in the driver. Only transitions for the interface that is currently connected to its AP can proceed, all other transitions are held pending and handled when the interface is scheduled next. Passive changes to the state of an interface such as receiving packets or updating statistics are allowed at all times.

Fig. 7 summarizes the FatVAP drivers' actions when switching from an *old* interface-AP pair to a *new* pair.

- First, FatVAP traps all future packets handed down by the kernel that need to go out to the old AP and buffers them until the next time this interface-AP pair is connected.

- Second, FatVAP sends out an 802.11 management frame indicating to the old AP that the host is going into power save mode. The AP then buffers all future packets that need to go to the host.
- Unfortunately, these above two cases do not cover packets that may already be on-the-way, i.e., packets might be in the card's transmit queue waiting to be sent or might even be in the air. To prevent packet loss, FatVAP waits a little bit for the current packet on the air to be received before halting the hardware. FatVAP also preserves the packets waiting in the interface's transmit queue. The transmit queue of the old interface is simply detached from the HAL and is re-attached when the interface is next scheduled.
- Fourth, FatVAP resets the hardware settings of the card and pushes the new association state into the HAL. If the new AP is on a different channel, the card changes channels and listens at the new frequency band.
- Finally, waking up the new interface is simple as the hardware is now on the right channel. FatVAP sends out a management frame telling the new AP that the host is coming out of power save, the AP immediately starts forwarding buffered packets to the host.

3.3.2 Transparent Switching

We would like FatVAP to work with unmodified APs. Switching APs transparently involves handling these practical deployment scenarios.

(a) Cannot Use a Single MAC Address: When two APs are on the same 802.11 channel (operate in the same frequency band), as in Fig. 8a, you cannot connect to both APs with virtual interfaces that have the same MAC address. To see why this is the case, suppose the client uses both AP1 and AP2 that are on the same 802.11 channel. While exchanging packets with AP2, the client claims to AP1 that it has gone into the power-save mode. Unfortunately, AP1 overhears the client talking to AP2 as it is on the same channel, concludes that the client is out of power-save mode, tries to send the client its buffered packets and when un-successful, forcefully deauthenticates the client.

FatVAP confronts MAC address problems with an existing feature in many wireless chipsets that allows a physical card to have multiple MAC addresses [4]. The trick is to change a few of the most significant bits across these addresses so that the hardware can efficiently listen for packets on all addresses. But, of course, the number of such MAC addresses that a card can fake is limited. Since the same MAC address can be reused for APs that are on different channels, FatVAP creates a pool of interfaces, half of which have the primary MAC, and the rest have unique MACs. When FatVAP assigns a MAC address to a virtual interface, it ensures that interfaces connected to APs on the same channel do not share the MAC address.

(b) Light-Weight APs (LWAP): Some vendors allow a physical AP to pretend to be multiple APs with different ESSIDs and different MAC addresses that listen on the same channel, as shown in Fig. 8b. This feature is often used to provide different levels of security (e.g., one light-weight AP uses WEP keys and the other is open) and traffic engineering (e.g., preferentially treat authenticated traffic). For our purpose of aggregating AP bandwidth, switching between light weight APs is useless as the two APs are physically one AP.

FatVAP uses a heuristic to identify light-weight APs. LWAPs that are actually the same physical AP share many bits in their MAC addresses. FatVAP connects to only one AP from any set of APs that have fewer than five bits different in their MAC addresses.

4 Evaluation

We evaluate our implementation of FatVAP in the Madwifi driver in an internal testbed we built with APs from Cisco and Netgear, in hotspots served by commercial providers like T-Mobile, and in residential areas which have low-cost APs connected to DSL or cable modem backends.

Our results reveal three main findings.

- In the testbed, FatVAP performs as expected. It balances load across APs and aggregates their available backhaul bandwidth, limited only by the wireless capacity and application demands. This result is achieved even when the APs are on different wireless channels.
- In today's residential and Hotspot deployments (in Cambridge/Somerville, MA), FatVAP delivers to the end user a median throughput gain of 2.6x, and reduces the median response time by 2.8x.
- FatVAP safely co-exists with unmodified drivers and other FatVAP clients. At each AP, FatVAP competes with unmodified clients as fairly as an unmodified MadWifi driver, and is sometimes fairer because FatVAP moves away from congested APs. FatVAP clients are also fair among themselves.

4.1 Experimental Setup

(a) Drivers We compare the following two drivers.

- *Unmodified Driver:* This refers to the madwifi v0.9.3 [4] driver. On linux, MadWifi is the current *defacto* driver for Atheros chipsets and is a natural baseline.
- *FatVAP:* This is our implementation of FatVAP as an extension of madwifi v0.9.3. Our implementation includes the features described in §3, and works in con-

Operation	Time (μs)	
	Mean	STD
IP Checksum Recompute	0.10	0.09
TCP/UDP Checksum Recompute	0.12	0.14
Flow Lookup/Add in HashTable	2.52	2.30
Running the Scheduler	16.21	4.85
Switching Delay	2897.48	2780.71

Table 1: Latency overhead of various FatVAP operations.

junction with autorate algorithms, carrier-sense, CTS-to-self protection, etc.

(b) Access Points: Our testbed uses Cisco Aironet 1130AG Series access points and Netgear's lower-cost APs. We put the testbed APs in the 802.11a band so as to not interfere with our lab's existing infrastructure. Our outside experiments run in hotspots and residential deployments and involve a variety of commercial APs in the 802.11b/g mode, which shows that FatVAP works across 802.11a/b/g. The testbed APs can buffer up to 200 KB for a client that enters the power-save mode.[3] Testbed APs are assigned different 802.11a channels (we use channels 40, 44, 48, 52, 56 and 60). The wireless throughput to all APs in our testbed is in the range $[19-22]$ Mb/s. The actual value depends on the AP, and differs slightly between uplink and downlink scenarios. APs in hotspots and residential experiments have their own channel assignment which we do not control.

(c) Wireless Clients: We have tested with a few different wireless cards, from the Atheros chipsets in the latest Thinkpads (Atheros AR5006EX) to older Dlink and Netgear cards. Clients are 2GHz x86 machines that run Linux v2.6. In each experiment, we make sure that FatVAP and the compared unmodified driver use similar machines with the same kernel version/revision and the same card.

(d) Traffic Shaping: To emulate an AP backhaul link, we add a traffic shaper behind each of our test-bed APs. This shaper is a Linux box that bridges the APs traffic to the Internet and has two Ethernet cards, one of which is plugged into the lab's (wired) GigE infrastructure, and the other is connected to the AP. The shaper controls the end-to-end bandwidth through a token bucket based rate-filter whose rate determines the capacity of AP's access link. We use the same access capacity for both downlink and uplink.

(e) Traffic Load: All of our experiments use TCP. A FatVAP client assigns traffic to APs at the granularity of a TCP flow as described in §3.2. An unmodified client assigns traffic to the single AP chosen by its unmodified driver [4]. Each experiment uses one these traffic loads.

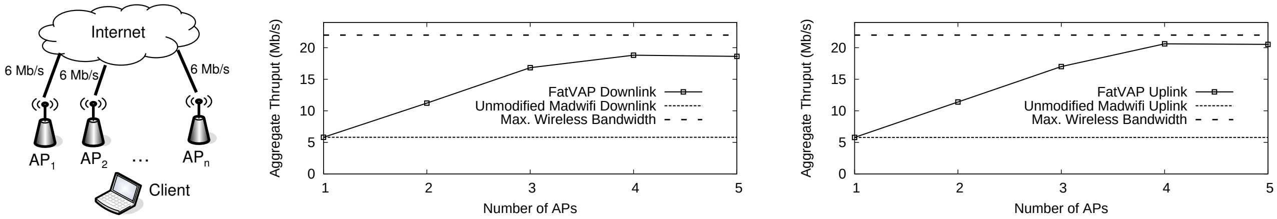
- *Long-lived iperf TCP flows:* In this traffic load, each client has as many parallel TCP flows as there are APs. Flows are generated using iperf [2] and each flow lasts for 5 minutes.
- *Web Traffic:* This traffic load mimics a user browsing the Web. The client runs our modified version of WebStone 2.5 [8] a tool for benchmarking Web servers. Requests for new Web pages arrive as a Poisson process with a mean of 2 pages/s, the number of objects on a page is exponentially distributed with a mean of 20 objects/page, the objects themselves are copies of actual content on the CSAIL Web server and have sizes that are roughly a power-law with mean equal to 15KB. Note that popular browsers usually open multiple parallel connections to the same server or different servers to quickly download the various objects on a web page (e.g., images, scripts) [18].
- *BitTorrent:* Here, we use the Azureus [1] BitTorrent client to fetch a 500MB file. The tracker is on a CSAIL machine, and 8 Planetlab nodes act as peers. Note that BitTorrent fetches data in parallel from multiple peers.

4.2 Microbenchmarks

To profile the various components of FatVAP, we use the x86 rdtscll instruction for fine-grained timing information. rdtscll reads a hardware timestamp counter that is incremented once every CPU cycle. On our 2 GHz client, this yields a resolution of 0.5 nano seconds.

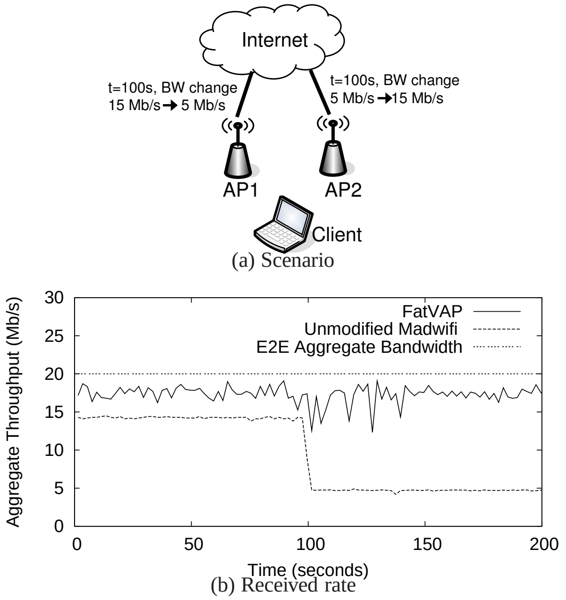
Table 1 shows our microbenchmarks. The table shows that the delay seen by packets on the fast-path (e.g., flow lookup to find which AP the packets need to go to, recomputing checksums) is negligible. Similarly, the overhead of computing and updating the scheduler is minimal. The bulk of the overhead is caused by AP switching. It takes an average of 3 ms to switch from one AP to another. This time includes sending a power save frame, waiting until the HAL has finished sending/receiving the current packet, switching both the transmit and receive queues, switching channel/AP, and sending a management frame to the new AP informing it that the client is back from power save mode. The standard deviation is also about 3 ms, owing to the variable amount of pending interrupts that have to be picked up. Because FatVAP performs AP switching in the driver, its average switching delay is much lower than prior systems (3ms as opposed to 30-600ms). We note that switching cost directly affects the throughput a user can get. A user switching between two APs every 100ms, would only have 40ms of usable time left if each switch takes 30ms, as opposed to 94ms of usable time when each switch takes 3ms and hence can more than double his throughput (94% vs. 40% use).

[3] We estimate this value by computing the maximum burst size that a client obtains when it re-connects after spending a long time in the power-save mode.



(a) Scenario (b) Uplink Aggregation (c) Downlink Aggregation

Figure 9: FatVAP aggregates AP backhaul bandwidth until the limit of the card's wireless bandwidth, i.e., wireless link capacity − switching overhead.



(a) Scenario

(b) Received rate

Figure 10: At time $t = 100$s, the available bandwidth at the first access link changes from 15Mb/s to 5Mb/s, whereas the available bandwidth at the second access link changes from 5Mb/s to 15Mb/s. FatVAP quickly rebalances the load and continues to deliver the sum of the APs' available end-to-end bandwidth. In the scenario, an unmodified driver limits the client to AP1's available bandwidth.

4.3 Can FatVAP Aggregate AP Backhaul Rates?

FatVAP's main goal is to allow users in a hotspot or at home to aggregate the bandwidth available at all accessible APs. Thus, in this section we check whether FatVAP can achieve this goal.

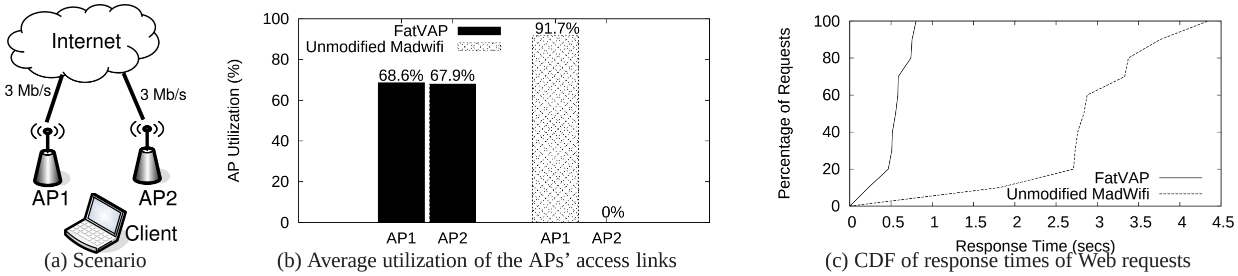
Our experimental setup shown in Fig. 9(a) has $n = \{1, 2, 3, 4, 5\}$ APs. The APs are on different channels and each AP has a relatively thin access link to the Internet (capacity 6Mb/s), which we emulate using the traffic shaper described in §4.1(c). The wireless bandwidth to the APs is about $[20 - 22]$Mb/s. The traffic constitutes of long-lived iperf TCP flows, and there are as many TCPs as APs, as described in §4.1(d). Each experiment is first performed by FatVAP, then repeated with an unmodified driver. We perform 20 runs and compute the average throughput across them. The question we ask is: does FatVAP present its client with a fat virtual AP, whose backhaul bandwidth is the sum of the AP's backhaul bandwidths?

Figs. 9(b) and 9(c) show the aggregate throughput of the FatVAP client both on the uplink and downlink, as a function of the number of APs that FatVAP is allowed to access. When FatVAP is limited to a single AP, the TCP throughput is similar to running the same experiment with an unmodified client. Both throughputs are about 5.8Mb/s, slightly less than the access capacity because of TCP's sawtooth behavior. But as FatVAP is given access to more APs, its throughput doubles, and triples. At 3 APs, FatVAP's throughput is 3 times larger than the throughput of the unmodified driver. As the number of APs increases further, we start hitting the maximum wireless bandwidth, which is about 20-22Mb/s. Note that FatVAP's throughput stays slightly less than the maximum wireless bandwidth due to the time lost in switching between APs. FatVAP achieves its maximum throughput when it uses 4 APs. In fact, as a consequence of switching overhead, FatVAP chooses not to use the fifth AP even when allowed access to it. Thus, one can conclude that FatVAP effectively aggregates AP backhaul bandwidth up to the limitation imposed by the maximum wireless bandwidth.

4.4 Can FatVAP Adapt to Changes in Bandwidth?

Next, if an AP's available bandwidth changes, we would like FatVAP to re-adjust and continue delivering the sum of the bandwidths available across all APs. Note that an unmodified MadWifi cannot respond to changes in backhaul capacity. On the other hand, FatVAP's constant estimation of both end-to-end and wireless bandwidth allows it to react to changes within a couple of seconds. We demonstrate this with the experiment in Fig. 10, where two APs are bottlenecked at their access links. As before, the APs are on two different channels, and the bandwidth of the wireless links to the APs is about [21-22]Mb/s. At the beginning, AP1 has 15Mb/s of available bandwidth, whereas AP2 has only 5Mb/s. At time $t = 100$s, we change the available bandwidth at the two APs, such that AP1 has only 5Mb/s and AP2 has 15 Mb/s. Note that since the aggregated available bandwidth remains the same, FatVAP should deliver constant throughput across this change. We perform the experiment with a FatVAP client, and repeat it with an unmodified client that connects to AP1 all the time. In both cases, the client uses iperf [2] to generate large TCP flows, as described in §4.1(d).

Fig. 10(b) shows the client throughput, averaged over



(a) Scenario (b) Average utilization of the APs' access links (c) CDF of response times of Web requests

Figure 11: Because it balances the load across the two APs, FatVAP achieves a significantly lower response time for Web traffic in comparison with an unmodified driver.

2s intervals, as a function of time. The figure shows that if the client uses an unmodified driver connected to AP1, its throughput will change from 15Mb/s to 5Mb/s in accordance with the change in the available bandwidth on that AP. FatVAP, however, achieves a throughput of about 18 Mb/s, and is limited by the sum of the APs' access capacities rather than the access capacity of a single AP. FatVAP also adapts to changes in AP available bandwidth, and maintains its high throughput across such changes.

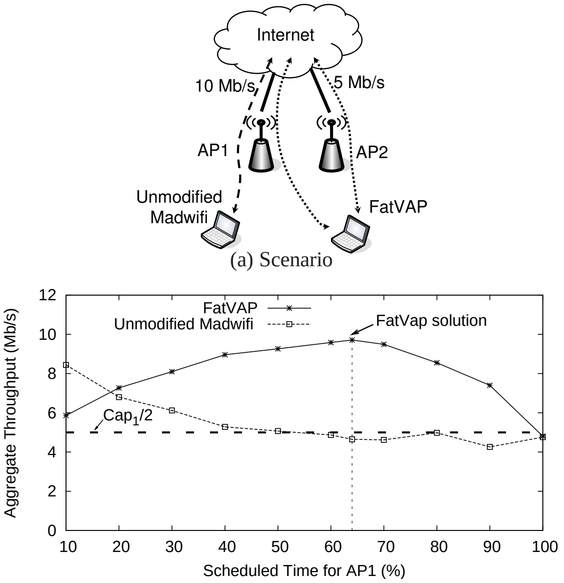
4.5 Does FatVAP Balance the Load across APs?

A second motivation in designing FatVAP is to balance load among nearby APs. To check that FatVAP indeed balances AP load, we experiment with two APs and one client, as shown in Fig. 11(a). We emulate a user browsing the Web. Web sessions are generated using WebStone 2.5, a benchmarking tool for Web servers [8] and fetch Web pages from a Web server that mirrors our CSAIL Web-server, as described in §4.1(d).

Fig. 11(b) shows that FatVAP effectively balances the utilization of the APs' access links, whereas the unmodified driver uses only one AP, congesting its access link. Fig. 11(c) shows the corresponding response times. It shows that FatVAP's ability to balance the load across APs directly translates to lower response times for Web requests. While the unmodified driver congests the default APs causing long queues, packet drops, TCP timeouts, and thus long response times, FatVAP balances AP loads to within in a few percent, preventing congestion, and resulting in much shorter response times.

4.6 Does FatVAP Compete Fairly with Unmodified Drivers?

We would like to confirm that regardless of how FatVAP schedules APs, a competing unmodified driver would get its fair share of bandwidth at an AP. We run the experiment in Fig. 12(a), where a FatVAP client switches between AP1 and AP2, and shares AP1 with an unmodified driver. In each run, both clients use iperf to generate long-lived TCP flows, as described in §4.1(d). For the topology in Fig. 12(a), since AP1 is shared by two clients, we have $w_1 = 19/2 = 9.5$Mb/s, and $e_1 = 10/2 = 5$Mb/s. AP2 is not shared, and has $w_2 = 20$Mb/s, and $e_2 = 5$Mb/s.



(a) Scenario

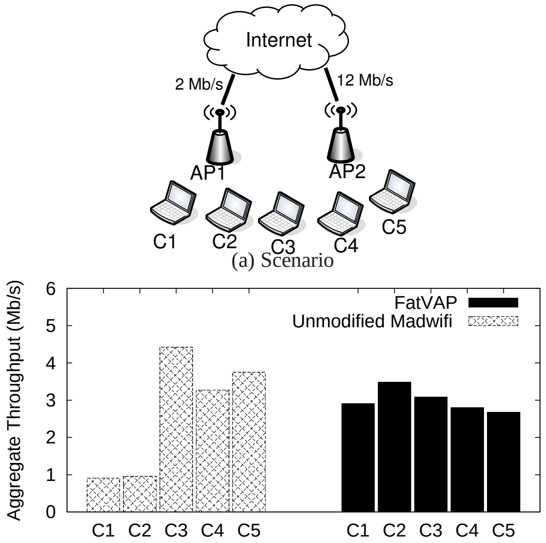
(b) Throughput of FatVAP and a competing unmodified driver

Figure 12: FatVAP shares the bandwidth of AP1 fairly with the unmodified driver. Regardless of how much time FatVAP connects to AP1, the unmodified driver gets half of AP1's capacity, and sometimes more. The results are for the downlink. The uplink shows a similar behavior.

Fig. 12(b) plots the throughput of both FatVAP and an unmodified driver when we impose different time-sharing schedules on FatVAP. For reference, we also plot a horizontal line at 5Mb/s, which is one half of AP1's access capacity. The figure shows that regardless of how much time FatVAP connects to AP1, it always stays fair to the unmodified driver, that is, it leaves the unmodified driver about half of AP1's capacity, and sometimes more. FatVAP achieves the best throughput when it spends about 55-70% of its time on AP1. Its throughput peaks when it spends about 64% of its time on AP1, which is, in fact, the solution computed by our scheduler in §3.1 for the above bandwidth values. This shows that our AP scheduler is effective in maximizing client throughput.

4.7 Are FatVAP Clients Fair Among Themselves?

When unmodified clients access multiple APs the aggregate bandwidth is divided at the coarse granularity of a



(a) Scenario

(b) Average throughput of five clients contending for two APs.

Figure 13: FatVAP clients compete fairly among themselves and have a fairer throughput allocation than unmodified clients under the same conditions.

client. This causes significant unfairness between clients that use different APs. The situation is further aggravated since unmodified clients pick APs based on signal strength rather than available bandwidth, and hence can significantly overload an AP.

Here, we look at 5 clients that compete for two APs, where AP1 has 2Mb/s of available bandwidth and AP2 has 12 Mb/s, as shown in Fig. 13(a). The traffic load consists of long-lived iperf TCP flows, as described in §4.1(d). Fig. 13(b) plots the average throughput of clients with and without FatVAP. With an unmodified driver, clients C1 and C2 associate with AP1, thereby achieving a throughput of less than 1Mb/s. The remaining clients associate with AP2 for roughly 4Mb/s throughput for each. However, FatVAP's load balancing and fine-grained scheduling allow all five clients to fairly share the aggregate bandwidth of 14 Mb/s, obtaining a throughput of roughly 2.8 Mb/s each, as shown by the dark bars in Fig. 13(b).

4.8 FatVAP in Residential Deployments

We demonstrate that FatVAP can bring real and immediate benefits in today's residential deployments. To do so, we experiment with FatVAP in three residential locations in Cambridge, MA, shown in Fig. 14. Each of these locations has two APs, and all of them are homes of MIT students, where neighbors are interested in combining the bandwidth of their DSL lines. Again, in these experiments, we run Web sessions that access a mirror of the CSAIL Web-server, as explained in §4.1(d). In each location, we issue Web requests for 10 min, and repeat the experiment with and without FatVAP.

Fig. 15(a) plots the CDF of throughput taken over all

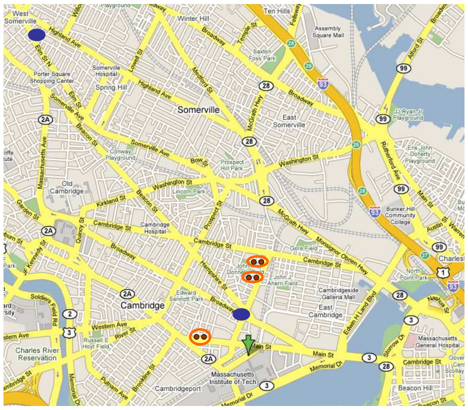
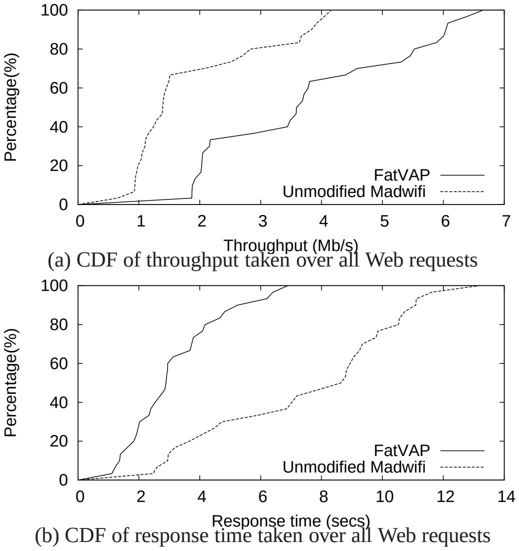


Figure 14: Location of residential deployments (in red) and hotspots (in blue)



(a) CDF of throughput taken over all Web requests

(b) CDF of response time taken over all Web requests

Figure 15: FatVAP's performance in three residential deployments in Cambridge, MA. The figure shows that FatVAP improves the median throughput by 2.6x and reduces the median response time by 2.8x.

the web requests in all three locations. The figure shows that FatVAP increases the median throughput in these residential deployments by 2.6x. Fig. 15(b) plots the CDF of the response time taken over all requests. The figure shows that FatVAP reduces the median response time by 2.8x. Note that though all these locations have only two APs, Web performance more than doubled. This is due to FatVAP's ability to balance the load across APs. Specifically, most Web flows are short lived and have relatively small TCP congestion windows. Without load balancing, the bottleneck drops a large number of packets, causing these flows to time out, which results in worse throughputs and response times. In short, our results show that FatVAP brings immediate benefit in today's deployments, improving both client's throughput and response time.

4.9 Hotspots

Results in Hotspots show that FatVAP can aggregate throughput across commercial access points. The traf-

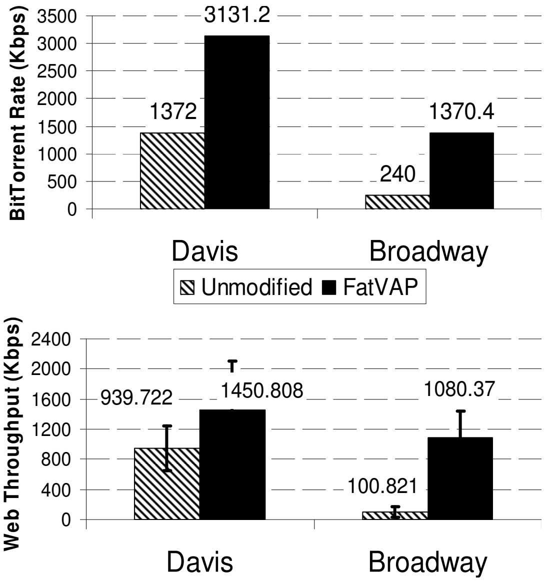


Figure 16: FatVAP's performance in two hotspots in Cambridge/Somerville, MA, showing that FatVAP improves throughput for Web downloads and BitTorrent.

fic load uses both Web downloads and BitTorrent and is generated as described in §4.1(d). Our results show that both Web sessions and BitTorrent obtain much improved throughput compared to the unmodified driver. Fig. 16 shows that, depending on the Hotspot, FatVAP delivers an average throughput gain of 1.5-10x to Web traffic, and $2 - 6$x to BitTorrent. The huge gains obtained in the Broadway site are because the AP with the highest RSSI was misconfigured with a very large queue size. When congested, TCPs at this AP experienced a huge RTT inflation, time-outs, and poor throughput.

5 Related Work

Related work falls in two main areas.

(a) Connecting to Multiple APs: There has been much interest in connecting a wireless user to multiple networks. Most prior work uses separate cards to connect to different APs or cellular base stations [5, 23, 27]. PERM [27] connects multiple WiFi cards to different residential ISPs, probes the latency via each ISP, and assigns flows to cards to minimize latency. Horde [23] uses multiple cellular connections via different providers. In contrast to this work which stripes traffic across independent connections, FatVAPuses the same card to associate and exchange data with multiple APs. Further, FatVAP uses virtual connections to these APs that are very much dependent and so are the throughput estimates that FatVAP uses to choose APs.

The closest to our work is the MultiNet project [13], which was later named VirtualWiFi [6]. MultiNet abstracts a single WLAN card to appear as multiple virtual WLAN cards to the user. The user can then configure each virtual card to connect to a different wireless network. MultiNet applies this idea to extend the reach of APs to far-away clients and to debug poor connectivity. We build on this vision of MultiNet but differ in design and applicability. First, MultiNet provides switching capabilities but says nothing about which APs a client should toggle and how long it should remain connected to an AP to maximize its throughput. In contrast, FatVAP schedules AP switching to maximize throughput and balance load. Second, FatVAP can switch APs at a fine time scale and without dropping packets; this makes it the only system that maintains concurrent TCP connections on multiple APs.

(b) AP Selection: Current drivers select an AP based on signal strength. Prior research has proposed picking an AP based on load [20], potential bandwidth [28], and a combination of metrics [21]. FatVAP fundamentally differs from these techniques in that it does not pick a single AP, but rather multiplexes the various APs in a manner that maximizes client throughput.

6 Discussion

Here, we discuss some related issues and future work.

(a) Multiple WiFi Cards: While FatVAP benefits from having multiple WiFi cards on the client's machine, it does not rely on their existence. We made this design decision for various reasons. First most wireless equipments naturally come with one card and some small handheld devices cannot support multiple cards. Second, without FatVAP the number of cards equals the number of APs that one can connect with, which limits such a solution to a couple of APs. Third, cards that are placed very close to each other may interfere; WiFi channels overlap in their frequency masks [16] and could leak power to each other's bands particularly if the antennas are placed very close. Forth, even with multiple cards, the client still needs to pick which APs to connect to and route traffic over these APs as to balance the load. FatVAP does not constrain the client to having multiple cards. If the client however happens to have multiple cards, FatVAP would allow the user to exploit this capability to expand the number of APs that it switches between and hence improve the overall throughput.

(b) Channel Bonding and Wider Bands: Advances like channel bonding (802.11n) and wider bands (40MHz wide channels) increase wireless link capacity to hundreds of Mb/s. Such schemes widen the gap between the capacity of the wireless link and the AP's backhaul link, making FatVAP more useful. In such settings, FatVAP lets one wireless card collect bandwidth from tens of APs.

(c) WEP and Splash Screens: We are in the process of adding WEP and splash-screen login support to our FatVAP prototype. Supporting WEP keys is relatively easy, the user needs to provide a WEP key for every secure AP

that he wants to access. FatVAP reads a pool of known <WEP key, ESSID> tuples and uses the right key to talk to each protected AP. Supporting splash screen logins used by some commercial hot-spots, is a bit more complex. One would need to pin all traffic to an AP for the duration of authentication, after which FatVAP can distribute traffic as usual.

7 Conclusion

Prior work has documented the abundance of 802.11 access points and the fact that APs occur in bunches—if you see one, it is likely that many others are close by. This paper takes the next step by aggregating bandwidth across the many available APs, that may be spread across different channels. To the best of our knowledge, FatVAP is the first driver to choose how to long to connect to each AP, maintain concurrent TCP flows through multiple APs and provide increased throughput to unmodified applications. FatVAP requires no changes to the 802.11 MAC or to access points. Fundamentally, FatVAP relaxes the constraint that a user with one card can only connect with one access point to achieve both better performance for users and a better distribution of load across APs.

Acknowledgments

We thank our shepherd Srini Seshan, Ranveer Chandra, Szymon Chachulski, Nate Kushman, Hariharan Rahul, Sachin Katti and Hari Balakrishnan for comments that improved this paper. This work is supported by the NSF Career Award CNS-0448287 and an ITRI gift. The opinions and findings in this paper are those of the authors and do not necessarily reflect the views of NSF or ITRI.

References

[1] Azureus. `http://azureus.sourceforge.net`.
[2] iperf. `http://dast.nlanr.net/Projects/Iperf`.
[3] Knapsack. `http://en.wikipedia.org/wiki/Knapsack_problem`.
[4] Madwifi. `http://madwifi.org`.
[5] Slurpr. `http://geektechnique.org/projectlab/781/slurpr-the-mother-of-all-wardrive-boxes`.
[6] Virtual WiFi. `http://research.microsoft.com/netres/projects/virtualwifi`.
[7] Virtual WiFi–FAQ. `http://research.microsoft.com/netres/projects/virtualwifi/faq.htm`.
[8] WebStone. `http://www.mindcraft.com/webstone/`.
[9] WiFi411. `http://www.wifi411.com`.
[10] Wigle.net. `http://wigle.net`.
[11] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and Techniques for Diagnosing Faults in IEEE 802.11 Infrastructure Networks. In MobiCom, 2004.
[12] P. Bahl, R. Chandra, T. Moscibroda, Y. Wu, and Y. Yuan. Load Aware Channel-Width Assignments in Wireless LANs. Technical Report TR-2007-79, Microsoft Research, 2007.
[13] R. Chandra, P. Bahl, and P. Bahl. MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card. In INFOCOM, 2004.
[14] K. Egevang and P. Francis. The IP Network Address Translator (NAT). In RFC 1631, May 1994.
[15] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP Traffic: A Study of the Role of Variability and the Impact of Control. In SIGCOMM, 1999.
[16] M. Gast. 802.11 Wireless Networks: The Definitive Guide. O'Reilly, 2005.
[17] M. Jain and C. Dovrolis. Pathload: A Measurement Tool for End-to-End Available Bandwidth. In PAM, March 2002.
[18] H. Jamjoom and K. G. Shin. Persistent Dropping: An Efficient Control of Traffic. In ACM SIGCOMM, 2003.
[19] H. Jiang and C. Dovrolis. Passive Estimation of TCP Round-Trip Times. In ACM CCR, 2002.
[20] G. Judd and P. Steenkiste. Fixing 802.11 access point selection. ACM CCR, 2002.
[21] A. J. Nicholson, Y. Chawathe, M. Y. Chen, B. D. Noble, and D. Wetherall. Improved Access Point Selection. In MobiSys, 2006.
[22] K. Papagiannaki, N. Taft, and C. Diot. Impact of Flow Dynamics on Traffic Engineering Design Principles. In INFOCOM, 2004.
[23] A. Qureshi and J. Guttag. Horde: Separating Network Striping Policy from Mechanism. In MobiSys, 2005.
[24] M. Rodrig, C. Reis, R. Mahajan, D. Wetherall, and J. Zahorjan. Measurement-based Characterization of 802.11 in a Hotspot Setting. In SIGCOMM E-WIND Workshop, 2005.
[25] S. Kandula and D. Katabi and S. Sinha and A. Berger. Dynamic Load Balancing Without Packet Reordering. In CCR, 2006.
[26] J. Strauss, D. Katabi, and F. Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In IMC, Oct. 2003.
[27] N. Thompson, G. He, and H. Luo. Flow Scheduling for End-host Multihoming. In IEEE INFOCOM, 2006.
[28] S. Vasudevan, D. Papagiannaki, and C. Diot. Facilitating Access Point Selection in IEEE 802.11 Wireless Networks. In IMC, 2005.

A End-to-end Available Rate Inflation

Suppose that the client spends f_i of its duty cycle D at an AP that has wireless and end-to-end available bandwidths, w_i and e_i. Thus, the client spends $f_i \times D$ time at this AP and the remaining $(1-f_i)D$ time at other APs. The AP buffers data that it receives when the client is away and delivers this data when the client next connects to the AP. Let x be the amount served out of the AP's buffer, then

$$x = e_i \left((1-f_i)D + \frac{x}{w_i} \right) \tag{14}$$

Eq. 14 means that the buffer gets data at rate e_i during two phases: when the client is away from the AP and when data in the buffer is being delivered to the client. The first phase lasts for $(1-f_i)D$ and the second lasts for $\frac{x}{w_i}$. The total data received by the client in D seconds is,

$$DataReceived = x + e_i(f_i D - \frac{x}{w_i}). \tag{15}$$

This means simply that the client receives x units from the buffer in time $\frac{x}{w_i}$ and once the buffer is depleted, receives data at the end-to-end rate e_i for the remaining $f_i D - \frac{x}{w_i}$. Since the client listens at the AP for

$$ListenTime = f_i D, \tag{16}$$

the client's estimate of end-to-end available bandwidth is

$$Estimate = \frac{DataReceived}{ListenTime} = \frac{e_i}{f_i}. \tag{17}$$

Eq 17 is obtained by eliminating x from Eqs. 14, 15, 16. But, if the fraction of time that the client listens to the AP is smaller than $\frac{e_i}{w_i}$, it is easy to see that the client will always be served data from the AP buffer at the wireless available bandwidth w_i. Hence, the estimate is

$$Estimate = \max\left(\frac{e_i}{f_i},\ w_i\right). \tag{18}$$