Linux Network Testing and Optimisation Techniques for Elasticsearch Clusters

May 17, 2026 / Gavin Jackson

linux networking elasticsearch logstash rhel performance observability nmon capacity-planning

A busy logging platform rarely fails in a glamorous way.

It normally starts with a small delay in dashboards, a few Logstash retries, a queue that grows during the business day but drains overnight, or Elasticsearch bulk requests that occasionally come back slower than expected. Then someone asks the obvious question: is the network fast enough?

Imagine a fairly common setup:

- three Elasticsearch nodes
- two Logstash servers
- more than 100 servers sending logs into the platform
- sustained ingest, with bursts during deployments, outages, batch jobs, and security events

That is exactly the kind of environment where network tuning matters, but it is also the kind of environment where tuning the wrong thing can waste a lot of time. A Linux server with a 10 GbE or 25 GbE NIC can still perform poorly if interrupts are pinned badly, NIC queues are undersized, switch links are dropping frames, Logstash is sending too many small bulk requests, or all traffic is accidentally landing on one Elasticsearch node.

The goal is not to collect sysctl settings like trading cards. The goal is to prove where the bottleneck is, remove avoidable packet loss, and make the network boring.

Understand the traffic first

In this sort of logging design, there are two important traffic paths.

The first path is from the estate into Logstash:

```
100+ servers -> Logstash nodes
```

This might be Beats traffic, syslog, HTTP, TCP, UDP, or a mixture. If the sending agents have disk-backed queues, this path is usually forgiving. If the traffic is UDP syslog, it is much less forgiving because drops may simply vanish.

The second path is from Logstash into Elasticsearch:

```
Logstash nodes -> Elasticsearch HTTP bulk API -> Elasticsearch transport traffic
```

Logstash writes to Elasticsearch over HTTP, normally to port `9200`, using the Elasticsearch output plugin. Elasticsearch then uses its transport layer, normally port `9300`, for node-to-node traffic, shard replication, cluster state, recoveries, and internal coordination.

That means a single incoming log event can create more than one network movement:

- from the original server to Logstash
- from Logstash to an Elasticsearch node
- from the primary shard to replica shards
- during recovery or rebalancing, between Elasticsearch nodes again

Before changing anything, draw the actual traffic flow. Which VLAN carries Logstash output? Which VLAN carries Elasticsearch transport traffic? Are the Logstash hosts writing to all three Elasticsearch nodes, or only one? Are the Elasticsearch nodes in the same rack, across racks, or across a routed fabric? Are you using TLS, compression, jumbo frames, or a load balancer?

The network shape matters as much as the network speed.

Start with Elasticsearch and Logstash basics

It is easy to blame Linux networking when the real issue is ingest behaviour.

Elastic's own indexing speed guidance starts with bulk requests, worker concurrency, refresh intervals, replicas during initial loads, filesystem cache, shard layout, and hot spotting. That is the right order. A perfect NIC cannot fix undersized shards, saturated disks, too many concurrent writers to one shard, or Logstash sending badly shaped batches.

For Logstash, the Elasticsearch output plugin is also worth reading closely. A few details matter for network planning:

- The plugin sends batches to the Elasticsearch Bulk API.
- Very large batches are split when they exceed 20 MB.

- Using one Elasticsearch output is usually more efficient than creating many outputs, because each output creates its own client and connection pool.
- Modern plugin documentation describes request compression through `compression_level`, which can reduce network IO at the cost of CPU.
- DNS caching and long-lived keepalive connections can affect failover and traffic distribution.

That leads to some practical checks:

- Configure Logstash outputs with all suitable Elasticsearch endpoints, not just `es01`.
- Watch for Elasticsearch `429` responses, rejected writes, indexing pressure, and long bulk latencies.
- Avoid one hot Elasticsearch node receiving most of the Logstash traffic.
- Keep Logstash persistent queues enabled if the platform must absorb short Elasticsearch or network interruptions.
- Tune Logstash batch and worker settings alongside Elasticsearch bulk sizing, not separately from it.

Elasticsearch networking settings should normally stay simple. The Elasticsearch network settings documentation recommends `network.host` for the common case where a node binds and publishes one address. Advanced bind and publish settings are useful for multi-homed hosts, but they are also a good way to create confusing cluster behaviour if DNS, routing, or interface selection is wrong.

For a production cluster, be deliberate:

```
network.host: 10.20.30.11
http.port: 9200
transport.port: 9300
```

Do not use `0.0.0.0` casually on a multi-homed Elasticsearch node. It can be fine for binding in controlled cases, but the publish address must be reachable by the nodes and clients that need it. If the node publishes the wrong interface, the network can look broken even when the OS is doing exactly what it was told.

Measure before tuning

I would split testing into three layers.

First, check the physical and OS layer:

```
ip -br link
ip -s link show dev ens1f0
ethtool ens1f0
ethtool -S ens1f0 | egrep -i 'drop|discard|error|timeout|miss|fifo|collision'
ethtool -g ens1f0
ethtool -k ens1f0
ethtool --show-channels ens1f0
```

You are looking for boring fundamentals: expected link speed, full duplex, no physical errors, no growing drop counters, sensible ring sizes, enabled offloads, and enough NIC queues for the hardware.

Second, check the kernel and socket layer:

```
ss -s
ss -nti
nstat -az Tcp\* Ip\*
awk '{for (i=1; i<=NF; i++) printf strtonum("0x" $i) (i==NF ? "\n" : " ")}' /proc/net/softnet_stat |
column -t
mpstat -P ALL 1
sar -n DEV,TCP,ETCP 1
```

The most useful symptoms here are retransmits, receive queues that do not drain, softirq pressure on a small number of CPUs, and kernel backlog drops. If `ss -nti` shows retransmits on Logstash-to-Elasticsearch connections, that is more useful than a vague feeling that "the network is slow".

Third, test the application layer:

```
curl -s https://es01.example.net:9200/_nodes/stats/http,transport,indices,thread_pool?pretty
curl -s https://es01.example.net:9200/_cat/nodes?v
curl -s https://es01.example.net:9200/_cat/thread_pool/write?v
curl -s http://logstash01.example.net:9600/_node/stats/pipelines?pretty
```

The exact Elasticsearch thread pool names can vary across major versions, so check your version rather than copying old examples blindly. What matters is the pattern: correlate network counters with bulk latency, write rejections, indexing rate, queue growth, and node-level hot spots.

Use iperf3, but use it properly

The first active test tool to reach for is `iperf3`.

The iperf3 project describes it as a tool for measuring maximum achievable bandwidth on IP networks, with support for TCP, UDP, SCTP, buffer tuning, zero-copy, and JSON output. Red Hat's RHEL network performance documentation also uses `iperf3` for TCP throughput testing, while warning that synthetic test results can differ from real application throughput.

That warning matters. `iperf3` does not tell you how fast Elasticsearch can index. It tells you whether two hosts can move traffic cleanly across the network path.

Install the usual toolbox on the test hosts:

```
dnf install iperf3 ethtool sysstat pcp tuned bcc-tools tcpdump
```

Temporarily open the iperf3 port on the receiver:

```
firewall-cmd --add-port=5201/tcp --timeout=1h
iperf3 --server
```

Then test from each Logstash node to each Elasticsearch node:

```
iperf3 --client es01.example.net --time 60 --omit 5
iperf3 --client es01.example.net --parallel 4 --time 60 --omit 5
iperf3 --client es01.example.net --parallel 8 --time 60 --omit 5 --reverse
iperf3 --client es01.example.net --parallel 8 --time 60 --omit 5 --bidir
iperf3 --client es01.example.net --parallel 8 --time 60 --omit 5 --json
```

I like to run tests in a matrix:

Source	Destination	Test
`logstash01`	`es01`, `es02`, `es03`	single stream, parallel streams, reverse
`logstash02`	`es01`, `es02`, `es03`	single stream, parallel streams, reverse
`es01`	`es02`, `es03`	transport network tests
`es02`	`es01`, `es03`	transport network tests
`es03`	`es01`, `es02`	transport network tests

Single-stream testing is important because one TCP flow will often be limited by one CPU path, one NIC queue, one bond member, or one switch hash decision. Parallel-stream testing is important because Logstash and Elasticsearch normally use multiple connections and multiple flows.

For higher speed networks, Red Hat documents `--zerocopy` as useful when simulating zero-copy-capable applications or trying to reach very high single-stream throughput:

```
iperf3 --client es01.example.net --time 60 --omit 5 --zerocopy
```

Do not run one test and declare victory. Run a baseline, make one change, run the same test again, and keep the JSON results.

When NIC bonding helps

Bonding NICs can help in two different ways: availability and aggregate throughput.

Those are not the same thing.

On RHEL, Red Hat's network bonding documentation describes bonding as a way to aggregate interfaces into one logical interface for higher throughput or redundancy. It can be configured with `nmcli`, the RHEL web console, `nmtui`, `nmstatectl`, or RHEL system roles.

The practical modes to think about are:

Bond mode	What it is good for	Switch requirement
`active-backup`	Redundancy. One link active, another takes over if it fails.	No special switch configuration.
`802.3ad` / LACP	Aggregate capacity across multiple flows and link redundancy.	LACP port channel on the switch.
`balance-xor`	Static aggregation with hash-based distribution.	Static EtherChannel, not LACP.
`balance-rr`	Round-robin packet distribution.	Static EtherChannel, but not a good default for ordered TCP workloads.

For Elasticsearch and Logstash, `active-backup` is excellent when resilience matters more than throughput. It will not make one Logstash-to-Elasticsearch TCP connection faster, because only one physical link is active at a time.

LACP is the more interesting option for throughput. With `802.3ad`, the bond can distribute different flows across different physical links. This can help a logging stack because you have multiple Logstash hosts, multiple Elasticsearch nodes, HTTP bulk connections, and Elasticsearch transport connections. The aggregate traffic can spread.

But there is a catch: a single TCP flow usually lands on one member link. If you bond two 10 GbE ports with LACP, do not expect one TCP stream to become 20 Gbps. You should expect many flows to have up to 20 Gbps of aggregate headroom, if the switch and host hashing distribute those flows well.

That is why the `xmit_hash_policy` matters. Red Hat documents `layer3+4` as a transmit hash policy that considers IP addresses and ports for port selection. For Logstash and Elasticsearch, where the same hosts may maintain multiple TCP connections, that can distribute traffic better than a policy that only considers MAC addresses or IP addresses. The switch side has to be compatible with the design.

A RHEL 9.4 or later `nmcli` example for an LACP bond looks like this:

```
nmcli connection add type bond con-name bond0 ifname bond0 bond.options
"mode=802.3ad,miimon=100,xmit_hash_policy=layer3+4"
nmcli connection add type ethernet port-type bond con-name bond0-eno1 ifname eno1 controller bond0
nmcli connection add type ethernet port-type bond con-name bond0-eno2 ifname eno2 controller bond0
nmcli connection modify bond0 ipv4.addresses 10.20.30.11/24 ipv4.gateway 10.20.30.1 ipv4.method
manual
nmcli connection modify bond0 connection.autoconnect-ports 1
nmcli connection up bond0
cat /proc/net/bonding/bond0
```

On older RHEL versions, the `nmcli` terminology may use `master`, `slave-type`, and `connection.autoconnect-slaves` instead of `controller`, `port-type`, and `connection.autoconnect-ports`.

Two cautions are worth spelling out.

First, do not use NIC teaming for a new RHEL 9 design. Red Hat marks NIC teaming as deprecated in RHEL 9 and recommends the bonding driver instead.

Second, bonding is not a substitute for buying the right NIC speed. If the cluster needs deterministic throughput above 10 Gbps between any two nodes, 25 GbE, 40 GbE, or 100 GbE is usually cleaner than hoping a bond will make every flow faster.

Jumbo frames: useful, but only when boring

Jumbo frames can reduce packet overhead and CPU work for large contiguous data streams. Red Hat's RHEL performance documentation notes that a 9000 byte MTU reduces Ethernet frame overhead compared with the standard 1500 byte payload.

That sounds attractive for Elasticsearch bulk traffic and shard recovery traffic. It can be, especially on a dedicated backend network.

The problem is that every device in the path must agree: source NIC, switch ports, port channels, VLANs, routed interfaces, firewalls, destination NIC, and sometimes virtual switches. A partial jumbo-frame configuration is worse than no jumbo frames because it creates fragmentation, drops, and strange latency.

If you enable jumbo frames, do it on a dedicated ingest or cluster transport network and test it explicitly:

```
nmcli connection modify bond0 mtu 9000
nmcli connection up bond0
ip link show dev bond0
nstat -az IpReasm\*
ping -c1 -Mdo -s 8972 es02.example.net
```

The ping payload calculation for IPv4 is `MTU - 8 bytes ICMP header - 20 bytes IPv4 header`, so `8972` is the common test size for a 9000 byte MTU.

If there is any doubt, stay at 1500 MTU until you can test the whole path properly.

RHEL tools that are worth using

RHEL has a very good network performance toolbox, and the best part is that much of it integrates with NetworkManager rather than disappearing after reboot.

TuneD

`tuned` is the first RHEL-specific tool I would check.

Red Hat's TuneD documentation for RHEL 9 lists `network-throughput` as a profile for streaming network throughput and `network-latency` as a profile focused on low-latency network performance.

For Elasticsearch ingest nodes, `network-throughput` is usually the more natural starting point:

```
dnf install tuned
systemctl enable --now tuned
tuned-adm list
tuned-adm active
tuned-adm profile network-throughput
tuned-adm verify
```

Do this in a maintenance window and measure before and after. TuneD profiles are useful, but they are still system-wide behaviour changes.

NetworkManager and ethtool settings

The old habit was to run `ethtool -G` or `ethtool -K`, then forget that the setting would vanish on reboot. RHEL's NetworkManager ethtool settings avoid that by storing offload, coalescing, ring buffer, and channel settings in the connection profile.

Useful checks:

```
ethtool -S ens1f0 | egrep -i 'drop|discard|error|miss|timeout'
ethtool -g ens1f0
ethtool -k ens1f0
ethtool --show-coalesce ens1f0
ethtool --show-channels ens1f0
```

If RX drops are rising and the NIC supports larger rings:

```
nmcli connection modify bond0 ethtool.ring-rx 4096
nmcli connection modify bond0 ethtool.ring-tx 4096
nmcli connection up bond0
```

If interrupt rate is too high and CPU is spending too much time handling packets, interrupt coalescing can improve throughput:

```
nmcli connection modify bond0 ethtool.coalesce-rx-frames 128
nmcli connection up bond0
```

That can add latency, so test it with real ingest traffic. For logging pipelines, a tiny increase in latency may be acceptable if it buys lower CPU usage and fewer drops. For latency-sensitive request paths, it might not be.

If the NIC has more channels available than it is using:

```
ethtool --show-channels ens1f0
nmcli connection modify bond0 ethtool.channels-combined 8
nmcli connection up bond0
```

Do not blindly set channels to the maximum. Match the NIC, CPU topology, interrupt distribution, and workload.

irqbalance

On RHEL, `irqbalance` is enabled by default and should normally stay enabled. Red Hat warns that disabling it can hurt network throughput. If one CPU is doing all interrupt work, network performance can suffer even when the link itself is fine.

Check it:

```
systemctl status irqbalance
cat /proc/interrupts
mpstat -P ALL 1
```

If CPU 0 is much busier than everything else during network tests, interrupts and queues are worth investigating.

PCP and sysstat

For historical evidence, use `sysstat` and Performance Co-Pilot.

`sar` is still excellent for quick before-and-after work:

```
sar -n DEV,TCP,ETCP 1
sar -n SOCK 1
```

Performance Co-Pilot is more powerful for longer investigations. Red Hat's PCP data sheet describes it as a supported system-level performance monitoring suite with live and archived metrics, broad Linux coverage, and tools that overlap with familiar utilities such as `iostat`, `pidstat`, `vmstat`, and `mpstat`.

For an Elasticsearch cluster, PCP is useful because the performance problem may have happened at 2:00 AM during an index rollover, shard recovery, or log storm. Live commands are nice. Archived metrics are better.

> #### *nmon and old-school capacity planning*
>
> *Before every platform had a time-series database attached to it, `nmon` was one of the nicer ways to get long-running performance evidence from Unix and Linux systems.*
>
> *The appeal was simple: start a collector, let it run through real business cycles, then analyse the generated files later. That made it useful for capacity planning because you could capture quiet periods, daily peaks, backup windows, month-end processing, and the awkward spikes that never happen while someone is watching a terminal.*
>
> *A typical collection pattern looked like this:*
>
> ```
> nmon -f -s 60 -c 1440
> ```

> *That records one sample every 60 seconds for 24 hours. For longer studies, you would schedule it from cron, keep the `.nmon` files, and compare days or weeks rather than arguing from one busy five-minute sample.*
>
> *The reporting workflow was also very practical. You could feed the generated files into a spreadsheet-based analyser, often a custom Excel workbook, and turn raw counters into charts for CPU, disk, memory, paging, network, and process behaviour. In more automated environments, the same kind of data could be pushed into an RRDTool-backed web view so teams could browse historical graphs without passing spreadsheets around.*
>
> *The lesson still applies even if the modern tooling is PCP, Prometheus, Grafana, Elastic monitoring, or a vendor platform: capacity planning needs history. One `iperf3` run can prove a network path. It cannot tell you whether the cluster runs out of IO every weekday at 9:15 AM.*
>
> *For this kind of troubleshooting, I think of the core tuning loop as an I/O triangle:*
>
> - ***Disk**: indexing, merging, translog writes, shard recovery, and queue spillover all become disk problems eventually.*
> - ***Network**: Logstash bulk requests, Elasticsearch transport traffic, replication, and recovery all need clean paths with low retransmits and no drops.*
> - ***Memory**: filesystem cache, JVM heap, Logstash queues, socket buffers, and paging decide whether the system absorbs bursts or thrashes.*
>
> *CPU still matters, but often as the tax paid to compress, encrypt, copy, interrupt, parse, merge, and garbage collect. If disk, network, and memory are all healthy, CPU tuning becomes much easier to reason about.*

BCC/eBPF tools

RHEL also ships BCC tooling for low-overhead network tracing. Red Hat's BCC network tracing documentation includes tools for TCP drops, retransmits, connection latency, TCP session summaries, softirq time, and per-connection throughput.

Useful examples:

```
/usr/share/bcc/tools/tcpretrans
/usr/share/bcc/tools/tcpdrop
/usr/share/bcc/tools/tcptop
/usr/share/bcc/tools/tcplife
/usr/share/bcc/tools/softirqs
```

These are not first-line tuning tools. They are excellent when normal counters say "there are retransmits" but not "which connections are doing it and when".

TCP buffers, backlog, and drops

RHEL defaults are good for most systems, but high-throughput ingest can still hit kernel limits.

Red Hat's RHEL 9 network performance tuning documentation highlights several areas that matter for fast NICs:

- NIC ring buffers
- network device backlog queues
- SoftIRQ budget
- TCP socket buffers
- TCP window scaling
- TCP SACK
- TCP timestamps
- Ethernet flow control

The important part is to tune based on counters.

If `/proc/net/softnet_stat` shows the second column incrementing over time, the kernel backlog queue is dropping frames. Red Hat suggests increasing `net.core.netdev_max_backlog` progressively and verifying that the counter stops increasing.

If the third column increments, SoftIRQ processing may not be getting enough budget. Red Hat documents `net.core.netdev_budget` and `net.core.netdev_budget_usecs` for that case.

If `ss -nti` shows receive queues growing while the application is not reading fast enough, the problem may be application CPU, JVM pauses, Logstash backpressure, or socket buffers. Do not assume it is the switch.

I would avoid throwing a generic "high performance sysctl.conf" at Elasticsearch. It is much better to record the counter that proves the problem, change one setting, and then record the counter again.

A practical optimisation sequence

If I were working through this cluster, I would go in this order.

1. Prove the baseline

Record:

- NIC model, driver, firmware, speed, duplex
- switch port configuration
- MTU
- bond mode and hash policy
- Elasticsearch `network.host`, `http.port`, and `transport.port`
- Logstash Elasticsearch output hosts
- current TuneD profile
- `irqbalance` status
- NIC counters before and after a busy period

This is dull work, which is exactly why it pays off.

2. Remove obvious design bottlenecks

Make sure both Logstash servers can write to all appropriate Elasticsearch endpoints. Avoid routing all bulk traffic through one node unless it is a deliberate coordinating or load-balanced design.

If Elasticsearch nodes have multiple interfaces, make sure HTTP and transport publish addresses are correct and stable.

If the switch uplinks are oversubscribed, no Linux tuning will make that disappear.

3. Test every important path with iperf3

Run single-stream and parallel-stream tests between:

- each Logstash host and each Elasticsearch host
- each Elasticsearch host and every other Elasticsearch host

Save JSON output where possible. Test both directions. Run tests while watching `sar`, `mpstat`, `ethtool -S`, and switch counters.

4. Fix physical and link-layer errors first

Any growing CRC, frame, carrier, FIFO, missed, or dropped counters deserve attention before sysctl tuning. Replace cables, check optics, verify switch port settings, check LACP state, and update NIC firmware or drivers if necessary.

5. Choose the right bonding mode

Use `active-backup` if the requirement is failover.

Use `802.3ad` if the requirement is aggregate throughput across many flows and the switch is configured for LACP.

Use `layer3+4` hashing only after confirming it fits the switch and network design.

Do not expect bonding to make one TCP flow faster than one member link.

6. Apply RHEL tuning carefully

For throughput-oriented ingest nodes:

```
tuned-adm profile network-throughput
```

For rising NIC drops:

```
ethtool -S ens1f0 | egrep -i 'drop|discard|error'
ethtool -g ens1f0
nmcli connection modify bond0 ethtool.ring-rx 4096 ethtool.ring-tx 4096
nmcli connection up bond0
```

For interrupt and queue issues:

```
systemctl enable --now irqbalance
ethtool --show-channels ens1f0
mpstat -P ALL 1
cat /proc/interrupts
```

For jumbo frames:

```
nmcli connection modify bond0 mtu 9000
nmcli connection up bond0
ping -c1 -Mdo -s 8972 es02.example.net
nstat -az IpReasm\*
```

Make one change at a time. Keep the before-and-after output.

7. Validate with real ingest

Once the network path looks clean, validate with Elasticsearch itself.

Rally is the proper Elasticsearch benchmarking tool when you want repeatable indexing and search tests. It can run benchmarks, record results, compare runs, and attach telemetry. Use `iperf3` to prove the network path; use Rally or a controlled Logstash replay to prove ingest performance.

During the test, watch:

- Logstash pipeline throughput and queue growth
- Elasticsearch indexing throughput
- bulk request latency
- write rejections or `429` responses
- node CPU and disk IO
- network retransmits
- NIC drops
- shard recovery or relocation traffic

If improving the network does not improve ingest, the bottleneck is probably somewhere else. That is still a useful result.

What I would recommend for the three-node cluster

For a three-node Elasticsearch cluster fed by two Logstash servers, my default recommendation would be:

- Use at least 10 GbE for the Logstash-to-Elasticsearch and Elasticsearch transport paths; prefer 25 GbE or faster if daily ingest is high or recoveries must complete quickly.
- Put Elasticsearch transport traffic on a reliable, low-latency backend VLAN or subnet where possible.
- Configure Logstash to use all suitable Elasticsearch HTTP endpoints.

- Use `active-backup` bonding where resilience is the main goal.
- Use LACP bonding where aggregate throughput across many flows is needed, with switch configuration and hash policy tested.
- Avoid NIC teaming on new RHEL 9 builds; use bonding.
- Consider jumbo frames only on a fully controlled path.
- Use `network-throughput` TuneD as a tested profile, not as folklore.
- Keep `irqbalance` enabled unless there is a very specific reason not to.
- Use NetworkManager to persist `ethtool` ring, coalescing, offload, and channel settings.
- Use PCP or sysstat to keep historical evidence.
- Use BCC tools when you need to trace TCP retransmits, drops, and connection behaviour.

Most importantly, keep the work evidence-driven.

The best network optimisation is not a magic set of kernel parameters. It is a repeatable loop:

```
measure -> change one thing -> test again -> keep or revert -> document
```

In an Elasticsearch logging platform, that discipline matters because the network is only one part of the ingest path. Logstash batching, Elasticsearch shard layout, disk IO, JVM heap, filesystem cache, replicas, refresh intervals, and cluster hot spots can all look like "network slowness" from a distance.

Get close enough to the problem, and the tuning usually becomes much less mysterious.
