CSCI-1680

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, Scott Shenker and John Jannotti
Breathe

i am a tiny cactus
and i believe
in you

you can do the thing
TCP officially due tonight (Tuesday, Nov 21)
  – Office hours 2-5pm (Me); 5-7pm (Alex); 7-9pm (Rhea)

  – Like with IP: you can continue to make small bugfixes after the deadline
    • OK: Fixing small bugs, README, capture files, code cleanup
    • Not OK: eg. implementing sendfile/recvfile, teardown, submitting untested code

  – Grading meetings: after break

If you want to submit late
Monday 11/27 by 11:59pm EST => one day late
The final project

Out after break, handout online after class
...maybe skim it before break?

What it is

- Open-ended: build something new related to class topics
- List of ideas in document... or propose your own!
Project examples

• Make your own iterative DNS resolver
• Build a simple HTTP server
• Make your own web API (more next week)
• Implement Snowcast, etc. using RPCs (more next week)
• Extend your IP/TCP in some way…

These are only a few ideas!
Final project  Logistics

Out after break, document online after class
...maybe skim it before break?

**Deadlines**
- Team assignment form: Due Tuesday, 11/28
  - Keep your current groups, or form new ones, or work solo
- Project proposal: Due Friday, 12/1
- Final submission: Due Thursday, 12/14
IPoAC
How can we improve the physical layer?

Traditional links have fixed bandwidth
- Media limits what frequencies can be used for signal
- Places upper bound on channel capacity
What if we weren’t constrained by the EM spectrum?

How else can we transmit data?
A Standard for the Transmission of IP Datagrams on Avian Carriers

Status of this Memo

This memo describes an experimental method for the encapsulation of IP datagrams in avian carriers. This specification is primarily useful in Metropolitan Area Networks. This is an experimental, not recommended standard. Distribution of this memo is unlimited.

Overview and Rational

Avian carriers can provide high delay, low throughput, and low altitude service. The connection topology is limited to a single point-to-point path for each carrier, used with standard carriers, but many carriers can be used without significant interference with each other, outside of early spring. This is because of the 3D ether space available to the carriers, in contrast to the 1D ether used by
RFC1149: IPoAC

IP over Avian Carriers (1 April 1990)

• High delay, low throughput, low altitude datagram service
• Nearly unlimited movement in 3D etherspace
• Intrinsic collision avoidance
• Typical MTU: 256 milligrams
IPoAC: Design

Packet → Print it out → P10t

 orc

 SCN 1r

↑

SCAN IT IN

↑

PI10t → P16t0n
IPoAC: Implementation

Proof of concept: 28 April 2001
Bergen, Norway
IPoAC in practice

$ ping -c 9 -i 900 10.0.3.1
PING 10.0.3.1 (10.0.3.1): 56 data bytes
64 bytes from 10.0.3.1: icmp_seq=0 ttl=255 time=6165731.1 ms
64 bytes from 10.0.3.1: icmp_seq=4 ttl=255 time=3211900.8 ms
64 bytes from 10.0.3.1: icmp_seq=2 ttl=255 time=5124922.8 ms
64 bytes from 10.0.3.1: icmp_seq=1 ttl=255 time=6388671.9 ms

--- 10.0.3.1 ping statistics ---
9 packets transmitted, 4 packets received, 55% packet loss round-trip
min/avg/max = 3211900.8/5222806.6/6388671.9 ms
Pigeon-powered Internet takes flight

One of the Internet's newest forms of life: transmitting news and data by carrier pigeon.

**Today:** microSD card: ~250mg, 1TB
But actually

What happens if you have a LOT of data to move into the cloud?
Example: AWS
# Feature comparison matrix

<table>
<thead>
<tr>
<th></th>
<th>AWS SNOWCONE</th>
<th>AWS SNOWBALL EDGE STORAGE OPTIMIZED</th>
<th>AWS SNOWBALL EDGE COMPUTE OPTIMIZED</th>
<th>AWS SNOWMOBILE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usable HDD Storage</strong></td>
<td>8 TB</td>
<td>80 TB</td>
<td>N/A</td>
<td>100 PB</td>
</tr>
<tr>
<td><strong>Usable SSD Storage</strong></td>
<td>14 TB</td>
<td>1 TB</td>
<td>28 TB</td>
<td>No</td>
</tr>
<tr>
<td><strong>Usable vCPUs</strong></td>
<td>4 vCPUs</td>
<td>40 vCPUs</td>
<td>104 vCPUs</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Usable Memory</strong></td>
<td>4 GB</td>
<td>80 GB</td>
<td>416 GB</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Device Size</strong></td>
<td>9in x 6in x 3in</td>
<td>548 mm x 320 mm x 501 mm</td>
<td>548 mm x 320 mm x 501 mm</td>
<td>45 ft. shipping container</td>
</tr>
<tr>
<td><strong>Device Weight</strong></td>
<td>4.5 lbs. (2.1 kg)</td>
<td>49.7 lbs. (22.3 kg)</td>
<td>49.7 lbs. (22.3 kg)</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Storage Clustering</strong></td>
<td>No</td>
<td>Yes, 5-10 nodes</td>
<td>Yes, 5-10 nodes</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>256-bit Encryption</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HIPAA Compliant</strong></td>
<td>No</td>
<td>Yes, eligible</td>
<td>Yes, eligible</td>
<td>Yes, eligible</td>
</tr>
</tbody>
</table>
The Internet Header Format [RFC-791]
### IP over Burrito Carriers

<table>
<thead>
<tr>
<th>Obvious</th>
<th>Onion</th>
<th>Jalapenos</th>
<th>Physical Length (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Written on Foil</td>
<td>Bean Type</td>
<td>Number of Beans</td>
<td></td>
</tr>
<tr>
<td>Given Delivery Time</td>
<td>Guacamole</td>
<td>Receipt</td>
<td></td>
</tr>
<tr>
<td>Lettuce</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beef</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Burrito Internet Header Format
April Fool’s Day RFCs

April Fools' Day Request for Comments

From Wikipedia, the free encyclopedia
(Redirected from Peg DHCP)

A Request for Comments (RFC), in the context of Internet governance, is a type of publication from the Internet Engineering Task Force (IETF) and the Internet behaviors, research, or innovations applicable to the working of the Internet and Internet-connected systems.

Almost every April Fool's Day (1 April) since 1989, the Internet RFC Editor has published one or more humorous Request for Comments (RFC) documents, for RFC 527 called ARPAWOCKY, a parody of Lewis Carroll's nonsense poem "Jabberwocky". The following list also includes humorous RFCs published on other days.

Contents [hide]

1 List of April Fools' RFCs
2 Other humorous RFCs
3 Non-RFC IETF humor
4 Submission of April Fools' Day RFCs
5 References
6 Further reading
7 External links

List of April Fools' RFCs [edit]

1978


A parody of the TCP/IP documentation style. For a long time it was specially marked in the RFC index with "note date of issue".

Enjoy!
Back to HTTP
Welcome!

```html
<html>
<title>hi</title>
<h1>Welcome!</h1>
</html>
```
Welcome!
HTTP server

Client

Request: GET /thing

URL/ENDPOINT
Client

Request: GET /thing

Response: 200 OK + thing

HTTP server

HTML, IMG
HTTP request: a way to fetch (GET) or send (POST) some object
- Doesn’t need to be a web page
- Doesn’t need to be from a browser

⇒ Generic way to ask the server to do something => an API over the network!
Modern websites don’t just load pages when you click links:

Every modern webpage is filled with arbitrary code, usually Javascript, which can make more requests:

```javascript
async function doRequest() {
    const response = await fetch("http://example.com/thing.json");
    const data = await response.json();
    console.log(data);
}
```

Can make requests when….
- User does something (click button, scroll, ...)
- Periodic events, timers, etc
- ...

MAKE HTTP REQUEST
Modern websites don’t just load pages when you click links:

Every modern webpage is filled with arbitrary code, usually Javascript, which can make more requests:

```javascript
async function doRequest() {
    const response = await fetch("http://example.com/thing.json");
    const data = await response.json();
    console.log(data);
}
```

Can make requests when….
• User does certain action
• Periodic events, timers, etc
• …

When does this not work?
HTTP server

Browser

Request: GET /thing

Response: 200 OK + thing

Request, response model doesn’t always fit...

=> Server may need to send data asynchronously!
But it’s TCP right?

TCP is bidirectional, but the HTTP protocol is not.
What can be done?

Can the server connect to the client?

Almost always no.

⇒ NAT, Firewalls, security policies are in the way
⇒ Don’t want to allow browser to open a listen port => security risk!
How to wait for the server’s response?

One way: Polling

```go
for {
    resp, err := doRequest("http://example.com/do-you-have-my-data")
    if resp != nil {
        doThing(resp)
    }
    time.Sleep(1 * time.Second)
}
```

- TIME TO WAKE UP
- LOTS OF EXTRA REQUESTS ON SERVER/NETWORK
How to wait for the server’s response?

Another way: long polling
⇒ Require server to hold connection open with long timeout, respond when data is ready

```go
for {
    resp, err := doRequest("http://example.com/do-you-have-my-data")
    // ^ Assume this will block for very long time
    doThing(resp)
}
```

⇒ Client always has pending request
⇒ More cached on server.
Another way: websockets (RFC6455, 2011)

Persistent, bidirectional transport layer between browser and server
=> Can start with an HTTP request!

GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+1YFgZWCXczP8Q==
Sec-WebSocket-Version: 13

Gives you same bidirectional semantics like a TCP socket
How it works:
- Client starts a TCP connection to webserver and sends a normal HTTP request
- Header in request asks to “upgrade” to websocket connection
- Server switches protocol: client and server can now use the same TCP connection to communicate using websocket protocol
  => Can send arbitrary bytes => not limited to HTTP format
  => Bidirectional, like a standard TCP socket => not limited to request/response semantics!!
  => This means the client can WAIT for packet s from the server (think like snowcast!), which is what we want!

Q: why bother starting with an HTTP request?
- Servers already support HTTP: the idea is to extend the protocol with new functionality without altering how servers normally do things
- Browsers don’t let website code make arbitrary connections on the client:
  => Generally, can only make HTTP requests (and with many restrictions), so a websocket connection mostly fits within the existing model for how browsers operate (few new security risks)
- Most firewalls/NATs/etc are designed to permit web traffic: don’t need to rethink how these work, or invent new security policies to support websockets!
Speaking of chat...

Old IM applications (AIM, ICQ, MSN); one TCP connection to server, can only receive message when online (not so different from Snowcast)

Old chat/IM applications: one TCP connection => Can we still do that?
Why doesn’t this work anymore?

Smartphones!
- Maintaining a persistent connection uses battery! Want to avoid this as much as possible
  => Android/iOS halts most apps when phone goes to sleep, unless special permissions granted => Can’t maintain a persistent connection!

Therefore…
=> MUST be able to receive messages when offline => server needs to store messages for later retrieval
  (not a new concept, but was new to chat apps way back then)
=> Need to rethink how we send messages asynchronously!
Push notifications: service provided by OS to handle pushing events for an application
- OS (iOS, Android) maintains a push notification service

**Initial setup**
1. App developer registers application with push service
2. Each user's device is registered with service, also registered with app after user installs it

Device maintains one persistent connection with push service (could be its own TCP connection, one of the web methods we've been talking about, etc.)

When an app wants to send a notification:
3. Application tells the push notification service (has an API for what messages can look like, how long they can be, etc.)
4. When device wakes up, it checks in with the push service and loads any notifications

**Why is this important?**
- Each app doesn't need to maintain its own push service (hard, expensive)
- Using only one push service means the device needs to do less work when it wakes up (and the OS can control when it wakes up)
  => Preserves battery for mobile devices!!
> telnet www.cs.brown.edu 80
Trying 128.148.32.110...
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Thu, 24 Mar 2011 12:58:46 GMT
Server: Apache/2.2.9 (Debian) mod_ssl/2.2.9 OpenSSL/0.9.8g
ETag: "840a88b-236c-49f3992853bc0"
Accept-Ranges: bytes
Content-Length: 9068
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
Example: Github public API

$ curl https://api.github.com/users/ndemarinis
{
    "login": "ndemarinis",
    "id": 1191319,
    "node_id": "MDQ6VXNlcjExOTEzMTk=[",
    "gravatar_id": "",
    "url": "https://api.github.com/users/ndemarinis",
    "type": "User",
    "site_admin": false,
    "name": "Nick DeMarinis",
    "blog": "https://vty.sh",
    "twitter_username": null,
    "public_repos": 10,
    ...
}
Web browser

DNS

example.com?

Webserver example.com

page.html
<html>
<title>hi</title>
<h1>Welcome!</h1>
</html>

Server returns response (in this case, with HTML)