Rethinking RPKI data distribution: A different way to split the bill

Job Snijders <job@bsd.nl>

Internet Engineering & Planning Group
IETF 124
Montreal, Canada

Some observations about RPKI object movement

A day in the life of RPKI: October 30st, 2025

- at start 475,434 objects chained up to the 5 Trust Anchors
- every second 2 new objects appeared [1]
- the median object size was 1,924 bytes
- 488,492,596 bytes of raw material moved (175,040 new objects)
- the day ended with 478,682 valid objects
- in raw form, the dataset was 872MiB (compressed 445MiB)
- 4,509 relying party instances [2]

[1] https://miso.sobornost.net/rpki/ccr/2025/10/30/
[2] https://rov-measurements.nlnetlabs.net/stats/

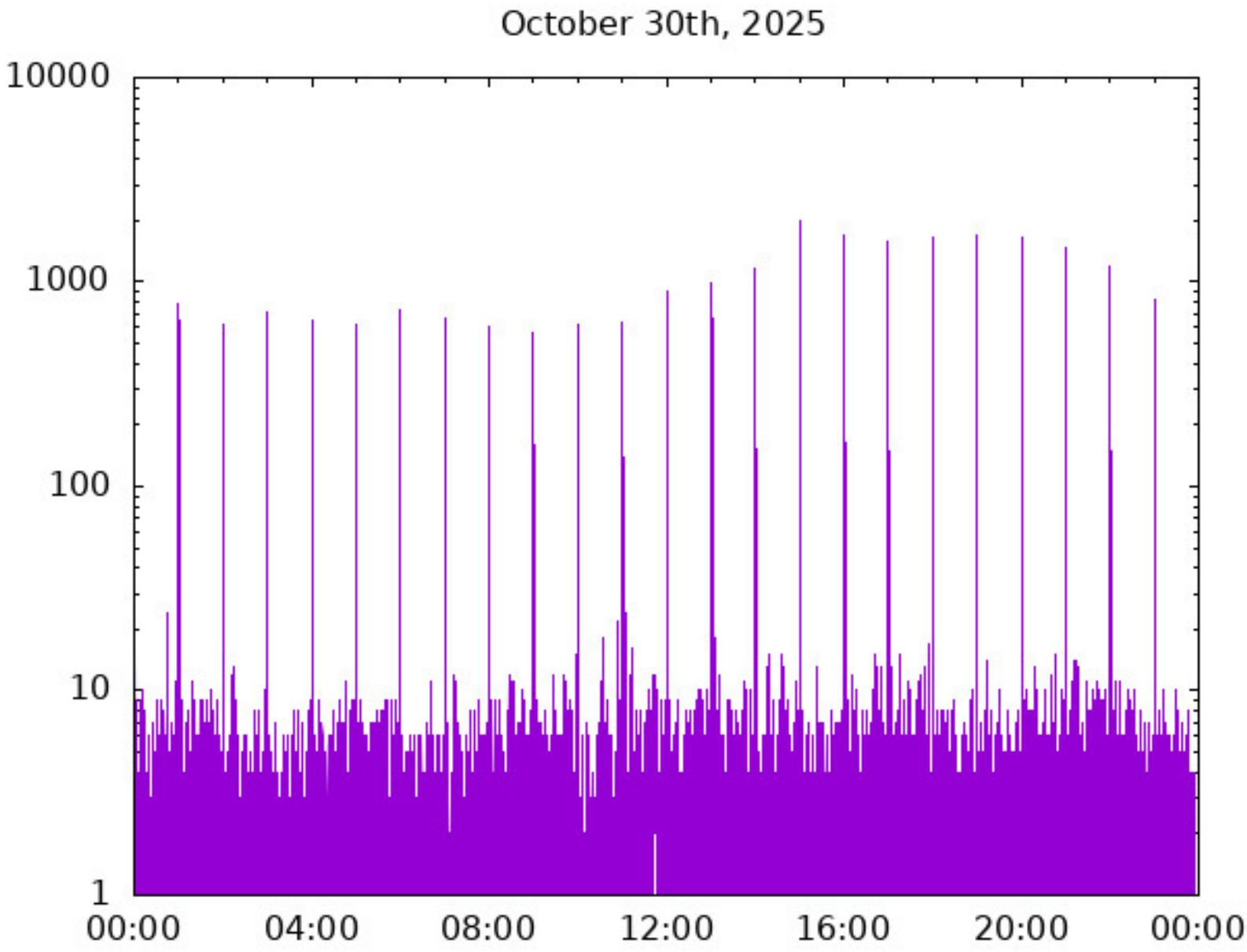

Figure 1: newly discovered RPKI objects (log scale)

The hand we were dealt:

Star topologies without redundancy

- RPKI CAs are "single-homed" to a given publication server
- Every publication server has everything for its constituents
- Every RP fetches everything from every publication server
- Multiple RPs per Autonomous System

Consequences & Impact

RPKI is slow settling sludge. . .

Always: data revocation suffers delays (all caches must see it)

Global scope: publication servers congest from time to time

Localized scope: loss of specific RP / publication server links

Scaling concerns with current protocols

Rsync *TL;DR*: client & server exchange full listing of objects, client then downloads the dataset difference to arrive at the current state.

- the difference is determined on the fly
- quite cheap to transfer the dataset difference
- very expensive to determine the difference

Scaling concerns with current protocols

RRDP *TL;DR*: the server writes repository change operations into a journal, the client downloads this journal as chapters, and then processes chapter after chapter to arrive at the current state.

- clients won't know ahead of time what they'll be downloading
- clients cannot avoid downloading information already overtaken by later events
- excessive bandwidth usage: many paths in the RRDP FSM result in retransfer of data

There isn't a single best choice between Rsync and RRDP

RP network traffic consumption measurements (**bold** is better):

	Rsync	RRDP+Rsync
Handshake	4 MB	**0.5 MB**
Every 15 minutes	40 MB	**5 MB**
Every hour	**50 MB**	100 MB

Clients pick a preferred protocol and try to stick to it, because switching back and forth is costly.

RRDP is extremely salty! Following any type of connection failure, client & server must retransfer the full dataset.

What's been done so far?

Opportunisticly. . .

- Pack more payloads into fewer objects
- Use of deterministic timestamps in *Rsync*
- Use of HTTP-based *RRDP* instead of *Rsync*
- Use of `If-Modified-Since` request header
- Use of compressed HTTP content encoding

What else *could* be done?

Knowing that. . .

Publication server operators choose how they make their data available: through regional or global distribution infrastructure.

The clients have very little choice *where* they fetch data.

What else *could* be done?

> *Validated caches may also be created and maintained from other validated caches. Network operators SHOULD take maximum advantage of this feature to minimize load on the global distributed RPKI database. Of course, the recipient relying parties should re- validate the data.*[3]

[3] RFC 7115, section 3

A different approach: *Erik synchronisation*[4]

Erik synchronisation is a data replication system using the following concepts:

- Merkle trees (1979)
- Content-addressable naming scheme (1950s?)
- Concurrency control through sequence numbers (1975)
- HTTP transport (1989)
- Intermediate nodes called “Erik relays”

[4] Named in honor of Erik Bais who passed away in 2024.

Things to know about Erik synchronisation

- Relays consolidate data from multiple publication servers
- Relays coalesce different RPKI transport protocols
- Relays are spoolers
- Relays can synchronize to other relays
- Clients can use multiple different relays interchangeably
- Relays are operated by third parties

Things to like about Erik Synchronisation

- clients *jump to latest* (similar to Rsync)
- clients only download what changed (similar to Rsync)
- all content is static (just like with RRDP)
- HTTP-based (like RRDP)
- light on state: no persistent sessions (like Rsync, unlike RRDP)
- easy to combine with other protocols (like Rsync, unlike RRDP)
- clients only fetch *newer* data (unlike Rsync & RRDP)
- efficient, fast, cheap

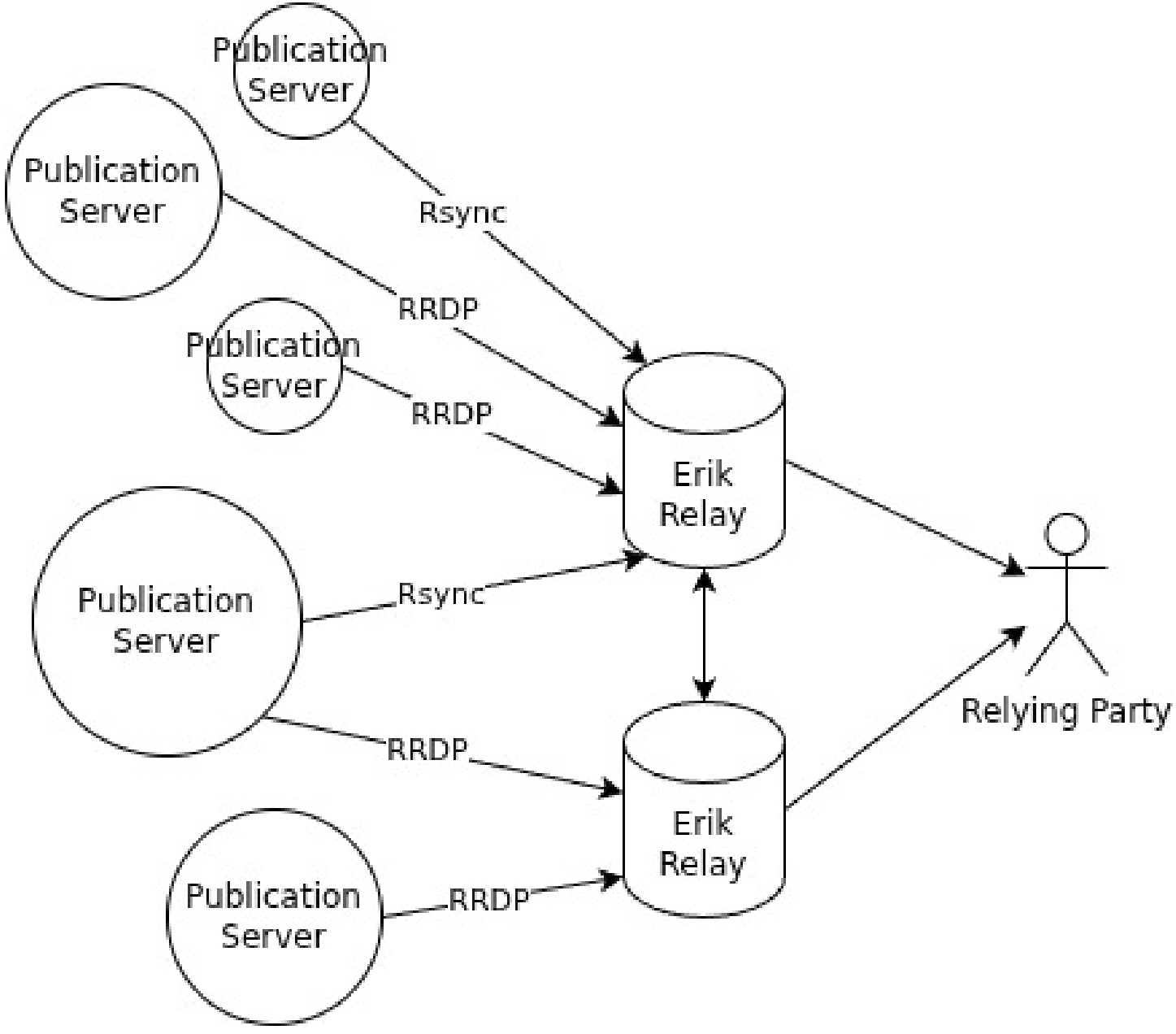

Figure 2: Deployment diagram

Erik Synchronisation: some preliminary results

Network traffic consumption:

	Rsync	RRDP+Rsync	Erik
Handshake	4 MB	0.5 MB	**0.5 MB**
Every 15 minutes	40 MB	5 MB	**4 MB**
Every hour	50 MB	100 MB	**20 MB**

Next steps?

The usual. . .

- Write some relay server & client software
- More testing with global anycast relay `http://relay.rpki-servers.org/`
- Update `draft-spaghetti-sidrops-erik-protocol`
- Iterate until happy
- WG Adoption → Implementation Reports → WGLC → onwards. . .
- Deployment strategy / endgoal: *enabled by default* in RPs?

Who is going to run the relays?

Tentatively planned...

- RIPE NCC?
- Amazon?
- Cloudflare?
- You?

Think of Erik relays as `1.1.1.1`, `8.8.8.8`, or `9.9.9.9`, but ... for the RPKI!

The end! Buy me a cup of coffee?

job@bsd.nl