Rate Limiting in Ficcus

ccertain *

October 25, 2026

1 Sketch

A ficcus node limits three things: how often an address may open connections to it, how fast a connected peer may send messages, and how fast the namespaces of one signer may be created. Connections and namespaces are priced in proof of work whose difficulty rises with use and decays with time; messages are counted against a budget and a peer that overspends is disconnected.

Identities are free to make, so nothing here limits how many signers there are. Connections are limited per address, which an attacker has fewer of; namespaces are limited per signer, and their proof of work is collected from many nodes at creation and carried with the namespace, so a node far from where a namespace was made can still check that it was paid for. Which nodes' challenges count is decided by how long a node has known them (Section 5).

The implementation is the `ficcus-pow` crate (the primitive, Section 2), `ficcus/src/network/rate_limit.rs` (connections), `ficcus/src/network/connection.rs` (the handshake and the message budget), `ficcus/src/network/uptime.rs` (ratings) and `ficcus/src/namespace/creation.rs` (namespaces).

2 Proof of Work

Puzzle. Let H be BLAKE3 and $D =$ `"ficcus-pow v1"`. For a seed $s \in \{0,1\}^{256}$ and a nonce $n \in [0, 2^{64})$ the hash value is the first 16 bytes of $H(D \,\|\, s \,\|\, \mathrm{le}_{64}(n))$ read as a big endian integer $h(s,n) \in [0, 2^{128})$. A challenge is a pair (s,t) of a seed and a target; a nonce solves it if $h(s,n) \le t$. The difficulty of a challenge, in bits, is

$$d(t) = 128 - \log_2(t+1),$$

so a uniformly random nonce solves it with probability 2^{-d} and solving takes 2^d hashes in expectation. A challenge of difficulty d has target $t(d) = \lfloor 2^{128-d} \rfloor - 1$, and $t = 2^{128} - 1$ for $d \le 0$. Seeds are random, so solutions can't be computed ahead of time, and a solution is only accepted for the challenge it was asked for.

Solvers search nonces from 0 in slices: 2^{18} nonces on a blocking thread at a time natively, 2^{14} between yields to the page in the browser. Dropping the search stops it after the current slice. One core does about $1.2 \cdot 10^7$ hashes per second in a release build and $2.4 \cdot 10^6$ in a debug build.

*ccertain@tutanota.com

Difficulty. Parameters $(d_0, \sigma, \delta, d_{\max})$: a base difficulty, a step, a decay per second and a maximum. Each subject (an address, a signer) has a difficulty $d(\tau)$ over time. A subject that hasn't been seen has $d = d_0$. Between events it decays linearly, never below the base,

$$d(\tau + \Delta) = \max(d_0,\ d(\tau) - \delta\Delta),$$

and at an event (a challenge issued to the subject) the challenge is issued at the current difficulty d, after which the difficulty becomes $\min(d+\sigma, d_{\max})$. Difficulty rises when a challenge is issued, not when it is solved, so asking for many challenges at once costs the same as asking one after another, and abandoned challenges count.

A subject issued challenges at most δ/σ times per second stays at the base. Above that rate its difficulty climbs by about σ per event, and since each bit doubles the work, a subject with c times more compute gets about $\log_2(c)/\sigma$ more events per burst, not c times more.

Replay. Nodes keep the times of past events in a database rather than the running difficulty. `Params::difficulty_after` computes the difficulty now from the ages $a_1 \geq a_2 \geq \cdots \geq a_k$ of past events, oldest first, by running the rule above:

$$\begin{aligned} e_1 &= \min(d_0 + \sigma, d_{\max}), \\ e_i &= \min\big(\max(d_0,\ e_{i-1} - \delta(a_{i-1} - a_i)) + \sigma,\ d_{\max}\big), \\ d_{\text{now}} &= \max(d_0,\ e_k - \delta a_k), \end{aligned}$$

and $d_{\text{now}} = d_0$ for no events. This equals the difficulty of an in-memory limiter that issued the same challenges at the same times (tested in `pow/tests/limiter.rs`). Events older than $(d_{\max} - d_0)/\delta$ have decayed to the base on their own and can be forgotten with little effect; each use below keeps events a little longer than that.

3 Connections

Handshake. A connection opens with an identity exchange: both peers send an 8 byte prefix and their 24 byte identity. The peer that opened the connection checks the identity against the one it dialed, if it knows it. Right after, before anything expensive (public key exchange, ML-KEM, signatures, database writes for keys), the accepting peer sends a challenge and the opening peer answers:

from	bytes	content
accepting	32	seed s
accepting	16	target t, big endian
opening	8	nonce n, big endian

The accepting peer closes the connection unless $h(s, n) \leq t$. The opening peer refuses challenges above 28 bits rather than hash for them. The whole handshake has to finish within 30 seconds.

Pricing. The subject is the address the connection came from: an IPv4 address, or the /64 network of an IPv6 address, since a single host is usually given a whole /64. IPv4 mapped IPv6 addresses count as the IPv4 address. Every inbound attempt is stored in the network database's `connection_attempts` collection as (address, time in milliseconds, random nonce). On each attempt the node deletes every attempt older than 500 seconds, stores the new one, and prices the challenge with the replay of the address's other attempts under

$$d_0 = 8, \quad \sigma = 1, \quad \delta = 1/30 \text{ per second}, \quad d_{\max} = 24.$$

An address reconnecting every 30 seconds, the reconnect interval of a node, stays near 2^8 hashes. A burst of 16 attempts reaches 2^{24} hashes, about 1.5 seconds in a release build, and the difficulty takes 480 seconds to fall back, within the 500 second window. A connection with no known address (a test stream, a local pipe) is asked the base difficulty and not recorded.

Remark 1. *Behind a reverse proxy every peer has the proxy's address. Nodes accept TCP and WebSocket connections directly for now. Two simultaneous attempts from one address each see the other, which errs on the strict side.*

4 Messages

After the handshake every message a peer sends is charged against a budget of 5000 messages that refills at 500 messages per second, up to the 5000. A message arriving with the budget below one closes the connection with `CloseReason::RateLimited`. The budget is per connection; a disconnected peer has to pay the connection proof of work to come back. Following a namespace takes a few messages, so a node syncing a thousand namespaces at once fits in a burst. The budget counts messages, not bytes; messages are at most 10 MiB.

5 Ratings

Every node records, for each peer identity it has been connected to, when it first saw the peer and how long it has been connected to it in total, in the network database's `peer_uptimes` collection. A peer's time starts when it authenticates and stops when its last connection closes; the time of connected peers is saved every 5 minutes so a crash loses at most that much.

A peer known for K days and connected for C days in total has

$$\text{rating} = \begin{cases} \text{none} & C < 1, \\ C + K/7 & \text{otherwise}, \end{cases}$$

and is *highly rated* at a rating of at least 2, e.g. connected for a day and known for a week, or connected for two days. Connection counts for more than acquaintance. These constants are placeholders to tune.

6 Namespaces

A namespace is identified by its signer and a subject. Gossiping a namespace and creating one look the same to the receiving node, so namespaces are paid for once, at creation, and the proof travels with them.

6.1 Challenges

Issuing. A node issues at most one challenge per namespace. Its difficulty is the replay of the challenges the node issued lately for the same signer, under

$$d_0 = 8, \quad \sigma = 1, \quad \delta = 1/3600 \text{ per second}, \quad d_{\max} = 26,$$

one bit more per namespace of the signer, one bit less every hour. A signer making a namespace an hour stays near the base; 18 in an hour reach 2^{26} hashes per challenge, about 6 seconds each in a release build. Issued challenges are stored in the `namespace_challenges` collection as (signer, subject, time, seed) and dropped after 24 hours, longer than the 18 hours the difficulty takes to fall from the maximum. Challenges are issued one at a time, so each sees the ones before it.

Signing. A challenge for namespace (signer, subject) issued by node I is signed by I's identity over

$$H(D_{ns} \,\|\, I \,\|\, \text{signer} \,\|\, \text{subject} \,\|\, s \,\|\, \text{be}_{128}(t)), \qquad D_{ns} = \texttt{"ficcus namespace challenge v1"}.$$

The signed challenge is $(I, (\text{signer}, \text{subject}), s, t, \text{signature})$.

6.2 Creation

Let C be the creator, the identity that signs the namespace. Creating a namespace is local and works offline. When C first publishes it:

1. C sends `RequestNamespaceChallenges`(namespace) to every node it is connected to. If it is connected to none, this happens when it next connects to a node, before the namespace is announced to it.

2. A node N answers only requests from the namespace's signer. It issues its own challenge, and sends `RequestNamespaceChallenge`(namespace) to every peer it is connected to other than C, waiting at most 10 seconds for each.

3. A peer P answers with `NamespaceChallengeOffer`: its signed challenge, or nothing if it has issued one for this namespace already.

4. N answers C with `NamespaceChallenges`: its own challenge and the offered ones, leaving out those already issued.

5. C keeps one challenge per issuer, stores them all in its `namespace_certificates` collection before solving any, then solves them one by one, storing each solution as it is found. A challenge above 28 bits is left unsolved. The stored challenges and solutions are the namespace's *certificate*. A certificate that is already stored is not requested again; an interrupted run resumes solving.

6. C announces the namespace.

Challenges are only collected from the nodes C is connected to and their direct peers.

6.3 Acceptance

A node R that hears of a namespace it doesn't have, from a peer S, before taking it in:

1. asks S for the certificate with a `RequestNamespaceCertificate`. The answer, a `NamespaceCertificateMaybe`, holds the solved challenges S has for the namespace, or nothing;
2. looks at the first 256 challenges, and keeps those for this namespace that are solved, fetching the public keys of issuers it doesn't know from S (an identity is a hash of its public key, so S can't substitute one), and whose signatures check out;
3. accepts the namespace if the check below passes, stores the valid challenges as its own copy of the certificate, so it can pass them on, and syncs the namespace.

Check. Let V be the set of (issuer, seed) of the valid solved challenges, s_R the seed R issued for this namespace if it issued one, and $\mathcal{H}$ the set of peers R rates highly. The certificate is accepted iff

$$\begin{aligned} &V \neq \emptyset, \\ &s_R \text{ exists } \implies (R, s_R) \in V, \\ &\left|\{I : (I, \cdot) \in V\} \cap \mathcal{H}\right| \geq \min(10, |\mathcal{H}|). \end{aligned}$$

That is, a node that issued a challenge requires its own, and a node requires challenges from 10 distinct highly rated peers, or from all of them if it knows fewer than 10. A node that knows no highly rated peer, e.g. in a network less than a day old, accepts any certificate with a valid solved challenge. A certificate stored once is not checked again, so a sync that was cut off resumes.

A rejected namespace is asked for again later, from every connected peer.

Remark 2. *A node far from where the namespace was made only accepts it if its highly rated peers are among the nodes that gave challenges. The cost to the creator grows with the number of issuers: every node the creator asks and every peer of those nodes adds a challenge, each up to 2^{26} hashes. Payloads in a namespace a node has are not priced; they are limited by the namespace's size limits and the message budget.*

7 Testing

Unit and in-process tests run with `cargo test -p ficcus -p ficcus-pow`: the replay against the limiter, address keys and attempt pruning, the message budget, a peer flooding messages, ratings, the acceptance check, issuing once per namespace, a namespace travelling along a chain of nodes with its certificate, and a node turning down a namespace without challenges from its highly rated peers.

Swarm. The `ficcus-swarm` crate runs nodes as separate processes. Each node is a `swarm peer` process with an in-memory state that never dials the seed nodes, listening on its own loopback address 127.77.x.y and connecting out from it, so nodes look like different hosts to each other (Linux routes all of 127.0.0.0/8 to the loopback interface). A node prints events to stdout as JSON lines (listening, dialed with the difficulty solved and the time taken, accepted, rejected) and takes commands on stdin (dial a target a number of times, one after another or all at once); it exits when stdin closes. `cargo test -p ficcus-swarm` checks that a mesh of nodes connects at the base difficulty, that a node reconnecting back to back pays more while another address doesn't, that a flood of simultaneous connections is priced one step per connection, that both ends agree on the difficulty, and that nodes keep working through reconnect storms. `swarm run --peers 6 --spammers 1` runs a scenario by hand and prints the difficulty each node paid.

8 Constants

constant	value	where
connection $(d_0, \sigma, \delta, d_{\max})$	$(8, 1, 1/30\,\mathrm{s}^{-1}, 24)$	`connection_pow_params`
attempt window	500 s	`ATTEMPT_TTL`
highest connection difficulty solved	28	`MAX_SOLVED_DIFFICULTY`
handshake timeout	30 s	`HANDSHAKE_TIMEOUT_MS`
message burst, rate	5000, $500\,\mathrm{s}^{-1}$	`MESSAGE_BURST`, `MESSAGES_PER_SEC`
connection time before a rating	24 h	`MIN_RATED_CONNECTION`
high rating	2	`HIGH_RATING`
namespace $(d_0, \sigma, \delta, d_{\max})$	$(8, 1, 1/3600\,\mathrm{s}^{-1}, 26)$	`namespace_pow_params`
issued challenge memory	24 h	`ISSUED_TTL`
highest namespace difficulty solved	28	`MAX_SOLVED_NAMESPACE_DIFFICULTY`
highly rated challenges required	10	`REQUIRED_RATED_CHALLENGES`
challenges looked at per certificate	256	`MAX_CERTIFICATE_PROOFS`
challenge fan out timeout	10 s	`FANOUT_TIMEOUT`