PySkyWiFi: completely free, unbelievably stupid wi-fi on long-haul flights

09 Jul 2024

The plane reached 10,000ft. I took out my laptop, planning to peruse the internet and maybe do a little work if I got really desperate.

I connected to the in-flight wi-fi and opened my browser. The network login page demanded credit card details. I fumbled for my card, which I eventually discovered had hidden itself inside my passport. As I searched I noticed that the login page was encouraging me to sign in to my airmiles account, free of charge, even though I hadn't paid for anything yet. A hole in the firewall, I thought. It's a long way from London to San Francisco so I decided to peer through it.

I logged in to my JetStreamers Diamond Altitude account and started clicking. I went to my profile page, where I saw an edit button. It looked like a normal button: drop shadow, rounded corners, nothing special. I was supposed to use it to update my name, address, and so on.

But suddenly I realised that this was no ordinary button. This clickable rascal would allow me to access the entire internet through my airmiles account. This would be slow. It would be unbelievably stupid. But it would work.

Several co-workers were asking me to review their PRs because my feedback was "two weeks late" and "blocking a critical deployment." But my

ideas are important too so I put on my headphones and smashed on some focus tunes. I'd forgotten to charge my headphones so Limp Bizkit started playing out of my laptop speakers. Fortunately no one else on the plane seemed to mind so we all rocked out together.

Before I could access the entire internet through my airmiles account I'd need to write a few prototypes. At first I thought that I'd write them using Go, but then I realised that if I used Python then I could call the final tool `PySkyWiFi` . Obviously I did that instead.

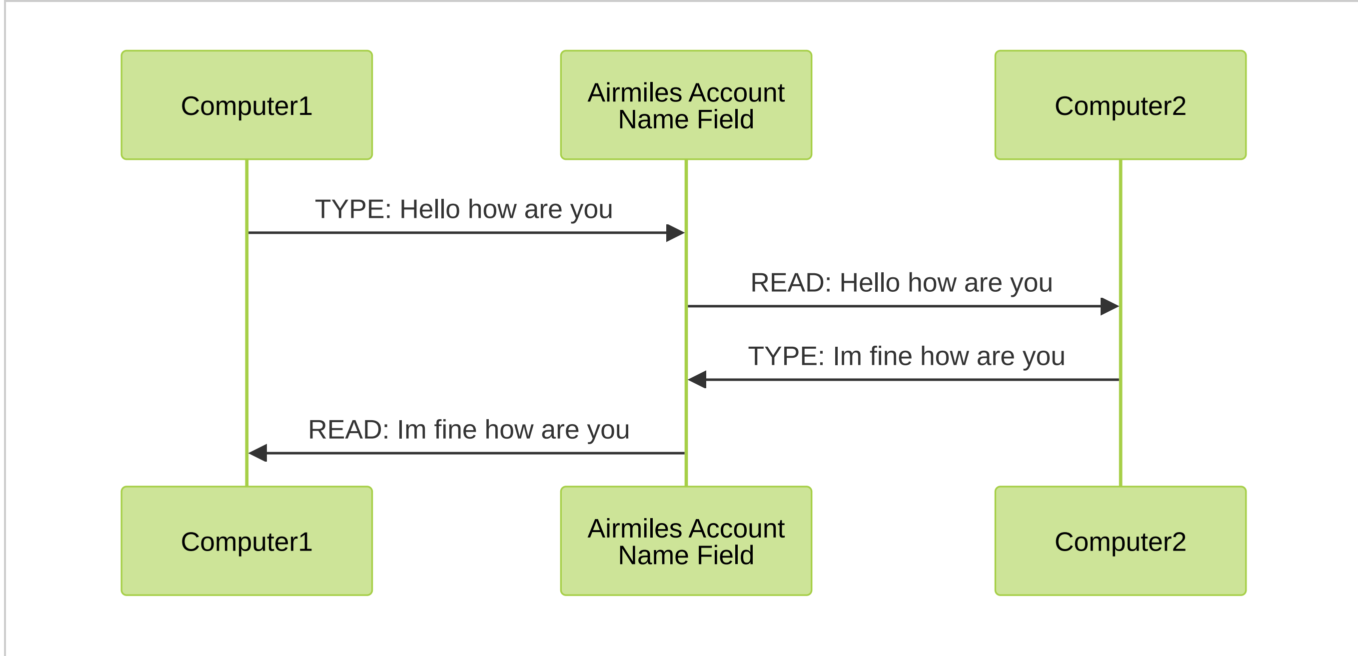
Prototype 1: Instant Messaging

Here's the basic idea: suppose that I logged into my airmiles account and updated my name. If you were also logged in to my account then you could read my new name, from the ground. You could update it again, and I could read your new value. If we kept doing this then the name field of my airmiles account could serve as a tunnel through the airplane's wi-fi firewall to the real world.

This tunnel could support a simple instant messaging protocol. I could update my name to " `Hello how are you` ." You could read my message and then send me a reply by updating my name again to " `Im fine how are you` ." I could read that, and we could have a stilted conversation. This might not sound like much, but it would be the first step on the road to full internet access.

I paid for the internet on my old laptop. I hadn't finished migrating my data off this computer, so it still had to come everywhere with me. I messaged my wife to ask her to help me with my experiments. `no, what are you talking about, i'm busy` she replied, lovingly.

So instead I took out my new laptop, which still had no internet access. I created a test airmiles account and logged into it on both computers. I found that I could indeed chat with myself by updating the name field in the UI.



This was a lousy user experience though. So I wrote a command line tool to automate it. My tool asked the user for a message, and then behind the scenes it logged into my airmiles account via the website, using my credentials. The tool updated the name field of my test account with the user's message. It then polled the name field every few seconds to see if my account's name had changed again, which would indicate that the other person had sent a message back. Once the tool detected a new value it printed that value and asked the user for their next reply, and so on.

Me
Airmiles Account Name Field
You
(poll for new data)
(no new data)
WRITE: Hello how are you
(poll for new data)
READ: Hello how are you
(poll for new data)
(no new data)
WRITE: Im fine how are you
(poll for new data)
READ: Im fine how are you
Me
Airmiles Account Name Field
You

Using this tool I could chat with someone on the ground, via my terminal. I wouldn't have to pay for wifi, and neither of us would have to know or care that the messages were being sent via my SkyVenture Premium Gold Rewards account.

I still needed to find someone who would chat with me. But this was a good start!

> *NB: at this point I didn't want to send any more automated data through my airmiles account in case that got me in trouble somehow. Nothing I was doing could possibly cause any damage, but some companies get jumpy about this kind of thing.*
>
> *I therefore proved to myself that PySkyWiFi would work on my airmiles accounts too by updating my name ten or so times in quick succession. They all succeeded, which suggested to me that my*

> *airmiles account probably wasn't rate-limiting the speed or number of requests I could send to it.*
>
> *I then wrote the rest of my code by sending my data through friendly services like GitHub Gists and local files on my computer, using the same principles as if I were sending it through an airmiles account. If PySkyWiFi worked through GitHub then it would work through my Star Power UltimateBlastOff account too. This had the secondary advantage of being much faster and easier for iteration too.*
>
> *I'm going to keep talking about sending data through an airmiles account, because that's the point I'm trying to make.*

Prototype 2: Live headlines, stock prices, and football scores

The tunnel I'd constructed through my airmiles account would be useful for more than IMing. For my next prototype I wrote a program that would run on a computer back at my house or in the cloud, and would automatically send information from the real world up to me on the plane, through my airmiles account. I could deploy it before I left for my next flight and have it send me the latest stock prices or football scores while I was in the sky.

To do this I wrote a daemon that would run on a computer that was on the ground and connected to the internet. The daemon constantly polled the name field in my airmiles account, looking for structured messages that I sent to it from the plane (such as `STOCKPRICE: APPL` or `SCORE: MANUNITED`). When the daemon saw a new request it parsed it, retrieved the requested information using the relevant API, and sent it back to me via my airmiles account. It worked perfectly.

Now I could use my first prototype to send IMs through my airmiles account, and I could use my second prototype tio follow the markets and the sports.

It was time to squeeze the entire internet through my airmiles account.

The real thing: PySkyWiFi

During the rest of the flight I wrote PySkyWiFi. PySkyWiFi is a highly simplified version of the TCP/IP protocol that squeezes whole HTTP requests through an airmiles account, out of the plane, and down to a computer connected to the internet on the ground. A daemon running on this ground computer makes the HTTP requests for me, and then finally squeezes the completed HTTP responses back through my airmiles account, up to me on my plane.

This meant that on my next flight I could technically have full access to the internet, via my airmiles account. Depending on network conditions on the plane I might be able to hit speeds of several bytes per second.

> *DISCLAIMER: you obviously shouldn't actually do any of this*

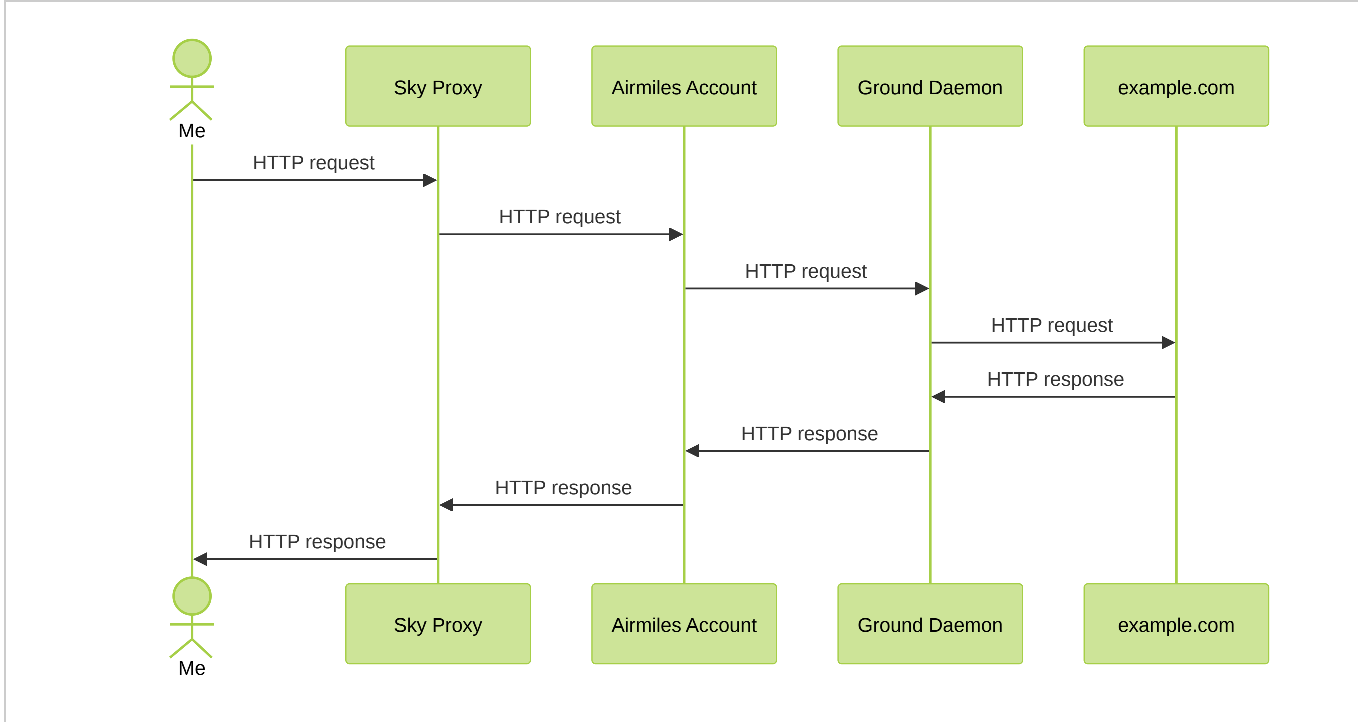
Here's how it works (and [here's the source code]()).

How PySkyWiFi works

PySkyWiFi has two components:

- **The sky proxy** - a proxy that runs on your laptop, on a plane
- **The ground daemon** - a daemon that runs on a computer connected to the internet, at your home on the ground or in the cloud

Here's a simplified diagram:



Setup starts before you leave your house. First you start up the ground daemon. Then you get a taxi to the airport, get on the plane, and connect to the plane's wi-fi network. You boot up the sky proxy on your laptop. Your PySkyWiFi relay is now ready to go.

You use a tool like `curl` to make an HTTP request to the sky proxy that you've started on your laptop. You address your request to the proxy (eg. `localhost:1234/`) and you put the actual URL that you want to query inside a custom HTTP header called `X-PySkyWiFi` . For example:

```
curl localhost:1234 -H "X-PySkyWiFi: example.com"`
```

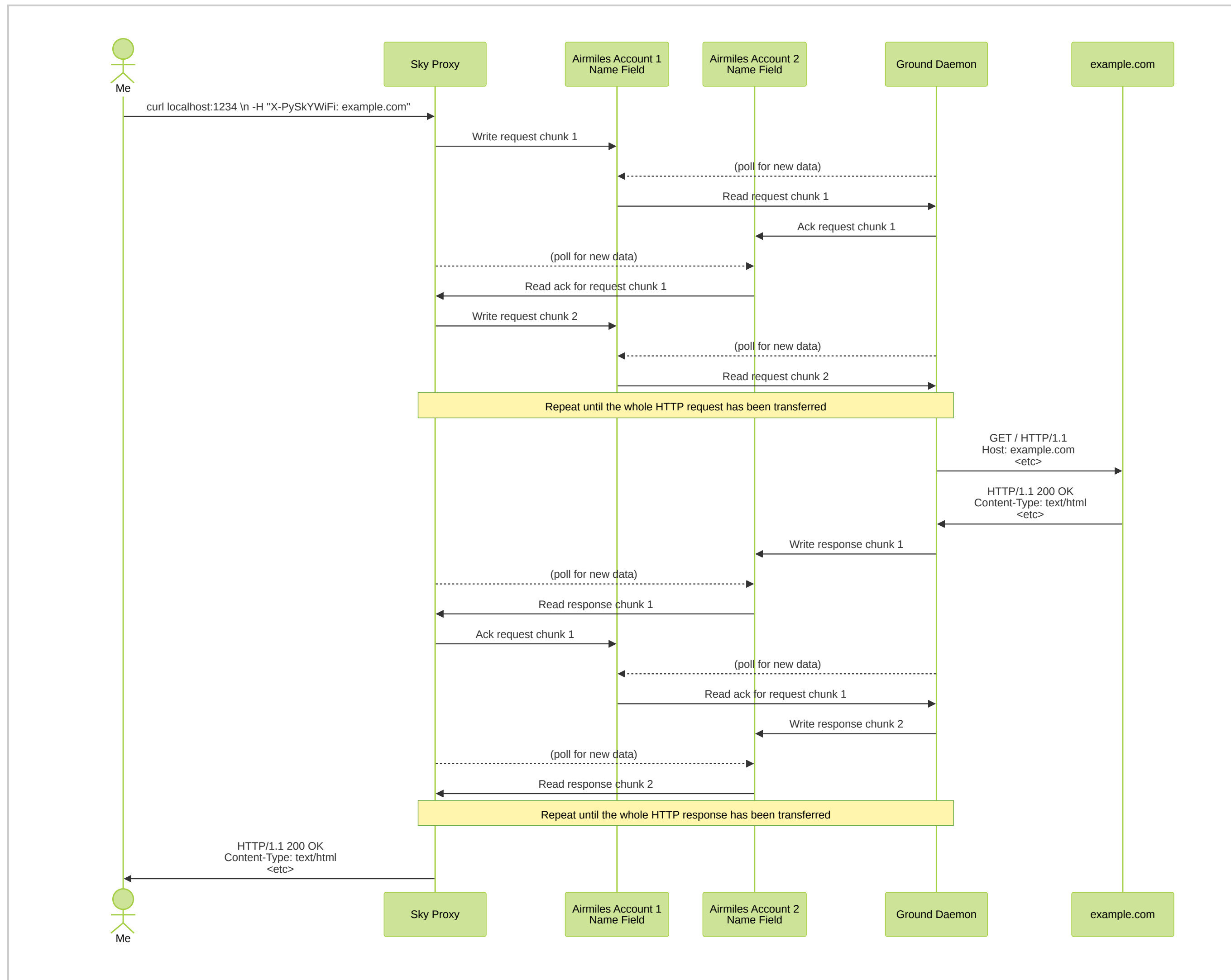
The `X-PySkyWiFi` header will be stripped by the ground daemon and used to route your request to your target website. Everything else about the request (including the body and other headers) will be forwarded exactly as-is.

Once you make your request it will hang for several minutes. If by some miracle nothing breaks then you'll eventually get back an HTTP response, exactly as if you'd sent the request over the normal internet like a normal person. The only difference is that it didn't cost you anything. You will now

almost certainly pay for wi-fi, because your curiosity has been satisfied and your time on this earth is very short.

Step-by-step

Here’s what happens behind the scenes:



In order:

1. The sky proxy receives the HTTP request from your `curl` call. It splits the request into chunks, because the entire request is too large to fit into you airmiles account in one go
2. The sky proxy writes each chunk one-by-one to the name field in your airmiles account.
3. The ground daemon polls your airmiles account. When it detects that the name field has changed to a new chunk, it reads that chunk and sends an acknowledgement to the sender so that the sender knows it’s

safe to send the next chunk. The receiver sticks the chunks back together and rebuilds the original HTTP request
4. Once the ground daemon has received and rebuilt the full HTTP request, it sends the request out over the internet.
5. The ground daemon receives an HTTP response.
6. The ground daemon sends the HTTP response up to the sky proxy using the same process as before, in reverse. This time the ground daemon splits the HTTP response up into chunks and writes each chunk one-by-one to the name field in your airmiles account (it actually writes these response chunks using a second airmiles account to make the protocol simpler)
7. The sky proxy polls the second airmiles account. It reads each chunk and sticks them back together to rebuild the HTTP response
8. The sky proxy returns the HTTP response to the original call to `curl`. As far as `curl` is concerned this is a perfectly normal HTTP response, just a little slow. `curl` has no idea about the silliness that just transpired

The sky proxy and the ground daemon are relatively simple: they send HTTP requests and parse HTTP responses. The magic is in how they squeeze these requests and responses through an airmiles account. Let’s look closer.

Squeezing HTTP requests through an airmiles account

PySkyWiFi’s communication logic is split into two layers: a **transport layer**, and a **network layer**. The transport layer’s job is to decide what data clients should send to each other. It dictates how senders should split up long messages into manageable chunks, as well as how senders and receivers should signal information like “I am ready to receive another chunk.” The PySkyWiFi transport layer is somewhat similar to the TCP protocol that powers much of the internet, if you squint very hard and don’t know much about TCP.

By contrast, the network layer’s job is to actually send data between clients, once the transport protocol has decided what that data should be. It’s

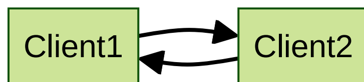
vaguely similar to the IP protocol, if you squint even harder and know even less what you're talking about.

This division of responsibility between layers is useful because the transport layer doesn't have to care about how the network layer sends its data, and the network layer doesn't care what the data it sends means or where it came from. The transport layer just hands the network layer some data, and the network layer sends it however it likes.

This separation makes it easy to add support for new airmiles platforms, because all we have to do is implement a new network layer that reads and writes to the new type of airmiles account. This separation also allows us to write test versions of the network protocol that write and read from local files instead of airmiles accounts. In each case the network layer changes, but the transport layer stays exactly the same. Here's how they work.

The transport layer

A PySkyWiFi transport connection between two clients consists of two "pipes" (or "airmiles accounts"). Each client has a "SEND" pipe that it can write data to, and a "RECV" pipe that it can read from. Clients write to their SEND pipe by writing data to it, and they read from their RECV pipe by constantly polling it and seeing if anything has changed.



From the transport layer's point of view, a pipe is just something that it can write and read data from. Beyond that the transport layer doesn't care how its pipes work.

At any given moment a PSWF (PySkYWiFi) client can only either send or receive data, but not both. A client in *send mode* will not see data sent by the other client, and a client in *receive mode* should never send data because the other client won't see it. This is unlike TCP, where clients can send or receive data at ay time.

When squeezing HTTP requests and responses through an airmiles account, the sky proxy sends the first message and the ground daemon receives it. Once the sky proxy has finished sending its HTTP request it switches to receive mode and the ground daemon switches to send. The ground daemon makes the HTTP request and sends back the response, at which point the two switch roles again so that the sky proxy can send another HTTP request.

How are long messages sent through such a small pipe?

PSWF uses small pipes (such as an airmiles name field) that can't fit much data in them at once. This means that it takes some work and care to squeeze long messages (like HTTP requests) through them.

To send a long message, the sender first splits up their message into chunks that will fit into their SEND pipe. They then send each chunk down the pipe one at a time.

To begin a message, a sender starts by sending its first chunk of message data inside a `DATA` segment:

> *A `DATA` segment consists of:*
>
> - *The letter `D`*
> - *The sequence number of the chunk (a number that uniquely identifies the chunk, padded to 6 digits)*
> - *The actual chunk of data.*
>
> *For example, a data segment in the middle of a message might read:* `D000451adline": "Mudslide in Wigan causes m`

Once the sender has sent a `DATA` segment, it pauses. It wants to send its next `DATA` segment, but it can't overwrite the airmiles account name field until it knows that the receiver has received and processed the previous one.

The receiver tells the sender that it's safe for to send a new `DATA` segment by acknowledging every segment that it reads. The receiver does this by writing an `ACK` segment to its own SEND pipe:

> *An `ACK` segment consists of:*
>
> - *The letter `A`*
> - *The sequence number of the segment that is being acknowledged (padded to 6 digits)*
>
> *For example: `A000451`*

The sender is constantly polling its own RCV pipe to check for changes, and so it reads this new `ACK` segment promptly. Once the sender reads the `ACK`, it knows that the receiver has received the segment corresponding to the `ACK`'s sequence number. For example, if a sender receives an `ACK` segment with sequence number `000451`, the sender knows that it's safe to send the next `DATA` segment with sequence number `000452`. The sender therefore pulls the next chunk from its message and constructs a new `DATA` segment using this chunk and sequence number. The sender writes the new segment to its SEND pipe, and then pauses waits for another `ACK`.

This loop continues until the sender has sent all the data in its message. To tell the recipient that it's finished, the sender sends an `END` segment.

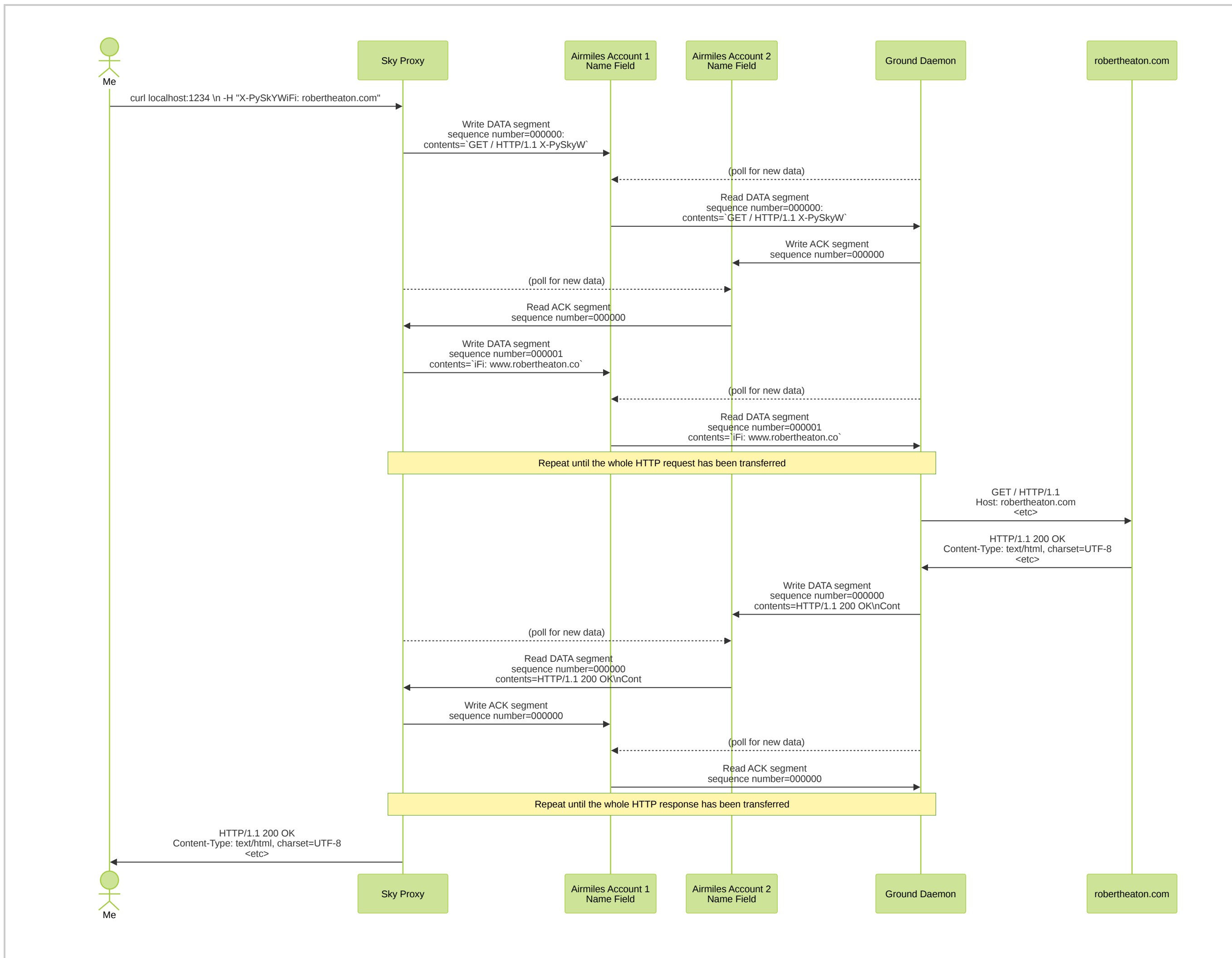
> *An `END` segment is just the letter `E`.*

When a receiver sees an `END` segment it knows that the sender's message is over. The sender and the receiver swap roles. The old sender starts polling its RECV pipe for `DATA` segments, and the old receiver starts chunking up its response message and sending it through its pipe, exactly as before.

None of this transport logic cares about the details of the network layer through which the segments are sent. The transport layer just needs the

network layer to provide two pipes that it can read and write to. The network layer can pipe this data around via local files, a Discord profile, or an airmiles account. This genericness is what allows PySkyWiFi to work with any airline's airmiles account, so long as the airline allows you to login to it from the plane without paying.

Here's how PSWF uses transport protocol segments to exchange long messages:



The transport layer decides what data the clients should send each other, but it doesn't say anything about how they should send it. That's where the network protocol comes in.

The network layer

The network layer's job is to send data between clients. It doesn't care about where the data came from or what it means; it just receives some data from

the transport layer and sends it to the other client (typically via an airmiles account).

This means that the network layer is quite simple. It also means that adding a new network layer for a new airmiles platform is straightforward. You use the new platform to implement a few operations and a few properties (see below), and then the transport layer can automatically to use your new airmiles platform with no extra work.

A network layer consists of two operations:

- `send(msg: str)` - write `msg` to storage. For an airmiles-based implementation, this writes the value of `msg` to the name field in the user's airmiles account
- `recv() -> str` - read the message from storage. For an airmiles-based implementation, this reads the value of the name field from the user's airmiles account.

A network layer implementation must also define two properties:

- `sleep_for` - the number of seconds that the transport layer should sleep for in between polling for new segments from a RECV pipe. `sleep_for` can be very low for test implementations like files, but it should be at least several seconds for an implementation like an airmiles account. This is in order to avoid hammering remote server with too many requests.
- `segment_data_size` - the number of characters that the transport layer should send in a single segment. Should be equal to the maximum size of the airmiles account field being used to transfer segments (often around 20 characters).

A network layer implementation can also optionally provide two more operations:

- `connect_send()` - a hook called by the sender when a SEND pipe is initialised. In an airmiles-based implementation this allows the client to login to the platform using a username and password. This gives the

client a cookie that it can use to authenticate future `send` and `recv` calls.

- `connect_recv()` - a hook called by the receiver when a RECV pipe is initialised

If you fill in all these methods, you'll be able to use PySkyWiFi on a new airline. But again, don't.

Tips and tricks

When writing a network layer that uses a new airmiles provider, there are a couple of tricks that can make your implementation faster and more reliable.

1. Encode messages to make sure the airmiles account accepts them

Airmiles HTML forms usually don't let users include non-alphabetic characters in their name. `Stephen` will probably be allowed, but `GET /data?id=5` will probably be rejected.

To work around this, the network layer should encode segments using base26 before writing them to an airmiles account. base26 is a way of representing a string using only the letters `A` to `Z` . In order to convert a byte string to base26, you convert the bytes to a single large number, then you represent that number using a counting system with base 26 (hence the name) where the digits are the letters `A` to `Z` .

```
def b26_encode(input_string: str) -> int:
    # Convert input string to a base-256 integer
    base256_int = 0
    for char in input_string:
        base256_int = base256_int * 256 + ord(char)

    # Convert base-256 integer to base26 string
    if base256_int == 0:
        return 'A'  # Special case for empty input or input that equals zero

    base26_str = ""
```

```
    while base256_int > 0:
        base26_str = chr(base256_int % 26 + 65) + base26_str
        base256_int //= 26

    return base26_str

b26_encode("Hello world")
# => 'CZEZINADXFFTZEIDPKM'
```

The transport layer never needs to know about this encoding. The network layer receives some bytes, encodes them using base26, and writes this encoded string of `A` to `Z` to the airmiles account. When the network layer reads the base26 value back out of the airmiles account, it decodes the encoded string back into a number and then back into bytes, and then returns those bytes to the transport layer.

Encoding a string using base 26 makes it significantly longer, just like how it takes many more digits to represent a number using binary than decimal. This reduces the bandwidth of our protocol. We could increase our bandwidth by using base52 (using both upper- and lower-case letters) instead of base26, which would shorten it somewhat. This is left as an enhancement for version 2.

2. Increase bandwidth by using more account fields

Another way to increase our PSWF bandwidth is to increase the segment size that a network layer can handle. If we double the size of our segments, we double the bandwidth of our protocol.

Fields in airmiles accounts usually have length limits. For example, you might not be allowed to set a name longer than 20 characters. However, we can maximise our bandwidth by:

1. Using the full length of the field
2. Spreading out a segment across multiple fields

Suppose we have control over 5 fields that can each store 20 characters. Instead of using one field to transmit segments of 20 characters, we can split a 100 character segment into 5 chunks of 20 and update them all at once in a single request. The receiver can then read all 5 fields, again in a single request, and stitch them back together to reconstruct the full segment.

Further enhancements

HTTP `CONNECT`

It would be better if PySkyWiFi used HTTP `CONNECT` requests to set up the tunnel from the sky proxy to the target site, instead of manually tossing around HTTP requests. `CONNECT` requests are how most HTTP proxies work, and using them would allow PySkyWiFi to act as the system-level proxy and so handle requests from a web browser. It would also mean that PySkyWiFi would negotiate TLS connections with the target website directly, so its traffic would be encrypted as it passed through the airmiles account.

On the other hand, using `CONNECT` would also be a lot more work and I've already taken this joke way too far.

In conclusion

When I was done with all of this I used PySkyWiFi to load the homepage of my blog using `curl`, tunneling the data via a GitHub Gist. Several minutes later I got a response back. I scrolled around the HTML and reflected that this had been both the most and least productive flight of my life.

(*PySkyWiFi source code here*)

Get new essays sent to you

Subscribe to my new work on programming, security, and a few other topics. Published a few times a month.

Your email address

Subscribe

Follow me on Twitter ➜ RSS ➜

MORE ON PROGRAMMING

- Hello Deep Learning
- Gamebert: a Game Boy emulator built by Robert
- Gameboy Doctor: debug and fix your gameboy emulator
- What is a self-hosting compiler?
- Migrating bajillions of database records at Stripe
- Code review without your glasses
- Jaccard Similarity and MinHash for winners
- Is Python pass-by-reference or pass-by-value?