Generating NFL Game Headlines from Box Score Statistics

Parker Greene, Itamar Belson, John Clarke
{pwgreene, ibelson, jaclarke}@mit.edu

I. Introduction

With the large number of users and diverse use of applications within the MIT wireless network, maintaining performance becomes critical. When devices attempt to connect to the MIT network, it is common for multiple Access Points to be within range so devices will choose to connect to the strongest signal AP. However, this may not be optimal for performance, since signal strength indicates nothing about traffic congestion. Depending on the application, it could be better to connect to an AP whose signal strength is weaker, but has less congestion and thus higher throughput.

In looking to address this issue and optimize performance, MIT facilities seeks for its wireless internet users to be able to connect to Access Points (APs) with acceptable performance, either by connecting to an AP within range or by connecting to a nearby recommended AP, all while still maintaining maximum network utilization across the whole MIT network. In addition, for the purposes of network management, MIT facilities needs to collect data on each AP's amount of data transfer as well as its approximate number of unique users. To accomplish this, we propose Latch, a simple, reliable, and scalable system for optimizing network performance for users and optimizing network utilization.

II. System Design

Latch's basic design consists of three modules: Access Points, client users, and an MIT Information System & Technology (IS&T) server, as shown in Figure 1. Access Points are connected to the server, through which all congestion data is transmitted. The server stores this data as well as the locations for APs in order to relay to each AP congestion information about their neighboring APs. Clients connect to APs and send their application requirements as well as a list of nearby APs with acceptable performance. Using this information and the congestion information from the server, the AP uses an optimization algorithm to determine whether the the client should connect and if not, it recommends another nearby acceptable AP to connect to, sending this information to the client as a connection message.

A. Server

The data stored in the IS&T server will primarily be organized in two SQL tables: `Congestions` and `Local_AP`.

The `Congestions` table maps the MAC address of every AP in the network to its measure of congestion. It is defined in Listing 1.

```
CREATE TABLE Congestion (
  address VARCHAR(12),
  bytes_utilized BIGINT,
  capacity BIGINT,
  congestion BIGINT);
```

Listing 1: Congestion Table

The `bytes_utilized` field gives a straightforward measurement of remaining AP capacity, while the congestion column is reserved for a separate measure possibly based on dropped packets or user happiness levels. The `Local_AP` table stores each AP in the network with a list of neighboring APs within 625' of it. It is defined in Listing 2.

```
CREATE TABLE Local_AP (
  address VARCHAR(12),
  local_APs VARCHAR(12) MULTISET);
```

Listing 2: Local_AP Table

When the server receives packets from a certain AP, it updates the `Congestions` table appropriately. Then, the server will read all of the congestion data that needs to be sent to this AP from the table. The process happens sequentiallythe server receives data from AP a, then responds to a with the congestion data it needs.

For example, say an AP with MAC address `01-23-45-67-89-ab` sends its congestion data as two variables b,c=(# of bytes utilized, congestion) to the IS&T server. The IS&T server will run the following query in Listing 3 to update the congestion table.

```
UPDATE Congestions
  SET bytes_utilized=b,congestion=c
  WHERE address='01-23-45-67-89-ab';
```

Listing 3: Congestion Updates

Then it will retrieve a list of the APs nearby as in Listing 4.

```
SELECT local_aps
  FROM Local_AP
  WHERE address='01-23-45-67-89-ab';
```

Listing 4: Local AP Query

After it has this list of MAC addresses, it will run a query into `Congestions` to gather a list of each AP and its corresponding congestion data. The procedure is defined in pseudo-code in Listing 5.

Fig. 1. Communications Diagram

```
For MAC_address in local_aps:
  data <- SELECT congestion
          FROM Congestions
          WHERE address=MAC_address
send data to 01-23-45-67-89-ab
```

Listing 5: Congestion Response to AP

It will forward this result to the AP that initially sent its data to the server.

Because the congestion data is sent to the server every 2 minutes, the response data about local APs that the IS&T server sends back to the AP will never be more than 2 minutes out of date. Additionally, because the `Local_AP` table will be dealing with many reads in a single second, the reads can be distributed among the server's many cores, so that the process of retrieving the data can be done concurrently and responses will be nearly instantaneous. The update to the `Congestion` table, however, limits the ability for reads to happen concurrently. Therefore, we let the update happen only once every 30 seconds on a single, separate thread with a copy of the outdated table. Once the update is complete, it will replace the outdated table with the new updated `Congestion` table. The server needs to store a buffer that contains all of the congestion data it received since the last update of this table so that it can do these updates with batches of the congestion data.

The number of APs in the system at any given time is less than 4000, which means the server will be doing at most 33 reads/second from the `Congestion` and `Local_AP` tables, and every 30 seconds the server will need to update the approximately 500 entries in the `Congestion` table. We want to ensure that the APs are not all sending their data to the server at the same time so as to ensure the maximum number of packets received by the server at any given 30 second interval is unlikely to be too much higher than 500.

B. Initial Connection Processes

The client to AP connection process begins with a series of steps to find the optimal AP to which a client should be linked.

1) Client Phase: When a client attempts to initially connect to an access point, it needs to intelligently determine which APs it should try first. The following sequence creates an ordered list for the client to use to find the optimal AP.

For each channel from 1 to 11, the client will switch to that channel and wait for a heartbeat. As this happens, the client stores a list of tuples for the APs that it finds in the form [(AP MAC Address, AP to Client signal strength)]. This search process takes at most 385ms, allowing 5ms to switch channels and 30ms to wait for a heartbeat. The client then filters the list by filter strength, keeping only the APs whose strength it deems acceptable (this measure of acceptability is determined by the client itself, e.g. all signals above 60%). Next, it sorts the list in descending order by strength. These operations will take a small amount time, as the client is only searching on 11 channels, so there can only be at most 11 APs.

Once the client has this list, it will connect to the best, or strongest AP. If this AP is too congested, it will make space for the new client using the Kickoff Algorithm described in the next section. The AP will tell the clients who got kicked off which APs they should connect to next.

2) AP Phase: When a new client connects to an AP, the AP needs to know if it has room to support the new client. By storing certain relevant information and running a clever congestion analysis algorithm, the AP will know if it has room, and how to make room if it does not.

Each AP stores a table mapping its geographically neighboring APs to their levels of congestion. It also holds a list of its currently connected clients and the APs to which they have reasonable signal strength.

Every two minutes, the AP contacts the IS&T server. The AP tells the server its own congestion level, and the server sends back an updated table of the nearby AP congestion levels. A metric for determining the level of congestion is described in a later section.

The AP sorts its list of connected clients in order of whom it should kick off first, using the Kickoff Algorithm described in the next section. If the AP's congestion level is too high, it kicks clients off the AP, starting at the top of the list, until it is no longer congested. When it kicks a client off, it tells them which of their acceptable APs is the best one for them to connect to. If the congestion level is acceptable and there is room for the new client, the AP simply allows the connection to continue and includes the new client in its next congestion

analysis.

C. Kick-off Algorithm

Because our APs allow any client to connect, we can end up in a situation where too many clients have connected. To prevent congestion, we must prune the client list regularly.

In order to do the pruning, we maintain a list of connected clients, sorted in the order we would like to kick them off. When a new client connects, we just insert them into the list at the correct position. When our AP congestion map changes, we have to re-sort the entire list (but this happens infrequently).

Then we regularly measure our own congestion, and if it is above acceptable levels we kick clients from the top of the list until we have some breathing room.

So the question is how to rank the clients. We choose to rank them in terms of how good their next best alternative AP is. To do this, we need to know what APs each client can connect to, how much data the client needs, and how congested those APs are.

This scheme also has the advantage of calculating the best alternative for every client–so when we kick that client, we tell them which AP to try to reconnect to.

When a client connects, we get a list from them that tells us which APs they are happy to connect to. We assume this list never changes until they disconnect–we never update it. If the client does not move around much, the signal strengths they have to various APs should not change much, and none of the APs should cross from unacceptable to acceptable or vice versa. In the worst case, if the client for example walks around, we might kick them and redirect them to an AP they can no longer reach. Then they would re-initiate their scanning routine, and re-connect to us with a new list of acceptable APs–and this would only take a few seconds.

The client has the information about their signal strengths to various APs, and the AP knows how congested the nearby APs are. So the score for a client C with desired data rate A is the congestion on the least congested AP that C could connect to with the ability to support A more data.

That's quite a mouthful, so Python pseudo-code is shown in Listing 6.

```
def score_client(c):
  desired_data_rate = c['data']
  acceptable_aps = [a for a in c['aps'] if
   ↪ free_space(a) > desired_data_rate]
  return max(map(congestion,
   ↪ acceptable_aps))
```

Listing 6: Client Kick-off-ability

Our APs are memoryless, in the sense that we don't keep track of who has connected to us in the past. That's a nice property to avoid using tons of memory, but it means that we might end up in a bad infinite loop if we're not careful. If we redirect clients to an AP that cannot support them, that AP will kick them out as well–and if it kicks them back to us, we will just kick the client right back to the same AP. The loop would continue until either one of the APs could support the client, or the APs updated their congestion measurements and redirected differently. The former might not happen, and the latter takes up to two minutes (and might not even fix the problem). If this ever occurs, we will have an unhappy client–so measuring our congestion conservatively is very important. If other APs think we have space but we don't, the consequences are very bad. If they think we don't have space when we just barely do, they just won't redirect users to us.

Getting an accurate measure of congestion isn't necessarily easy. In our design, we calculate the amount of free bandwidth we would have if each client used their full requested data rate, and call that the amount of free space we have. We have designed our system to have room for a different measurement of congestion–such as the number of unhappy users, or some function of the times we drop packets. For our final report, we intend to analyse the properties of these various congestion measures and decide on one which makes our algorithm most robust.

Another problem with this scheme is that we don't redirect any clients until an AP is full. We could add some procedure to the AP that would try to prune clients if the neighboring APs were significantly less congested, but we think the unhappiness from balancing users who are getting adequate service won't be worth it.

III. Conclusion

Latch is a system for optimizing user's network performance while maximizing network utilization. It is designed for simplicity, reliability, as well as scalability. The system also meets the requirements for network management of MIT's network in collecting data on the APs' amount of data transfer and number of users. This is accomplished by leveraging the use of our designed algorithms for constant optimization updates and effective use of data models.

Currently, the algorithms used don't take care of the balancing of clients, until congestion is present. Also, the client's unhappiness is not taken into consideration during the network balancing of clients. Our further work will be geared towards improving the optimization algorithms both to take care of client balancing before any congestion as well as to make usage of clients' happiness in balancing optimization.