Deployment Considerations in RPKI

Measurements and Data

Contents

subject	slides
problem setting	3 - 7
measurements in the wild	8 - 18
lab tests	19 - 26
conclusions	27 - 28
next steps	29

Current situation

- rsync useful first implementation
  - no re-inventing of syncing protocol
  - works, mostly
  - good enough to build up experience

- But RPs and servers do see issues
  - evaluate further steps based on current experience
  - planning for new/add'l standards takes time

Problems with current repository

- Reliability
  - No support multiple publication points (yet)
  - Criticism from operators in RIPE community:

    “We do not believe this is robust enough to be used for secure internet routing”

Problems with current repository

- Scalability
  - many repositories remain flat (hard on RPs)
  - rsyncd resource heavy (see perf measurements)
  - poor or no support for proxying, caching, load balancing rsyncd instances
    - clients connect to real back-end 1-1
    - vulnerable to rsyncd outages
  - CDN solutions for rsync
    - Do they exist? Not as far as we know.
    - Creating our own is not trivial

Problems with current repository

- Consistency (as seen by 1 RP at a point in time)
  - rsyncd black box:
    - can’t make it respect transactions
    - contents can change during transfer
    - This is indistinguishable (automated) from errors due to server side issues (bugs) and monkey-in-the-middle messing.
- Manifest can help
  - But the manifest rfc (6486) is very loose with regards to missing / extra / wrong-hash objects

Problems with current repository

- Software (RP)
  - Only one rsync implementation available, no RFC describing the protocol
    - no native libraries for programming languages
    - calling external tool is resource hungry
    - parallel calls to ext. tool even more so
    - differences in rsync versions
    - error codes and messages difficult to parse
    - list of what-has-changed difficult to parse

Measurements in the wild

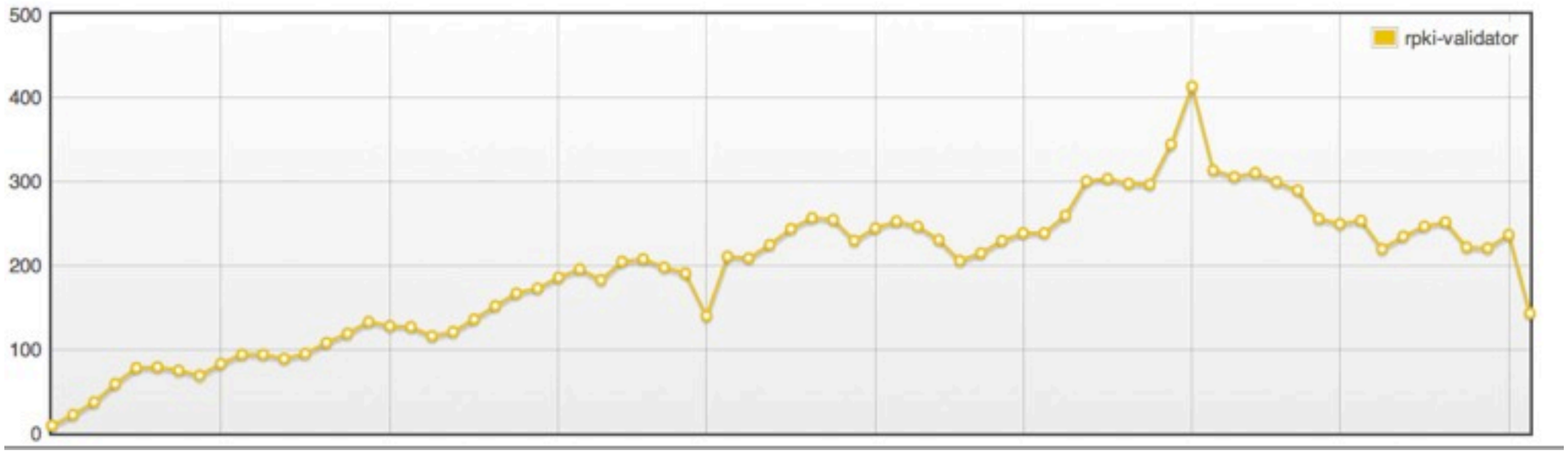
- Validator with feedback enabled
- Sends statistics after each validation run
- Pre-configured with RIR TAs

- Received 15.5k reports from 37 distinct instances across the globe

Measurements in the wild (# reports)

Unique clients reporting per day

Submitted reports per day



Measurements in the wild (reachability)

TA certificate retrieval failures over v4 and v6

	v4	v6
afrinic	0%	31%
apnic	3%	n/a
arin	0%	31%
lacnic-a	0%	50%
lacnic-b	26%	31%
ripe ncc	0%	31%

➡ RPs should work out whether they have good v6 connectivity and avoid wasting time trying (can add a lot of latency)

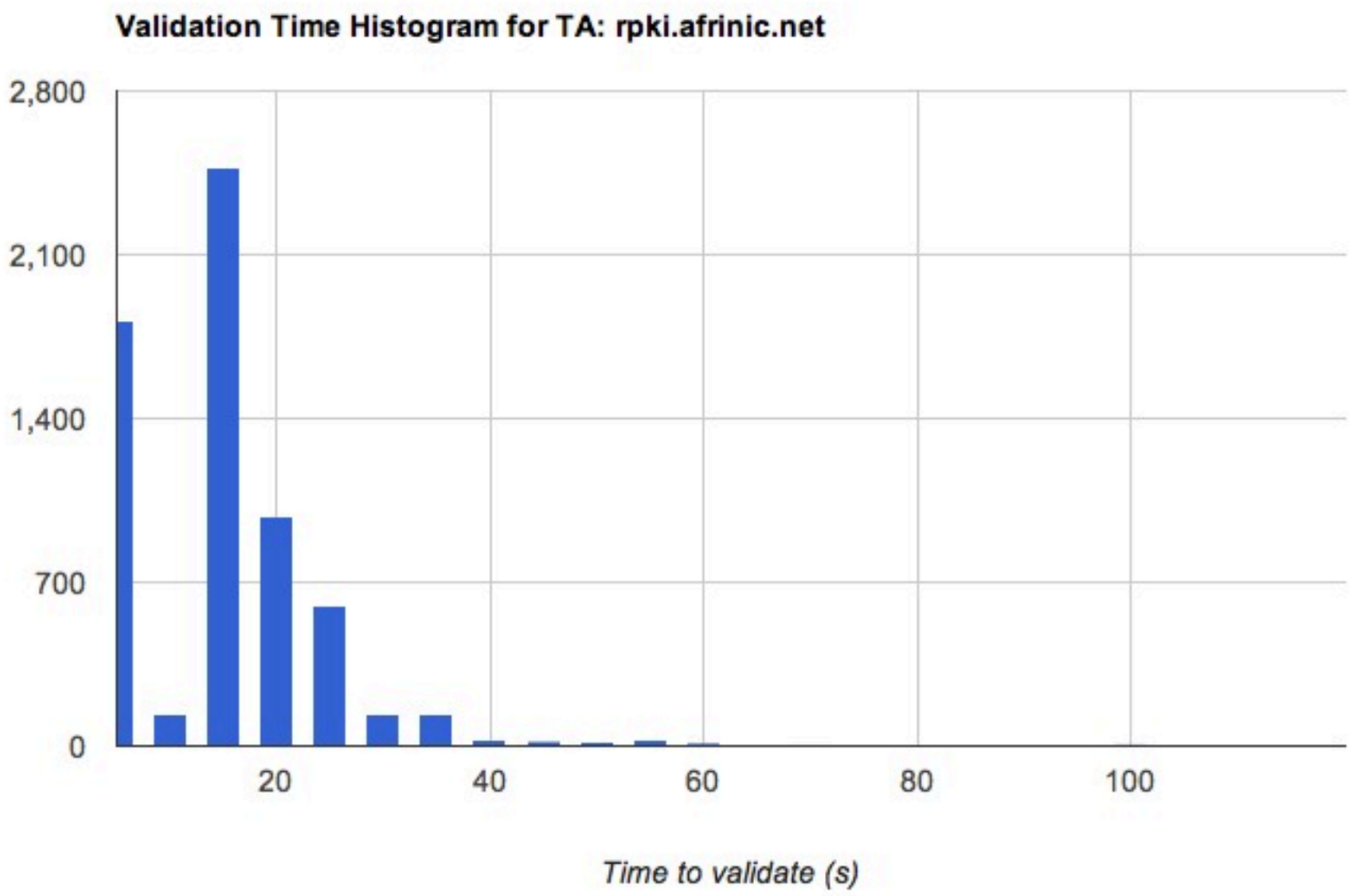
Measurements in the wild (time to validate)

- Hierarchical repository simulated by pre-fetching the repositories

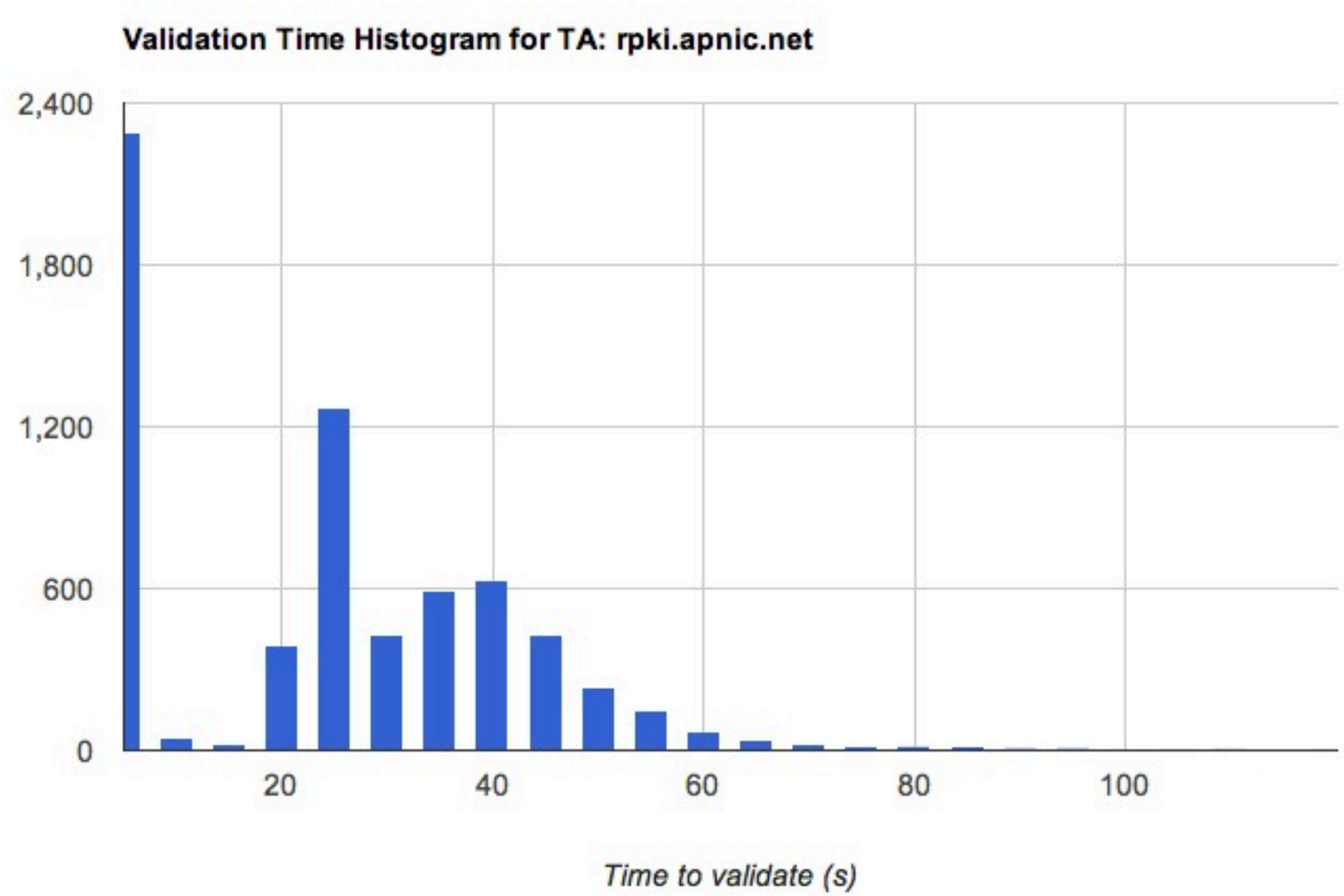
- Very short validation times (<5s) usually due to fetch errors / rejecting of the Trust Anchor or 1st layer CA (RIR online CA)

- Big differences between clients and between runs

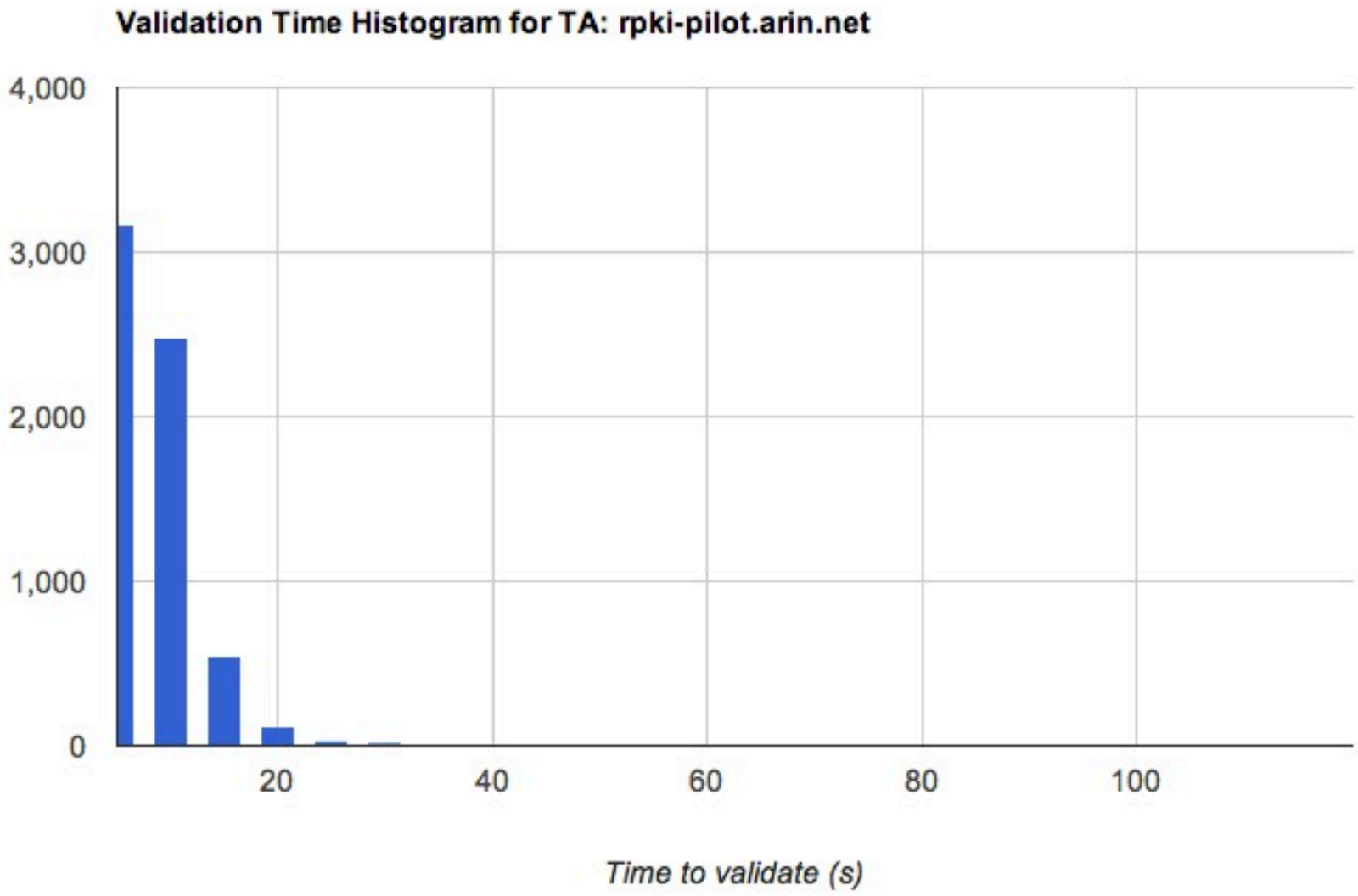
Measurements in the wild (time to validate)



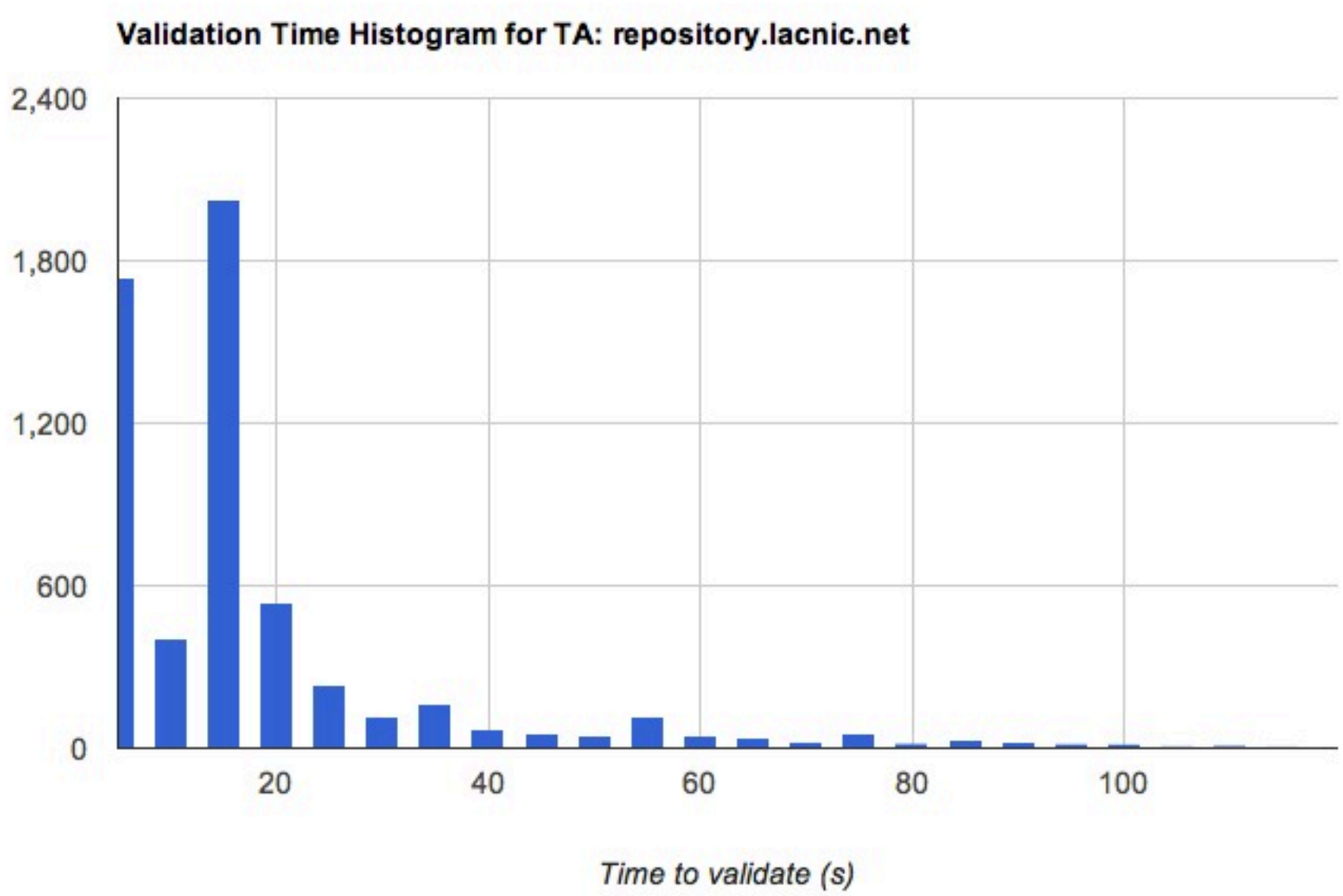
Measurements in the wild (time to validate)



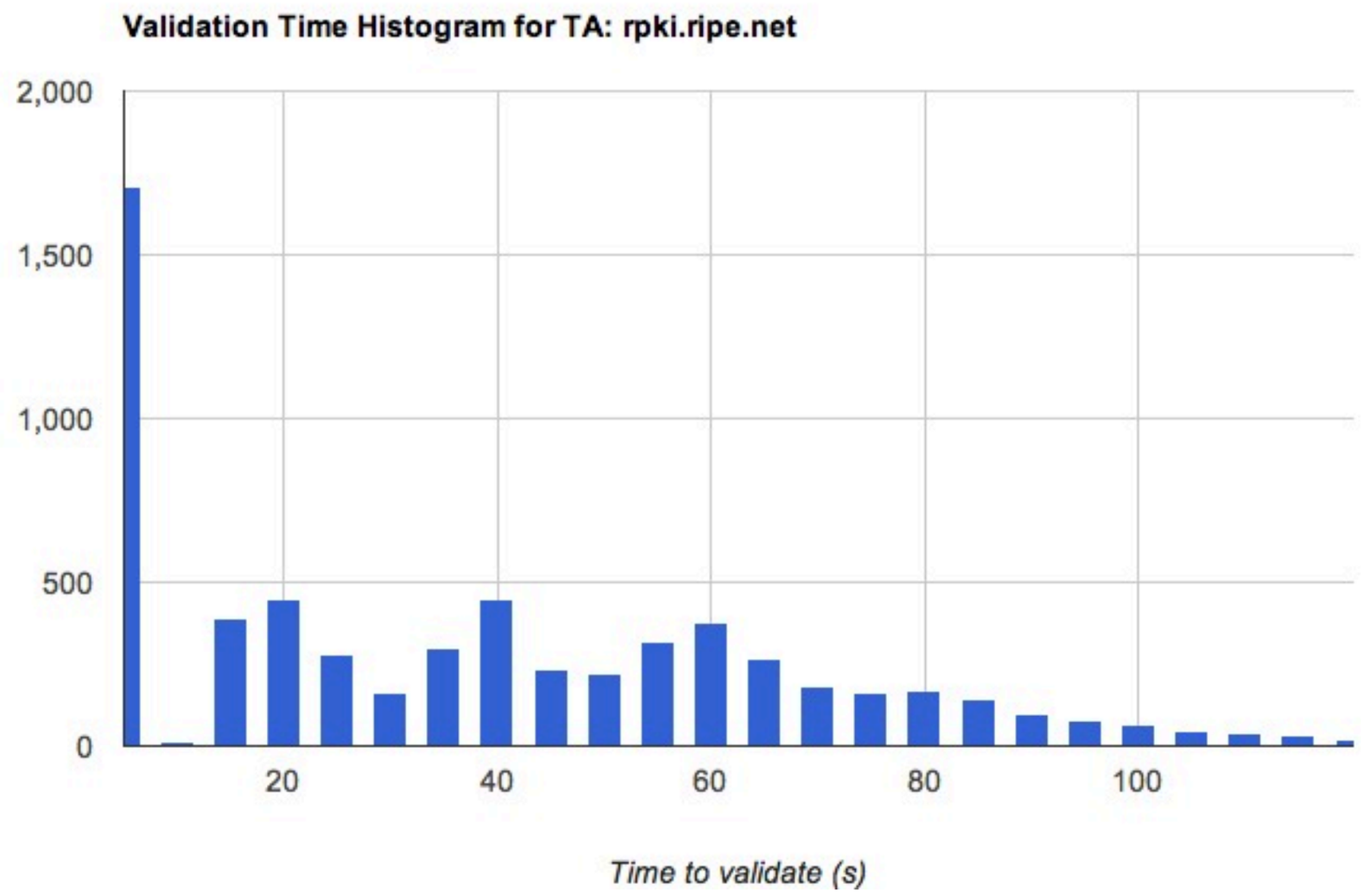
Measurements in the wild (time to validate)



Measurements in the wild (time to validate)



Measurements in the wild (time to validate)



Measurements in the wild (inconsistent mft)

Trust Anchor	runs w. ≥ 1 error vs total runs	Last error seen
AFRINIC	2101 / 4592 (46%)	June 14
APNIC	451 / 4483 (10%)	June 8
ARIN	0 / 4566 (0%)	n/a
LACNIC	3 / 4564 (0.07%)	July 12
RIPE NCC	1 / 4612 (0.02%)	July 5

AFRINIC and APNIC problems due to a bug (now fixed) where the CRL for the new MFT was not published until later.

LACNIC and RIPE NCC inconsistencies due to stuff changing while reading.

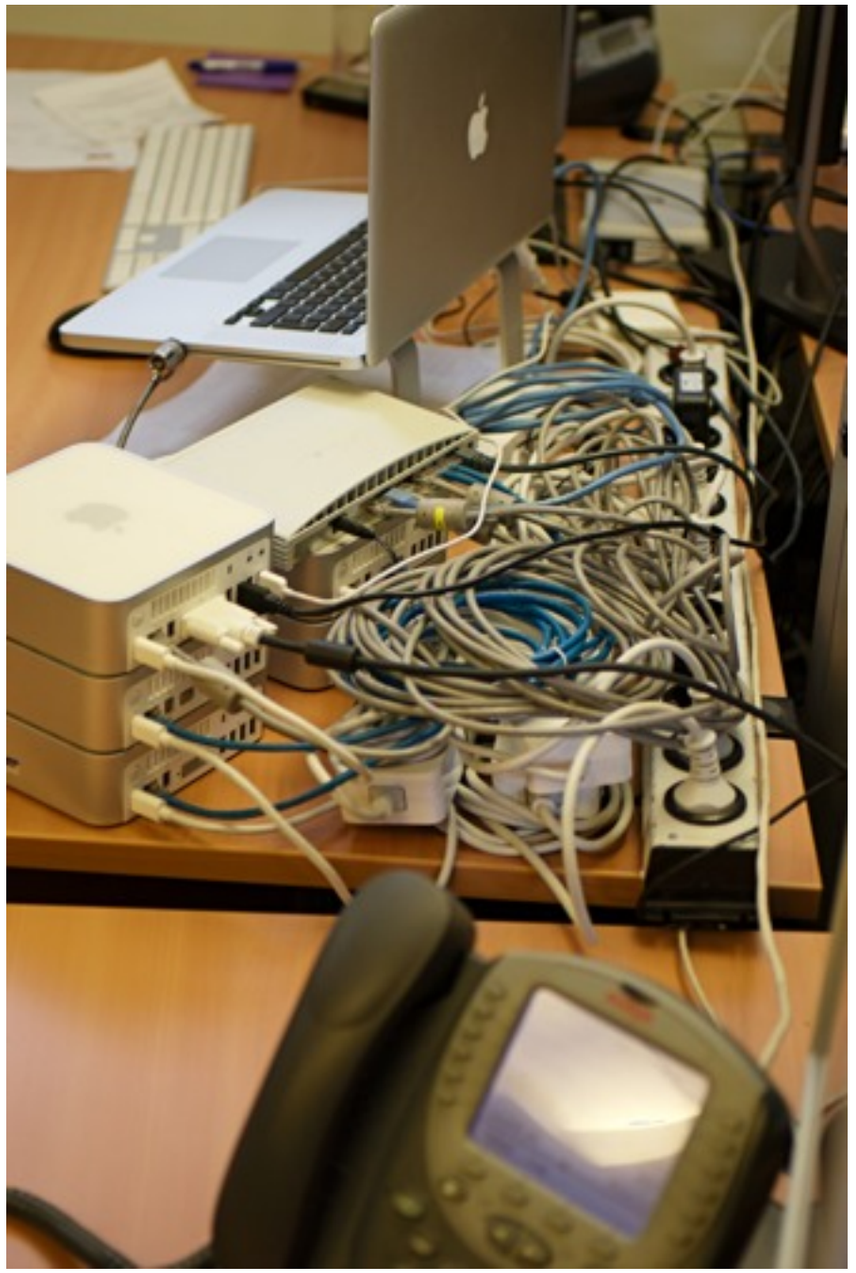
Measurements in the wild (conclusions)

- Difficult to generalize due to differences between clients (cpu, latency, memory) and repository sizes
- Prefetch done outside of standard. Simulates hierarchical lay-out, other RPs will see stronger latency effects especially for bigger repos

➡ Controlled lab more useful to analyze real issues

The lab - Hardware

- 5 Mac Minis
  - 2 GB Memory
  - Core2 CPU @2GHz
  - 2 cores, no hyperthreading
- 1 CA / repository server
- 4 clients
- Ubuntu 12.04 , 3.2.0 kernel



The lab - Full adoption repository

- Real world for RIPE NCC:
  - around 8000 members, plus roughly 25k PI holders
  - roughly 1/4 of the announced routes? Say 100k ROAs
  - or more if multiple pub points are allowed and RIRs publish in all regions for resilience... (400k routes)

- 40k ASNs
  - want to check for updates at least 2-3 times per day
  - but in an ideal world every couple of minutes..

- We are far from full adoption but the protocols and infrastructure must be built to handle it.

The lab - Test Repository

- 1 TA
- 1 top level CA
- 12,000 CAs
- Each CA has around 3 ROAs
- Hierarchical repository lay-out

70k objects

total repository size: 120MB

total size of mfts & crls: 30MB

The lab - rsyncd performance setup 1/2

```
for n in {1..50}; do

   for i in {1..4}; do

      run n full recursive rsyncs on host i &

   done

   time wait     # log total time needed

done
```

The lab - rsyncd performance setup 2/2

rsyncd.conf

```
uid = nobody
gid = nogroup
use chroot = no
max connections = 500
timeout = 600
dont compress = *

[ta]
comment = Benchmark Trust Anchor Repository
path = /export/repository/ta/published
read only = true
list = true

[repository]
comment = Benchmark RPKI Repository
path = /export/repository/online/published
read only = true
list = true
```

For this test we want to see when smoke starts to come out of the mac mini...

In the real world we should also use:

```
refuse options = c delete

or even....
refuse options = r
```

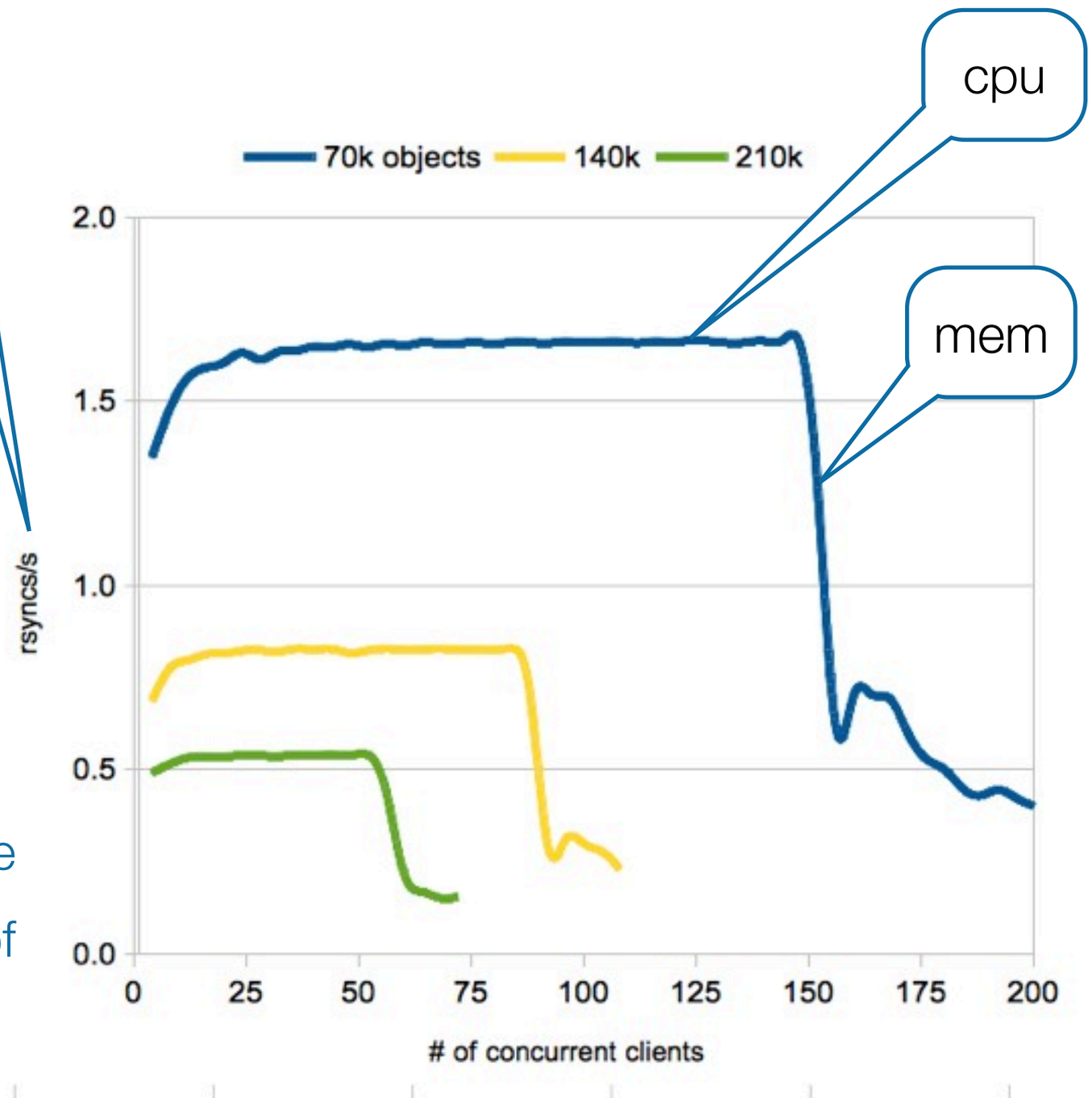
(see next slides)

The lab - rsyncd performance results

throughput of full
rsync update checking

- recursive fetch
- client is up to date
- no latency (local network)
- so just the rsync overhead...

- rsyncd cpu and memory bound
  - cpu 70% system, 30% user
  - needed cache size rel. to repo size
  - rsync mem depends on number of clients (forks) and repo size
  - no more mem -> disk I/O



The lab - simple rsyncd killer script

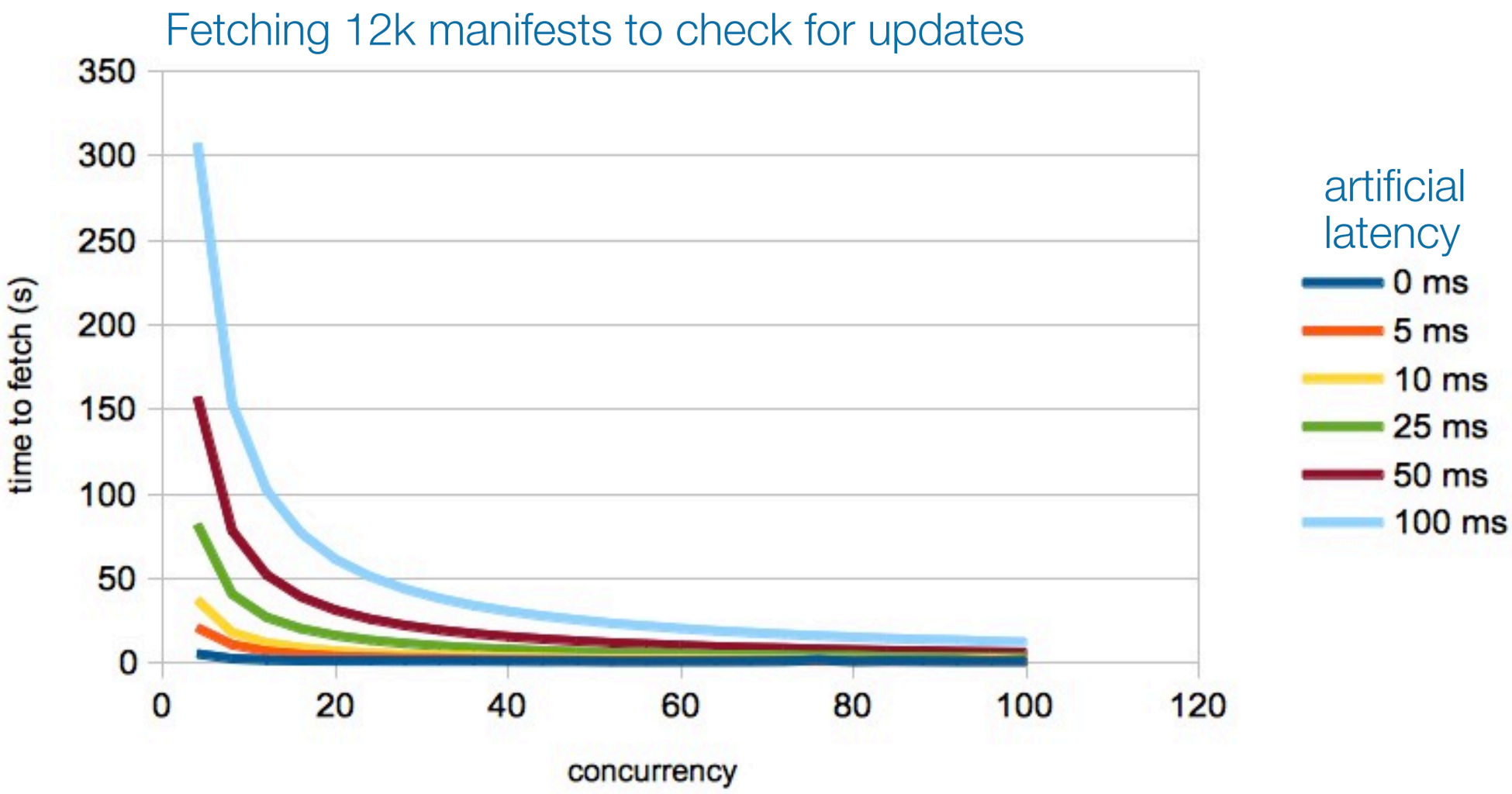
```
for i in {1..500}; do
    rsync --recursive --update --delete \
      --times --bwlimit=100 \
      rsync://host/repository/  &
    sleep 0.01
done
```

To bring the server to its knees the clients just need to connect faster than the server is able to process requests (hence --bwlimit), and make sure that the server has to use a lot of resources (full recursive on complete repository, make server use up its memory).

Server has few options:

- ➡ limit concurrent clients to its max capacity (measure!)
- ➡ disallow recursive fetching

The lab - http



- Latency has a huge impact on performance when fetching objects 1 by 1
- Fetching objects in parallel, or using http pipelining, can reduce this effect dramatically
- More work for client, and highly efficient http servers available

The lab - conclusions - DoS risk/impact

	risk	impact	possible mitigations?
rsync	high	high	multiple servers, limit clients, disable recursive, smart firewalls?
http	low	high	cdn, proxy, cache, fast http servers, smart firewalls

The lab - conclusions rsyncd vs http

	rsync	http
+	➡ built-in deltas nice for relying party	➡ proven scalability ➡ implementation diversity ➡ industry tooling & knowledge ➡ ietf standard ➡ native support in code
-	➡ built-in delta expensive for server, easy to DoS ➡ scaling up with hardware is a costly and losing battle ➡ server may be forced to disallow recursive fetch	➡ need delta protocol to get better relying party performance than parallelisation and pipelining can bring.

next steps

- Fear we may have to turn off recursive fetches when repository grows.
- Deltas.. make http work, or rsync with recursion disabled.
- Updates to RPs in minutes, not hours