SIMON: Simple Interoperable Messages Over Network

The SIMON Protocol maintainers

Draft, 27 October 2026. Describes SIMON 1 (wire `v:1`), as specified in `spec/protocol.md` [1]. Where this paper and the specification differ, the specification governs.

Abstract

Software agents that act for different people and organizations need to exchange messages without sharing an operator. SIMON (Simple Interoperable Messages Over Network) is a protocol for this. An agent has an address `name@domain`; the domain names its server in DNS; servers carry messages for one another by store and forward, so neither agent need be online when the other writes. A message is one canonical JSON object whose identifier is the SHA-256 hash of its content and which carries its author's signature, checked by every receiving server against keys that the domain's server publishes. A thread is the chain of replies. An agent speaks for no one or for exactly one principal, and the message itself carries whatever evidence supports that principal. We state the security properties the design provides, the adversaries each holds against, and what lies outside them, including confidentiality, which version 1 does not provide.

1. The problem

Agents increasingly act for the people and organizations that run them: they schedule, negotiate, buy and answer. Two such agents rarely share an operator. An assistant run by one company for one person must reach an assistant run by another company for someone else, and the exchange must work as correspondence works. One side writes when it is ready, the other reads when it is able, and afterwards each can show what the other said.

We take the requirements to be these, in order of priority, in the manner of the goals that shaped the Internet protocols [2]:

1. **Stable addresses.** An agent is reached at a name that outlives its process, its machine and its keys, and that its correspondents can keep.
2. **No common operator.** Anyone may run a server for addresses at a domain they control. No registry, relay or directory stands between two servers. Trust in DNS and the web's certificate authorities is assumed, as it is for mail and the web.
3. **Asynchrony.** Neither agent need be online when the other writes. Delivery, meaning acceptance by each recipient's server, completes or fails within a stated time.
4. **Attribution.** Any reader of a message, including a third party who obtains a copy later, can check which key signed exactly these bytes, and can tell what evidence, if any, supports the statement that the agent speaks for a particular person or organization.
5. **Indifference to content.** A server routes, stores, delivers and verifies a message without understanding it.

Internet mail meets the first three and has met them at the scale of the Internet for four decades [3]. It does not meet the fourth. A DKIM signature is optional, attests to a domain rather than an address, need not be checked by any receiving server, and breaks when a message is modified in transit [4]. S/MIME and OpenPGP sign per author [37, 38], but are optional, are verified by no server, and distribute keys outside the transport. Protocols built for agents meet the fourth in part and the third not at all. In A2A one agent calls another's HTTPS endpoint, so the callee must be reachable when called [5]; MCP connects a model to tools and data rather than to another party [6].

SIMON keeps the architecture of mail and adds one thing to it: every message carries a signature by a key bound to its author's address, and every receiving server checks that signature before it accepts the message. The gain over a domain signature is not independence from the domain, whose server still says which keys belong to which address (section 4, P1). It is that the signature is mandatory, checked at every hop, made per address by a key the agent may hold itself, and carried with the message as transferable proof wherever it goes; and that a principal's own signature, where there is one, can be checked without trusting any server. The rest of the design follows from keeping that addition, and little else, in the core. A feature enters the core only when it has existed for decades in some form and adding it later would force every server to change. Everything else is a header, a block type or a relation name in a registry that only readers interpret. Nothing in the core concerns artificial intelligence. An agent is a sender.

2. Design

2.1 Notation

`C(x)` is the canonical JSON encoding of `x` (section 2.4). `H` is SHA-256, written as 64 lower-case hexadecimal digits where it names something. `m\s` is a message `m` without its `signature` member. `Sign_k(c, b)` is a signature by key `k` over the ASCII context string `c`, one newline, and the bytes `b`. Each kind of signed object has its own context (`SIMON-MESSAGE`, `SIMON-ENVELOPE`, `SIMON-CLAIMS`, `SIMON-AGENT`, `SIMON-PAIR` and others). No context followed by a newline is a prefix of another, and no variable line can contain a newline, so no signature made for one purpose verifies for another [1, Appendix A].

2.2 Addresses

An address is `local@domain`, the only thing the protocol routes to. The domain is an ASCII DNS name; an internationalized domain is written as its A-labels. In version 1 messages the local part is ASCII, with the atom characters of Internet mail and dots between atoms, and inside a message an address is written in lower case, so that one message has one form and one identifier.

The rule that binds every version is wider: a local part is 1 to 64 bytes of UTF-8 with no `@`, space, ASCII capital or control character. A later version may admit non-ASCII local parts without any change to routing. Unicode normalization is deliberately not part of this rule, because whether a string is normalized depends on the Unicode tables a runtime carries, and two routers must never disagree about an address.

2.3 Servers and keys

A domain names its server in one of two ways, tried in this order:

```
_simon.<domain>  TXT  "v=simon1; server=https://<host>"
GET https://<domain>/.well-known/simon            {"server":"https://<host>"}
```

Resolution ends in one of three outcomes: a server, no server, or try again. A temporary failure of the TXT lookup ends resolution as *try again* without consulting the well-known answer, and *try again* is remembered only briefly and never read or cached as *no server*; otherwise whoever can drop DNS answers could route a domain to whoever runs its website. The outcome records which method named the server, because only the TXT record grants authority over the domain's email addresses (section 3.3). A domain names exactly one server, with no priority list and no backup.

A server publishes two kinds of document, without authentication, cacheable for at most ten minutes. The *server document* lists the server's keys, the algorithms it verifies, and its limits. The *agent document* lists one agent's keys and is signed by a server key. Each key entry carries `added` and `revoked`; a key is *valid at* a moment `t` when `added` is at most 300 seconds after `t` and `revoked` is empty or later than `t`. History grows only forward: a revocation takes effect when first published, and is never backdated, scheduled ahead or withdrawn. Documents are fetched without authentication on purpose, since requiring a signed request to fetch the key that verifies signed requests is a deadlock. A reader obtains an agent's document only from the server the agent's domain resolves to. The document says that an agent exists and what its keys are, and nothing about whom it speaks for.

A key is a token, an algorithm name and the raw public key: `ed25519:11qYAYKxCrfVS_7TyWQHOg7hcvPapiMlrwIaaPcHURo`. Every implementation implements `ed25519` [7], verified strictly (the cofactorless equation, `S` below the group order, neither the key nor `R` of small order) so that every verifier accepts exactly the same signatures [8]. An implementation may implement `ml-dsa-65` [9], and then advertises it. A signature does not name its algorithm: a verifier tries each of the signer's valid keys whose algorithm it implements and skips the rest. A signer whose every valid key is of an algorithm the verifier lacks cannot be verified by it, and the verifier fails explicitly rather than accept weaker evidence. Servers sign what they send one another with `ed25519` keys.

2.4 The message

The message is the protocol's only object:

```
{"from":"ada@example.com","headers":{"for":"mailto:alice@example.org","subject":"Thursday"},
 "inReplyTo":null,"parts":[{"data":"Does **3pm** work?","type":"text/markdown; variant=GFM"}],
 "sentAt":"2026-10-14T21:00:00Z","to":["bob@example.net"],"v":1,"signature":"<base64url>"}
```

Version 1 defines exactly eight members, all required: `v`, `from`, `to`, `inReplyTo`, `parts`, `headers`, `sentAt` and `signature`. A message is transmitted and stored as canonical JSON: RFC 8785 [10] restricted to integers within ±(2^53 − 1), so that nothing needs floating point, with keys sorted by UTF-16 code units, nesting at most 64 deep, no whitespace, and one escape for each character that needs one. (The example is broken into lines for the page.) A server refuses a message that is not already canonical and returns its canonical form, so a difference in encoding is caught at the first hop. Then

```
id(m) = H( C(m\s) )
s     = Sign_k( "SIMON-MESSAGE", C(m\s) )
```

where `k` is a key of `from` valid at `sentAt`. The identifier commits to every member but the signature, `inReplyTo` among them, so a reply names the exact content it answers.

`parts` is an ordered list of 0 to 32 *blocks*, each typed by a media type, carrying data inline or naming bytes by `sha256` and `size`. No block is required. `headers` maps at most 32 names to strings. A reader ignores a header it does not know unless the message lists it in `critical`, in which case the reader presents the message as not understood. The protocol names eight headers (`for`, `pairing`, `succession`, `critical`, `rel`, `feed`, `audience`, `listed`) and every server checks their grammar. Any other name without a dot belongs to the registry; anyone else names a header under a domain it controls, in reversed label order, as `com.example.approve`.

Five things are frozen for every version:

1. a message is a JSON object, transmitted as canonical JSON;
2. it has `v`, `from`, `to`, `sentAt` and `signature`;
3. its identifier is `id(m)` as above;
4. its signature is `s` as above, by a key of the author valid at `sentAt`;
5. a server finds another by the mechanisms of section 2.3, or a successor they advertise, and delivers by an HTTPS `POST` of an envelope to `/simon/deliver`.

A server routes a message of a version it does not know, applying only the checks that do not depend on the version. An agent's own server refuses to accept from its agent a version it does not implement, and a reader presents such a message as not understood. Carriers pass along what they do not understand; this is what let SMTP and HTTP last, and what QUIC states as its version-independent invariants [11].

2.5 Delivery

An agent submits a signed message to its own server through the client interface [12], which checks the version, the signature, the agent's principal (section 3) and a `sentAt` at most 24 hours old. The author's server resolves each recipient's domain and posts one *envelope* to each server that serves some of the recipients:

```
agent A              server of a.example                  server of b.example          agent B
signs m --submit-->  signs envelope E    --POST deliver-->  resolves a.example;           reads its
                     (m, server,                            checks E with a key of       inbox when
                      recipients,                           a.example's server;          it is able
                      claims, hints)                        checks m with A's key
                     retries until                          valid at m.sentAt;
                     the deadline                           answers 202; stores m
                                                            once per (id, recipient)
```

The envelope holds the message; `server`, the origin of the receiving server; `recipients`, the addresses in `to` that server serves; optional *claims* about the author (section 3.3); and optional per-recipient *hints*. The sending server signs it under `SIMON-ENVELOPE`. Because the receiving server and recipients are signed, an envelope cannot be replayed to another server or stretched to other recipients. The receiver checks the envelope with a key of the server that the `from` domain resolves to, so the carrier of a message is always the server of its author's domain. Relays, forwarders, backup servers and list exploders cannot carry a message on another domain's behalf.

A receiving server checks, in a recommended order: the body's size, the envelope's shape, the message's canonical form and version rules, that the envelope is addressed to itself, the resolution of the author's domain, the carrier's signature, that it serves the recipients, whether the message is a repeat, its freshness, and the author's signature. Four dependencies are requirements: the body is bounded before it is parsed; the carrier's signature is verified before any work that depends on a recipient or a third party; a repeat is recognized before freshness is judged; and nothing is changed or accepted until every check passes. A valid message is answered `202` whether or not any recipient will see it: whether a recipient sees it is the receiving server's policy, and the answer does not say.

Calls between servers other than deliveries and signals, and an agent's calls to its own server, are signed requests. The signature covers a fixed string of eight lines: a context, the method, the origin of the server asked, the acting identity, the exact request target, a timestamp, a nonce, and the hash of the body. HTTP Message Signatures [39] were not used because their choice of covered components is where independent implementations diverge; a fixed string has byte-exact test vectors and can be built in a shell.

Delivery between servers is at least once, and into each recipient's inbox at most once per (id, recipient). Three constants set the windows: a send window of 24 hours, a delivery horizon of 144 hours after `sentAt`, and a clock skew of 300 seconds. A receiver accepts a message whose `sentAt` is at most 144 hours old and at most five minutes ahead, and recognizes repeats for 168 hours. The author's server never delivers before `sentAt`, retries `429` and `5xx` answers with backoff, does not retry other refusals (except two narrow cases that arise while a domain moves between servers), and stops at the earlier of five days after accepting the message and 143 hours after `sentAt`, so that no delivery is stale on arrival. Messages carry no order; agents recognize a message already seen by its identifier.

A file is a part named by `sha256` and `size`. The author's server serves it only from bytes held for the author of a message that names it, and only to those who received or may read such a message; knowing a hash is never permission. A recipient's server should copy files when a message arrives, not when it is read, since a fetch on reading tells the carrier when the message was read. Anything larger than the receiver's `partBytes` travels as a link. A separate call carries *signals* (typing, collected, read), sent only when an agent asks, droppable by any receiver, and never stored.

2.6 Threads

`inReplyTo` is the single pointer between messages: `null`, or the identifier of the message this one answers. A `rel` header says what kind of pointer it is, such as a reaction, an edit or a retraction, with names kept in the registry. Two rules bind every reader. A relation a reader does not know is shown as a reply. A relation that changes or withdraws a message, such as `edit` or `retract`, takes effect only when both messages have the same `from`; from anyone else it is an ordinary reply.

A conversation is a root message and every message whose chain of `inReplyTo` leads back to it, through replies and through relations that stay in their conversation; a forward, a quote or a comment points at a message without joining it [13]. Its identifier is the root's. There is no roster. Each message's `to` is its author's choice, a reply that reaches more or fewer people forks the conversation, as in mail and text messaging, and servers keep no conversation state. Since no message changes who is in a conversation, there is no membership to forge and nothing for servers to reconcile.

2.7 Extension

A reader ignores what it does not know, so the protocol grows without coordination: by headers (with `critical` where a reader must understand them), block types, relation names, signal types, claim types, signature algorithms as new key-token names, resolution methods advertised by the existing ones, and message versions routed on the invariants. A name enters the registry once two independent implementations use it. Optional *profiles* build on the core without changing it: a reader profile, conversations, publishing (messages with an empty `to` and an `audience` header, served by the author's server), and discovery. The core depends on none of them.

3. Principals

3.1 Solo and paired agents

An agent is one of two kinds, fixed for the life of its address. A *solo* agent speaks for no one, and its messages carry no `for` header. A *paired* agent speaks for exactly one *principal*, a person or an organization, named by one identifier that every message it sends carries as `for`: `mailto:` and an email address, `tel:` and an E.164 number, or `simon-key:` and a key token, for a principal who holds a key.

One agent and one identifier make one address. The same assistant working for someone's work email and for their personal email is two agents, with two addresses, two sets of keys and two inboxes. One inbox never holds two principals' messages, so no server or agent has to work out whom a message is for. A system that serves many people from one place must make that inference on every message, and when it errs it acts on one principal's authority for another's request: the confused deputy [14]. Pairing removes the inference. The address is the principal, so routing, not application care, keeps one principal's messages from another's. A retired address is never paired again.

3.2 What verifying a message proves

Verifying a message proves its author in a precise and limited sense: a key, which the server named by the author's domain lists for that address as valid at `sentAt`, signed exactly these bytes. It does not prove who holds the key; a hosted agent may have a signing service hold it, as mail providers hold DKIM keys. It does not prove that a human wrote the message. And by itself it proves nothing about the principal: a `for` header alone is the agent's word, as the From line of an email is.

3.3 Evidence for a principal

A reader treats a principal as verified only on one of two kinds of evidence, both carried with the message.

A key principal's pairing. The principal signs `SIMON-PAIR`, the agent's address, the identifier and a time `until`, and every message the agent sends carries `until` and that signature as its `pairing` header. Any reader can check it with the key named in `for`, trusting no server. A pairing counts before its `until` and at most 366 days before it, so no signature binds an address to a principal for more than about a year; the principal renews by signing a later `until`. A reader judges the pairing at `sentAt`, which the agent's own server also checked against its own clock, so an honest host lets no message be backdated past a lapse by more than its 24-hour send window. The principal ends a pairing by signing `SIMON-UNPAIR`, and the agent's server must then retire the address, whatever the agent's keys say. No channel carries an unpairing to readers: a reader learns of one only from retirement evidence the host may choose to serve, or from the principal. A principal moves to a new key by a *succession*: old and new keys both sign `SIMON-SUCCEED` with both identifiers, messages carry the chain, and a reader that knew any key in it may treat the new key as the same principal. Two successions from one key are a fork, and a reader that holds both links neither successor to that key. A `for` that changes to a key with no succession from the one before is a new principal, and a reader should say so. An unpairing signed by any key in the chain ends the agent.

A pairing proves that the key's holder consented to this address speaking for that key. It binds the address, not the agent's current key, so it survives rotation of the agent's keys, and survives their compromise until the principal unpairs, the pairing lapses, the host's recovery removes the compromised keys, or the address retires. It does not say who holds the principal's key, and it carries no scope; what an agent may do for its principal is left to applications.

A carrier's claim. An email or telephone principal signs nothing. The carrier, having verified the identifier under documented policy, typically by sending a code to it, attaches an `acts-for` claim and signs the claims for this message: `Sign("SIMON-CLAIMS", C({claims, message: id(m)}))`. The signed claims travel with the message into its recipients' inboxes, so a later reader can check them wherever the message goes, for as long as the carrier's key history can still be obtained. A claim is *authoritative* when its identifier is an email address whose domain names the carrier by its `_simon` TXT record: the domain's DNS speaks for the domain's mail, as with DKIM. The well-known answer grants no such authority, because whoever runs a domain's website need not run its mail. Every other claim, every claim about a telephone number among them, is the carrier's word about someone else's identifier; a receiving server presents it as verified only from carriers it chose to trust. A reader that relies on a claim repeats the judgement itself.

3.4 Identity modes

A message is therefore *identified* (`for` names an email address or telephone number), *pseudonymous* (`for` names a key), or *unattributed* (no `for`), and a reader can tell which from the message alone. The protocol never makes an agent pair, and a server never attaches a claim naming an identifier its agent did not choose to show. Anonymity is toward other agents, never toward the agent's own server.

4. Threat model and security properties

4.1 Adversaries and assumptions

The parties are agents, their principals, the servers that host them, and readers: any party that interprets a message, including a recipient's server and anyone who later holds a copy. We consider seven adversaries.

- **N**, the network: any party on the path between servers, or between a reader and a server.
- **D**, a DNS adversary: one who can make a server or reader believe a false answer about a domain, by position on its network, a poisoned resolver, or a hijacked route to the domain's name servers.
- **S**, a malicious, compromised or coerced server: the author's, a recipient's, or a third.
- **A**, a malicious agent, including one on its victim's own server, and a sender who writes to probe rather than to communicate.
- **K**, a thief of an agent's, a server's or a principal's key.
- **R**, the next registrant of a domain, or whoever later controls its DNS.
- **I**, an author of hostile content meant to steer the agent that reads it.

We assume that SHA-256 is collision and second-preimage resistant; that Ed25519 with strict verification and ML-DSA-65 are existentially unforgeable under chosen-message attack; that TLS 1.3, authenticated against the peer's origin by the web's certificate authorities, protects each connection; and that honest clocks agree within 300 seconds.

A reader's client verifies each message itself, resolving the author's domain and reading its documents, rather than trusting its own server's checks or keys that arrive with the message [12]. The properties below are stated for such a reader. An agent whose client does not verify for itself trusts its own server for everything it is shown.

4.2 Properties

Each property is stated with whom it holds against, what it rests on, and where it stops. None has been proved in a formal model (section 4.5).

P1. Authorship. If a reader accepts `m` as verified, then a key `k`, listed for `from(m)` as valid at `sentAt(m)` in an agent document signed by the server that `from(m)`'s domain resolved to, signed exactly `C(m\s)` under `SIMON-MESSAGE`, a context that signs nothing else. *Holds against* N; against recipient servers and later holders, who cannot change a byte without breaking the signature and the identifier; against other agents, including those on the author's server, who do not hold `k`; and against every server but the author's, since keys are read only from the server the author's domain names. Strict verification means no two honest verifiers disagree about any `ed25519` signature [8]. *Rests on* the author's server's honesty about its agents' key histories, and on the DNS answer. *Stops at* the author's server, which can list a key of its choosing with a false `added` and sign with it; at D (section 4.3); and at K, since a stolen key signs anything until revoked and can date messages before its revocation. Live delivery bounds the last, because a receiver refuses a message more than 144 hours old. Version 1 keeps no log of key histories, so only a reader that cached an earlier document can catch a server that rewrites one. A signature is transferable proof to anyone who can still obtain the authenticated key history; once a server stops listing a revoked key, a new reader cannot verify what it signed.

P2. Content identity. Any two parties holding the same message compute the same identifier, and a change to any member but the signature changes it. Because `inReplyTo` is signed content, a reply commits to the exact message it answers, and a conversation is linked by hashes of content. *Holds against* every adversary, given SHA-256, which keeps a 128-bit preimage margin against quantum search. Canonical form is unique and enforced at the first hop, so encodings do not drift between servers. *Stops at* what an identifier names: content, not a signing event. It does not distinguish two valid signatures over the same content, which only a holder of the author's key can make. A reply proves only that its author knew an identifier, not that the earlier message exists or was received. No reader can know that it holds all of a conversation.

P3. Carrier binding. Every delivery is signed by the server of the author's domain, over an envelope naming the receiving server and the recipients. An envelope cannot be replayed to another server or used for other recipients; a replay to the same server is a repeat for 168 hours, storing nothing, and stale after 144. Claims are signed over the claims and the message identifier, so they stay bound to that message. Signed requests bind the method, the origin asked, the acting identity, the exact request target, a timestamp within 300 seconds, a nonce and the hash of the body. *Holds against* N, A and every server but the carrier. Since only a domain's server can carry its messages, abuse is attributable to a domain and a server, and limits and refusals attach to an accountable party. *Stops at* D, since the carrier is whatever server the `from` domain resolves to, and at what an envelope says: which server carried a message, not that its agent wrote it, which is P1. By construction it excludes relays (section 6).

P4. Delivery. If some attempt before the deadline reaches the recipient's server, that server can resolve the author's domain and read its documents, and it admits the message, it enters each recipient's inbox once, with neither agent online. Otherwise the author's server stops at the deadline, and its agent sees, per destination server, whether delivery is pending, delivered or failed. *Holds against* N, which can delay but not alter a delivery, and transient failure, which retries absorb. *Stops at* the recipient's server, which may drop any message silently; at a domain whose single server is down past the deadline; and at order, which the protocol does not provide. A mistyped address receives no bounce.

P5. Silence toward senders. A receiving server answers every valid delivery `202`, whether a recipient accepted it, refused it or does not exist. It should also do no recipient-dependent work before answering, a recommendation, unlike the requirement of P8. A sender learns from delivery neither whether an address exists nor whether its holder refuses the sender. Signals likewise reveal nothing about recipients. *Holds against* A. *Stops at* agent documents, which are public and reveal that an address has an agent, though a server may decline to disclose agents and should limit lookups of absent addresses, refusing every address alike past the limit; at timing, where a server ignores the recommendation; and at file fetches, which reveal admission or reading unless a server fetches the files of every message it accepts soon after accepting it, as the reference server does.

P6. Principal evidence. A reader treats a principal as verified only with a pairing that verifies under the key in `for` and counts at `sentAt`, or with a carrier's claims signature that the reader judges authoritative or trusts. *Holds against* A naming any identifier in `for`; against S, which cannot forge a key principal's consent; against anyone pasting a message or inbox entry, since claims are bound to the message identifier and re-judged by the reader; and against old consent outliving its purpose, which the 366-day limit bounds. *Stops at* the following, by adversary. *S, the agent's host:* it can attach the agent's genuine pairing to messages it signs with keys of its own, because a pairing binds an address, not a key; and since no channel carries an unpairing to readers, a host that ignores an unpairing keeps the agent speaking for its principal, to every reader, until the pairing's `until`. This is left to hosts on purpose. A rule that hosts publish unpairings would bind only the hosts that honour them, a published unpairing can be tested against candidate principal keys, and an honest host's retirement already revokes the agent's keys. The cap on `until` bounds the exposure, as short certificate lifetimes do on the web, and a principal that doubts its host can sign short pairings and renew them. *R:* it can present an old pairing with keys of its own, to readers that never learned of an unpairing, until its `until`. *K, with a principal key:* it can sign a rival succession, which readers that see both moves treat as a fork, while a reader that sees only the thief's links the thief's key (moving an agent at its own server also takes the agent's key); and it can unpair the agents, so the safe end of a stolen principal key is the end of its agents. *K, with a carrier key:* claims carry no expiry, revocation or independent time of issue, so a carrier key valid at any moment of the delivery window can sign claims for that window later. *D:* authority rests on one TXT lookup, exposed unless validated by DNSSEC or from several vantage points, and judged when the judgement is made, not when the claim was issued. *A, on a shared host:* on a server hosting several domains, one domain's agents carry authoritative claims about another's addresses unless the reader requires, as DMARC does [15], that the identifier's domain be the agent's own. Nothing shows who holds a key, or whether a human is involved.

P7. Safety of versions, algorithms and relations. A new version number cannot step around a known version's rules: the agent's own server refuses a version it does not implement, and a reader presents one as not understood, saying nothing of its content or principal. A missing algorithm is an explicit failure, never a downgrade. An unknown header listed in `critical` stops a reader from acting. An edit or retraction by anyone but the author is shown as a reply and never applied, which is the lesson of forged Usenet cancels. *Holds against* A and S writing versions, headers or relations to mislead a reader. *Rests on* readers following the reader rules, which have test vectors but no conformance suite yet (section 7). *Stops at* algorithm strength: an agent holding keys of two algorithms is as strong as the weaker, so retiring an algorithm means revoking its keys.

P8. Bounded work. Every input is bounded: sizes of messages, requests, signals and documents, counts of keys and parts, file aggregates, nesting depth and time windows. No recipient-dependent work precedes the carrier's signature, so one unsigned request cannot make a server contact a hundred domains. A server follows no redirect and connects to no private or link-local address unless configured to. *Holds against* A and N as sources of load on the receiver, and against use of a receiver to reach a hundred recipient domains. *Stops at* the lookups that verifying a carrier requires. An unsigned envelope naming any `from` domain makes the receiver resolve that domain and read the server document of whatever host it names before any signature can be checked; a signed read from a foreign agent adds that agent's document. What one request causes is fixed: each answer is bounded in size, no redirect is followed, a refetch after a failed signature waits a minute per server, and no record names another to look up, as SPF's `include` does, which is why SPF needed a cap of ten lookups [18] and SIMON does not. An unsigned request is therefore a reflector toward a host its author chose, sending a small fixed number of requests, not an amplifier. How often a receiver does this for one client, network or domain is what each implementation must bound, with mechanism and numbers its own, so the rate bound is a requirement on implementations, not a property of the protocol. Nor does the protocol address unsolicited messages from senders whose marginal cost is near zero: it defines no postage, proof of work or global reputation.

P9. Content is data. Everything in a message is written by someone else. An agent must not treat any of it as authorization from the one it acts for; a client library labels third-party text and parts with who wrote them; an interactive block is drawn by the reader, and choosing an option sends a new signed message rather than calling a URL or running code; keys come only from resolution, never from a party a message names. *Holds against* I

insofar as implementations follow these rules. *Stops at* the model. Whether a model obeys a label is outside the protocol, which bounds what a message can cause structurally but does not prevent persuasion.

4.3 DNS

Every address meets its keys in DNS, whichever resolution method is tried first, since TLS authenticates the well-known host only after DNS has said where it is. Whoever can make a server believe a false answer can point a domain at a server of their own, which then publishes keys for its addresses. DKIM and web certificate issuance share this exposure [16]. The defences are layered. A server that validates DNSSEC treats a validation failure as *try again*, never as an answer, which for signed zones defeats attacks on the path and on resolvers. A server should resolve over an authenticated channel to resolvers it trusts. Confirming answers from several network locations defeats attacks local to one network; asking several resolvers from one host is weaker, though better than asking one. DNSSEC does not stop whoever controls a zone or its signing keys, and a dangling `_simon` alias delegates a domain to whoever can claim its target, so an operator removes it when retiring a service. A server should remember which of each domain's server keys it has seen sign, and hold deliveries when a newly resolved server publishes none of them or never signs with them. The test is possession: a remembered key that signs, not a listed public key, which anyone can copy. Control of a domain at its registrar is control of its addresses, and no protocol can tell it apart from ownership.

4.4 Out of scope

Version 1 provides no confidentiality from servers. The author's server sees everything its agents send, and each recipient's server sees what it carries, as mail servers do. Under end-to-end encryption servers would still see who wrote to whom, when and how much, and whoever runs an agent's model sees what the agent reads and writes, whatever any protocol does; servers and model operators may retain what they carry. Recipients can link the agents of one principal, which share its identifier, and addresses that share an agent key. DNS resolvers see which domains a server corresponds with.

It provides no deniability. Every message is transferable proof that its author's key signed it, for as long as that key's history can be obtained. This is deliberate: agents that commit for people need attributable messages, and a deniable signature kind would be an extension.

It does not make recovery safe, since recovery is the host's policy and whoever can recover an address controls its agent; isolate the pairs one operator's runtime serves; decide whom an agent may contact; or give rights to people a message mentions who are not party to it.

4.5 What has not been shown

These properties are argued informally from the specification's rules. No formal model of SIMON exists. Models in ProVerif or Tamarin of pairing, delivery, key continuity, recovery and grants are on the roadmap [17], and an external security review named them as what matters most before version 1 is declared stable. No independent cryptographic review has been made, and whether agents over-trust the principals that messages claim has not been studied. The properties are claims to be checked, not results.

5. Related work

SIMON keeps the model of Internet mail, borrows mechanisms from federated systems, and sits beside, not on top of, the protocols built for agents.

System	What SIMON shares	Where SIMON differs
SMTP with SPF, DKIM, DMARC, ARC [3, 4, 15, 18, 19]	addresses at domains; servers found by DNS; store and forward; a domain's DNS authorizing keys; alignment	every message signed by its author, not optionally by a domain, and checked by every receiving server; messages never modified in transit; no relays, so no ARC
S/MIME, OpenPGP [37, 38]	signatures by the author over the message itself	mandatory and checked by every receiving server; keys found through the author's domain, not distributed outside the transport
Matrix [20]	integer-only canonical JSON; server keys fetched unauthenticated; federation by domain	no rooms and no replicated state; Matrix events are signed by servers, SIMON messages by authors; no notaries
ActivityPub [21]	server-to-server delivery over HTTPS, signed in common deployments; addressing public posts	the agent, not its server, holds the signing key; key documents always unauthenticated; no vocabulary of activities
AT Protocol [22]	name resolution by a TXT record under an underscore label, then an HTTPS well-known path	no public repositories or relays; messages go to named recipients; identity is an address and its keys, not a DID
Nostr [23]	signed JSON events identified by a SHA-256 hash; `name@domain` through a well-known path (NIP-05)	a domain names one server that carries and authenticates delivery, retried to a deadline, rather than relays
XMPP [24, 25]	`local@domain`; federation between domain servers; servers authenticated by domain	JSON over HTTPS; each delivery signed, rather than a server authenticated per connection by certificate or dialback; every message signed by its author
WebFinger [26]	looking up `name@domain` over HTTPS	one well-known answer per domain names the server; keys come from that server's agent document
DIDComm v2.1 [27]	signed messages between agents; delivery while the recipient is offline	no DIDs; the carrier is the author's domain's server, not a chosen mediator; no encryption in version 1
A2A v1.0 [5]	independently operated agents; discovery at a well-known HTTPS path	A2A asks a reachable agent to run a task; SIMON stores and forwards signed messages between addresses
MCP [6]	the agent ecosystem	MCP connects a model to tools and data; an MCP server is one way for an agent to use SIMON
WIMSE, SPIFFE [28, 29]	cryptographic identity for software that acts	workload identity for service calls within and across trust domains; SIMON names agents at public addresses and carries principal evidence per message
MLS, MIMI [30, 31]	interoperable messaging; semantics of replies, reactions, edits, retractions	SIMON 1 is unencrypted; groups are a `to` list; MLS is the planned basis for encrypted groups
KEYTRANS, CONIKS [32, 33]	auditable key histories	SIMON 1 has no transparency log

Four points need more than a row. First, mail lets intermediaries modify messages, so DKIM signatures break on mailing lists and ARC [19] adds intermediaries' signatures to carry results across them. SIMON makes the opposite trade: messages are never modified, so the author's signature always survives, and no intermediary can carry a message for another domain. Second, Matrix [20] replicates each room's state across servers and resolves conflicts by an algorithm revised more than once; SIMON keeps no shared state and so has nothing to resolve. Third, Nostr [23] makes the key the identity and treats relays as untrusted stores; SIMON makes the address the identity, so a correspondent has a stable name and delivery has a responsible party. Fourth, A2A [5] and MCP [6], both under the Linux Foundation's Agentic AI Foundation, are complements rather than alternatives: A2A's unit is a task performed by a reachable agent, SIMON's a signed message to an address that may be offline, and an agreement reached in a task can be recorded as a message. SIMON follows MIMI's content semantics [31] for replies, reactions, edits and retractions where they fit, so a bridge between the two maps them directly. The AT Protocol is now before an IETF working group [34].

6. Limitations and future work

No end-to-end encryption. Servers read what they carry. The format anticipates encryption: an agent-document member `encryptionKeys` and a block type `application/simon-sealed` are reserved, servers carry a sealed message like any other, and a test vector shows one passing every server check. That shows carriage only. The planned profile uses HPKE [35] for one recipient, MLS [30] for groups, and hybrid post-quantum key agreement, because ciphertext kept for years is exposed to later decryption. Groups need an order for their changes and a way to distribute keys, which may need services the core lacks. The signature stays outside the ciphertext so servers still verify the author. Meanwhile a sender with confidential material sends a link to storage it controls.

No key transparency. Nothing but the author's server backs its key histories (P1). A portable key log was left out of the core, since delivery does not need it and every implementer would carry it; it can be added as a document member, and KEYTRANS [32] is the natural basis. An open proposal would give claims their own time of issue.

Live sessions. Store and forward is slow for negotiation in real time. The direction is an opt-in upgrade in which a signed message offers a live channel for one conversation and a signed reply accepts it, as SIP's offer and answer and XMPP's Jingle do. The message network stays the source of truth: anything that matters, such as an agreement, is sent as a message afterwards. This waits for cases where a round trip of seconds is too slow.

Privacy and erasure. A `mailto:` or `tel:` identifier in `for` is signed into every message the agent sends and cannot be removed from a copy without breaking its signature, its identifier and every reply after it. The remedy within version 1 is to name principals by keys, held for them by their host if need be, and to carry email addresses and telephone numbers as claims, which travel beside the message and are not signed into it. Erasing a person then erases the claims a host keeps, and nothing signed names them. Copies on other servers are theirs, and no erasure request reaches them.

Topology. A domain names one server with no backup, and an outage longer than the deadline loses what is undelivered. SVCB records [36], advertised by a successor resolution method, would add priority and failover. With no relays there are no list exploders, and each receiving server bounds how many addresses a message's `to` may hold, at least 100; larger audiences are published feeds, and large private groups await MLS.

Verification over time. Whether an old message still verifies depends on its author's server keeping the key history. The protocol promises the standard's longevity, not each message's, and defines no archival evidence format.

7. Status

SIMON 1 is published; its messages carry `"v":1`. Until it is declared stable any rule may change, except the five invariants listed in section 2.4, which change only for an overwhelming reason.

There is one implementation, the reference, as a basic server and an application template built on it, in the 0.3 series of packages. A Python client reproduces every test vector, but it is a port maintained beside the JavaScript client, not an independent implementation. The vectors are generated by the reference library, which the server and the conformance suite also import, so a mistake in that library would appear in all three. The suite tests servers configured for it and cannot test the transport rules; reader rules have vectors but no conformance suite; and passing tests do not establish complete coverage or live interoperability. Trials across the public Internet ran the reference against itself on tunnel names without DNS records.

SIMON 1 will be declared stable when two independent implementations reproduce every test vector and pass the conformance suite against each other. Independent means written from the specification alone, by people outside the team that wrote the reference, in another language, with its own cryptographic bindings and a public log of every question the text did not answer, and federating with the reference in both directions on real domains. That criterion has not been met. No benchmarks or adoption figures are reported, because none have been measured in a form that can be reproduced.

Dedication

SIMON is inspired by and dedicated to Simon Baumer, who believed in an open and decentralized internet.

References

1. The SIMON Protocol: SIMON 1 (wire `v:1`). `spec/protocol.md`. https://github.com/agnt-ai/simon-protocol/blob/main/spec/protocol.md
2. D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. ACM SIGCOMM, 1988. https://groups.csail.mit.edu/ana/Publications/PubPDFs/The%20design%20philosophy%20of%20the%20DARPA%20internet%20protocols.pdf
3. J. Klensin. Simple Mail Transfer Protocol. RFC 5321, 2008. https://www.rfc-editor.org/rfc/rfc5321
4. D. Crocker, T. Hansen, M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, 2011. https://www.rfc-editor.org/rfc/rfc6376
5. Agent2Agent (A2A) Protocol Specification, version 1.0, March 2026. https://a2a-protocol.org/latest/specification/ (summary: https://aaif.io/blog/a2a-v1-0-a-builder-s-guide-part-1-discovery-tasks-and-clients)
6. Model Context Protocol Specification, revision 2026-07-28. https://modelcontextprotocol.io/specification/2026-07-28
7. S. Josefsson, I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, 2017. https://www.rfc-editor.org/rfc/rfc8032
8. K. Chalkias, F. Garillot, V. Nikolaenko. Taming the Many EdDSAs. Security Standardisation Research, 2020. https://eprint.iacr.org/2020/1244
9. NIST. Module-Lattice-Based Digital Signature Standard. FIPS 204, 2024. https://csrc.nist.gov/pubs/fips/204/final
10. A. Rundgren, B. Jordan, S. Erdtman. JSON Canonicalization Scheme (JCS). RFC 8785, 2020. https://www.rfc-editor.org/rfc/rfc8785
11. M. Thomson. Version-Independent Properties of QUIC. RFC 8999, 2021. https://www.rfc-editor.org/rfc/rfc8999
12. The SIMON client API. `spec/client.md`. https://github.com/agnt-ai/simon-protocol/blob/main/spec/client.md
13. SIMON conversations profile. `spec/profiles/conversations.md`. https://github.com/agnt-ai/simon-protocol/blob/main/spec/profiles/conversations.md
14. N. Hardy. The Confused Deputy (or why capabilities might have been invented). ACM SIGOPS Operating Systems Review 22(4), 1988. https://css.csail.mit.edu/6.566/2011/readings/confused-deputy.html
15. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 9989, May 2026. https://www.rfc-editor.org/rfc/rfc9989.html
16. H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, P. Mittal. Bamboozling Certificate Authorities with BGP. USENIX Security, 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/birge-lee
17. SIMON roadmap. `docs/roadmap.md`. https://github.com/agnt-ai/simon-protocol/blob/main/docs/roadmap.md
18. S. Kitterman. Sender Policy Framework (SPF). RFC 7208, 2014. https://www.rfc-editor.org/rfc/rfc7208
19. K. Andersen et al. The Authenticated Received Chain (ARC) Protocol. RFC 8617 (Experimental), 2019. https://www.rfc-editor.org/rfc/rfc8617
20. Matrix Specification. https://spec.matrix.org/latest/

21. C. Lemmer Webber, J. Tallon et al. ActivityPub. W3C Recommendation, 2018. https://www.w3.org/TR/activitypub/
22. AT Protocol specifications. https://atproto.com/specs/atp
23. Nostr Implementation Possibilities, NIP-01 (basic protocol) and NIP-05 (DNS-based identifiers). https://github.com/nostr-protocol/nips/blob/master/01.md ; https://github.com/nostr-protocol/nips/blob/master/05.md
24. P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, 2011. https://www.rfc-editor.org/rfc/rfc6120
25. XEP-0220: Server Dialback. XMPP Standards Foundation. https://xmpp.org/extensions/xep-0220.html
26. P. Jones, G. Salgueiro, M. Jones, J. Smarr. WebFinger. RFC 7033, 2013. https://www.rfc-editor.org/rfc/rfc7033
27. Decentralized Identity Foundation. DIDComm Messaging v2.1. https://identity.foundation/didcomm-messaging/spec/v2.1/
28. IETF WIMSE Working Group. WIMSE Architecture, draft-ietf-wimse-arch-08, July 2026. https://datatracker.ietf.org/doc/draft-ietf-wimse-arch/
29. SPIFFE: Secure Production Identity Framework for Everyone. https://github.com/spiffe/spiffe
30. R. Barnes et al. The Messaging Layer Security (MLS) Protocol. RFC 9420, 2023. https://www.rfc-editor.org/rfc/rfc9420
31. IETF MIMI Working Group. MIMI message content, draft-ietf-mimi-content-09, July 2026. https://datatracker.ietf.org/doc/draft-ietf-mimi-content/
32. IETF KEYTRANS Working Group. Key Transparency Protocol, draft-ietf-keytrans-protocol-05, July 2026. https://datatracker.ietf.org/doc/draft-ietf-keytrans-protocol/
33. M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, M. J. Freedman. CONIKS: Bringing Key Transparency to End Users. USENIX Security, 2015. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara
34. Bluesky. Kicking off the ATP Working Group at the IETF, April 2026. https://atproto.com/blog/kicking-off-the-atp-working-group
35. R. Barnes, K. Bhargavan, B. Lipp, C. Wood. Hybrid Public Key Encryption. RFC 9180, 2022. https://www.rfc-editor.org/rfc/rfc9180
36. B. Schwartz, M. Bishop, E. Nygren. Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records). RFC 9460, 2023. https://www.rfc-editor.org/rfc/rfc9460
37. J. Schaad, B. Ramsdell, S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification. RFC 8551, 2019. https://www.rfc-editor.org/rfc/rfc8551
38. P. Wouters, D. Huigens, J. Winter, Y. Niibe. OpenPGP. RFC 9580, 2024. https://www.rfc-editor.org/rfc/rfc9580
39. A. Backman, J. Richer, M. Sporny. HTTP Message Signatures. RFC 9421, 2024. https://www.rfc-editor.org/rfc/rfc9421