LEV WHITE PAPER

Make it true, and prove it

An agent system that proves its work: on one machine, across many machines, and for groups of people

ABSTRACT

An agent says that its work is done, and nobody else has checked. Lev answers with one rule: a thing carries what is needed to check it. You write the outcome that you want as one sentence, with the claims that must hold. Lev works until a check that the builder does not control says that each claim is true, or until the work reaches a limit that you set. It tells you which of the two happened, and it gives you a receipt that another person can check again. On one machine, the work stays inside the limits that you set, and an agent can use a secret of yours and never see it. Across machines, the machines of many owners work together with no company in the middle, and each one checks what it gets by its own rules. Among people, your double is an agent that acts for you inside limits that you set, and a commune is a group that counts what each member gives toward a shared goal.

CONTENTS

1. Work that looks done
2. The thesis, and the words this paper uses

PART I. ONE MACHINE

3. Define, build, prove, decide, record
4. FlowMind: a flow is data, with limits
5. Runs and receipts
6. Poly Protocol: one action, every surface
7. Web MCP shims: an API layer to websites
8. Who may do what
9. The graph

PART II. MANY MACHINES

10. The mesh
11. Lev beside Freenet

PART III. PEOPLE

12. The double
13. The commune and the spare machine
14. Teaching, credit and the free society
15. Threats, and how Lev answers each
16. Open questions
17. Related work
18. What the person holds

---

1. Work that looks done

An agent ends a task with a report. The report is fluent. It names the files that changed and the tests that ran, and it says that the task is done. A person who reads it cannot tell, from the report alone, a finished task from one that only looks finished. Good prose costs a model little, so the quality of the prose is no evidence of the quality of the work.

Think of a painter who paints a room and then writes the inspection report on the same room. The report can be honest and still miss the patch behind the door, because the same eyes made both. The person who paid for the room reads the report and has no way to know.

The usual remedy is a second look: the person asks the agent to check its own work. That second look starts from the same reading of the task and the same story of what was done. It checks the report against itself, and an error in the first reading of the task survives it. The problem is old in another form. A classic paper on trust showed that a tool that builds programs can hide harmful behaviour in what it builds, where no reading of the text of the program finds it [1]. Who or what does the check matters as much as what the check reads.

Many tools record what an agent did: a log of its steps, a monitor that watches it run, a receipt from one tool. Each answers a narrow question. None of them says when a step of an agent may count for good, as a finished task or as a claim that others build on. The rule of Lev is that a claim with no proof never becomes lasting only because the run that made it ended with no error.

Lev starts from the request that a person wants to make: make this outcome true, and prove it. The hard half is the second. A proof that only the builder can run, or that only the builder has read, has the weakness of the second look. The next section gives the rule that Lev takes from this, and the rest of the paper shows how Lev applies it on one machine, across many machines, and for groups of people.

2. The thesis, and the words this paper uses

A thing carries what is needed to check it. Lev applies this rule to three things: a piece of work, a check, and a thing that many owners hold.

For a piece of work, the claims about its outcome come first. Each claim has a proof that someone other than the builder can run again: a program that answers pass or fail, or a fresh reader, which is a person or an agent that took no part in the work. For a check, Lev decides pass only on evidence that it can read. An answer that points at evidence that is not there is not a pass, whatever the answer says. For a thing that many owners hold, the thing travels with its rules, so that each holder can check its own copy and needs to ask nobody.

The idea is older than agents. In proof-carrying code, a program arrives with a proof, and the machine that receives it checks the proof by its own rules before it runs the program [2]. In the supply chain of software, a statement about a piece of software keeps three parts apart: the piece, the claim about it, and the signed wrapper that shows who made the claim [3]. A transparency service gives receipts that a third party can verify [4]. Lev applies the same rule to the work of agents, in a loop of five parts: define, build, prove, decide and record.

Lev gives this rule a shape that a person can hold. Each run leaves one receipt: what was asked, what ran, under which authority, with which result, and what was checked. You can hand the receipt to another person, and that person can run the checks again.

One request, from start to end

Here is the whole idea in one example. A person asks Lev: "Every invoice for November is sent, and each one matches its order." Lev turns that sentence into claims. The total of each invoice equals the total of its order. Each invoice for November has gone out. No customer gets the same invoice twice. Each claim gets its proof. One program compares each total with its order. Another program reads the record of the mail service, and looks for each invoice there once and none twice. A fresh reader reads a sample of the invoices against the orders, for what a program cannot judge, such as the right name on the letter.

Then the work starts. A worker writes the invoices, and the check of the totals runs before anything goes out. One total is wrong, so the check fails. The failure goes back to the worker as the input of its next try, and Lev counts the try against a limit that the person set. On the next try the totals pass, and the worker sends the invoices. Then the checks of delivery run on the record of the mail service, and they pass. The person gets a receipt that names the request, the claims, the checks and their answers. If the person doubts it, they hand it to someone else. That person runs the same programs and gets the same answers, and can put the questions of the reader to a reader of their own.

The story of this paper

The paper follows three questions, one for each part. Part I, on one machine, asks: can I trust what it did? Part II, on many machines, asks: can my machines work with other people's machines, with no company in the middle? Part III, on people, asks: what can a group of people do with it? The close gives the threats and how Lev answers each, the open questions and the related work, and then returns to the question of section 1. The words that the paper uses stand below, with one meaning each.

DEFINITIONS

outcome	What is true when the work is done, said so that a person who did not do the work can check it.
claim	One sentence about an outcome that must hold, or that must never hold, with the proof that decides it.
proof	What decides a claim: a program that answers pass or fail, or a fresh reader who answers with the evidence.
fresh reader	A person or an agent that took no part in the work it reads.
check	One run of a proof. Its answer is pass, fail or could not check. Could not check is never a pass.
control	A planted fault that a check must fail. A check that passes its control cannot see that fault.
flow	Work written as data: steps, gates, and limits on attempts, failures in a row, time and commands.
run	One execution of a flow or an action, with an id and a record of each thing that it did.
receipt	The record that a run leaves: what was asked, what ran, under which authority, with which result, and what was checked.
action	One thing that Lev can do, declared once. Lev makes each way to call it from that declaration.
surface	A way to call an action: from the command line, from another program, over the web, or from an agent. Section 6 names each one.

lease	A grant that names the holder, the exact thing, the operations and an end time. Its giver can revoke it at any time.
keeper	The part of a node that holds a private thing and uses it for a caller under a lease. No operation returns the private thing.
node	One installation of Lev on a machine. Each node works for its owner.
owner	The person who decides what a node does and what it shares.
shared thing	State that travels with its rules, so that each holder of a copy can check the copy.
double	An agent that acts for one person in one part of that person's life, under a lease.
commune	A group of people and their machines round one goal, with rules of its own and a count of what each member gives.

Part I. One machine

Can I trust what it did?

3. Define, build, prove, decide, record

Each piece of work in Lev goes through one loop: define, build, prove, decide, record. Figure 1 draws the loop and who acts in each part. This section says what happens in each part, then how Lev turns the answer of a check into a decision.

The five parts

Define comes before the work. The owner of the work writes one acceptance sentence and its claims. Some claims must hold, and some must never hold. Each claim has its proof: a program, or a question for a fresh reader. The person who asked sees the claims before the work starts.

Build gives each write scope a worker of its own. A change of behaviour comes with a test that failed before the change and passes after it. A test that never failed can pass for the wrong reason.

In prove, a fresh reader answers each claim with pass, fail or could not check, and gives the path and the line of its evidence. The reader writes its answers in a file outside the work, where the builder cannot edit them.

In decide, the owner runs each program check again. Two words mark how far a piece of work has come. Work is accepted when no answer is fail, each answer of could not check names the command that settles it, and each finding is closed or has a record of debt. Work is verified only when someone other than the builder ran each program check again and each passed, or when the owner of the product watched the acceptance sentence pass. Accepted is never the same as verified.

Record writes the plan, the handoff, one decision for each ruling and one record of debt for each open fault. The next person who picks up the work starts from these records and not from memory.

The order matters. Claims that are written after the work, to match it, cannot fail it. That is the first way in which the loop goes wrong, and the reason why define comes first. The fresh reader of prove is the answer to the second look of section 1, in which the builder checked its own report.

Faults planted on purpose

A check that cannot fail proves nothing. So Lev plants faults on purpose. A control is a planted fault: a run of the same check on an input that is known to be bad. The check must fail its control. Mutation testing uses small changes to code in the same way, as faults that the tests should find [5]. Lev rejects a check whose control does not run the program of the check itself, because a control that runs another program says nothing about this one. At the end of a piece of work, the owner runs each planted fault once more, together with each program check.

From an answer to a decision

The evaluator of Lev turns the answer of a reviewer into pass, fail or could not check. It passes an answer that says the intent is kept and cites its evidence. It fails an answer that says the intent drifted, and an answer that decides nothing. When no answer comes, it says could not check. It does not decide pass when it cannot read the evidence that an answer names. Could not check is never a pass.

A check that exits with status 1 fails, also when its output reads "ok". The exit code decides, and the words that a program prints do not. When a check declares a report of its tool, the findings of that report can decide too.

A gate is the step of a flow that takes an answer and chooses where the flow goes. It takes an answer of the declared form and returns a decision that names the branch it chose. A flow declares a branch for pass, a branch for fail, and may declare a branch for could not check. When the gate has a folder for its decisions, it saves a record of its route there and returns a reference to that record. When it cannot save the record, it answers could not check and chooses no branch. So every route that a flow takes has its record. Figure 2 shows the three answers and what a flow does with each.

What a failure does is never a key on the check. The check only answers. The flow that uses the check declares the route for each answer, so one check can stop one flow and only warn in another.

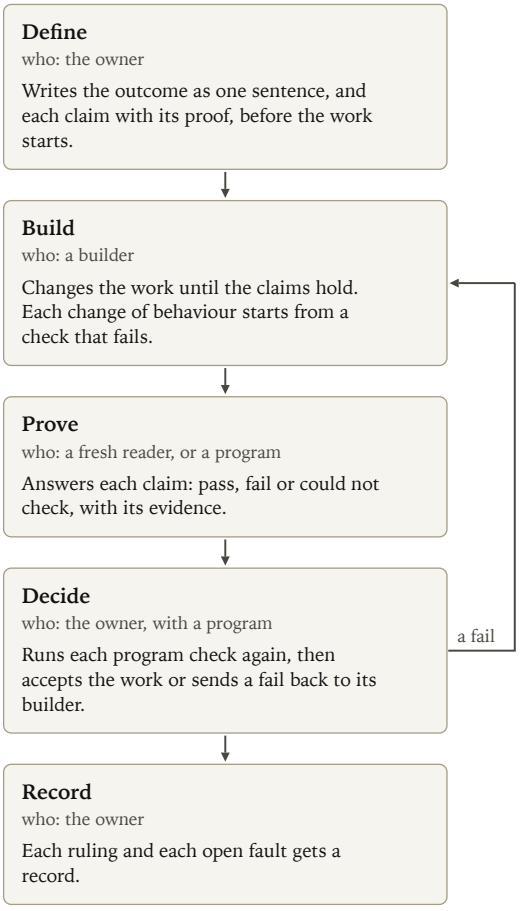
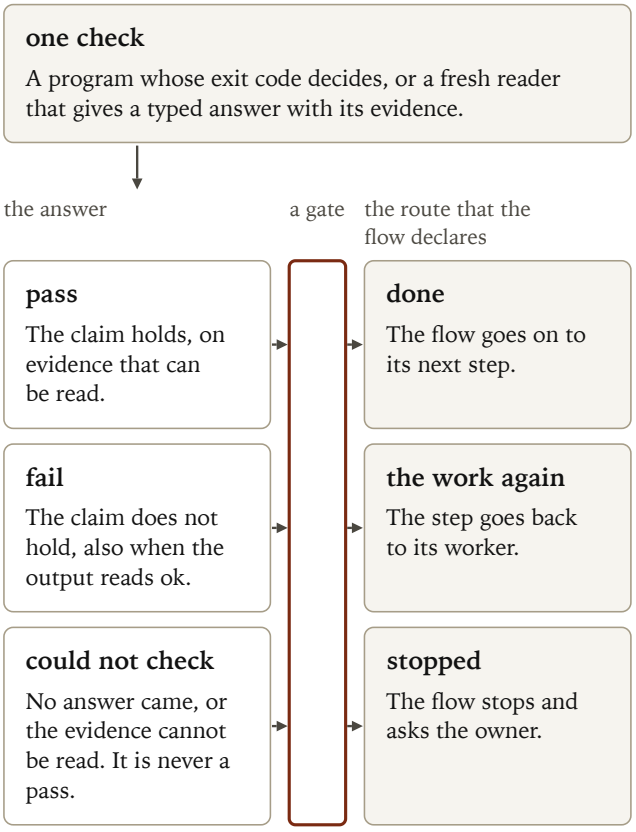


Figure 1. The loop of one piece of work. One piece of work passes through five parts. The owner writes the claims before the work, a builder makes them hold, and a fresh reader or a program answers each one. The owner runs each program check again, accepts the work or sends a fail back, and records each ruling and each open fault.



A flow declares its own route for each answer. These three routes are one example.

Figure 2. Three answers, and what a flow does with each. One check gives one of three answers: pass, fail or could not check. A gate takes the answer to the route that the flow declares for it. Could not check is never a pass.

4. FlowMind: a flow is data, with limits

A flow in Lev is a small map that a person can read: steps, gates, routes and limits, written as data in a file. FlowMind is the language of these files. This section says how Lev reads a flow and how it holds a running flow to its limits.

A flow is data

A flow file names its steps and the routes between them. A step can be the work of an agent, a command, or a gate that reads the answer of a check and chooses a branch (section 3). Because the flow is data, a person can read it before it runs, an agent can write one, and a program can check it. When the parser of Lev loads a flow, it freezes the document, its header, its list of steps and its first step, so that nothing changes the flow while it runs.

In a flow over a list of items, one item can depend on another. When an item fails, each item that depends on it is blocked and does not run. The other items still run.

A loop stops

A loop that tries again after a failure needs an end. The limits of a loop are settings in the flow file, never in code, so a person sees them before the flow runs. A flow has four limits:

- Attempts: a loop stops at its attempt limit, also when every gate answers pass.
- Failures in a row: a run of failures stops the flow sooner, before it reaches its attempt limit.
- Time: when the time limit passes, the run stops with the reason "time limit" and a failing exit status. It does not wait for a long command to end.
- Commands: when a loop has used its number of commands, the run stops with the reason "command limit" and a failing exit status.

A task and the runtime can replace a limit of the flow file. A reached limit is a fact of the run, and not the verdict of a check. A flow that stopped at its attempt limit did not pass. It ran out of attempts, and the receipt says so.

A reached limit can lead somewhere

A flow file can declare the route that a reached attempt limit takes: for example, to a step that asks a person for help. Then the run follows that route, ends normally, and records that the limit was reached. Figure 3 draws one loop with its gate, its limits and the route of a reached limit.

Earlier work on limits

Other workflow languages also put some of these choices in the definition. In the Amazon States Language, timeouts, retries and failure routes are part of the state machine that the workflow declares [6]. A Common Workflow Language document declares how the inputs of a step come from earlier outputs, and the specification of that language fixes how a failure is reported or passed on [7]. A flow of Lev puts the four limits of a loop, and the route of a reached limit, in the same file as its steps.

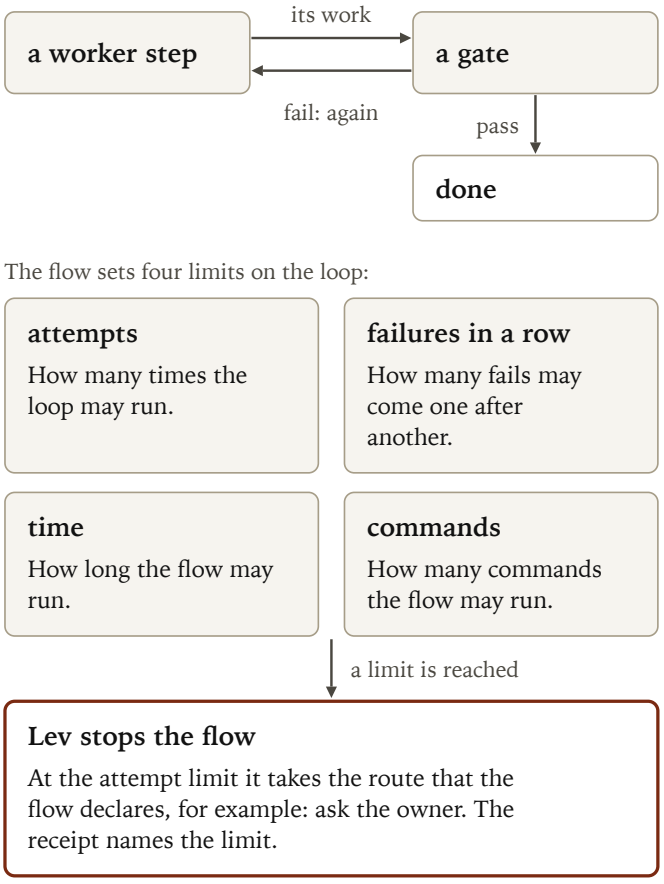


Figure 3. A loop with its four limits. A worker step and a gate form a loop, and the flow sets four limits on it: attempts, failures in a row, time and commands. A reached limit stops the flow. For the attempt limit, the flow can declare a route, and Lev follows it.

5. Runs and receipts

A run is one execution of a flow or an action. This section follows what Lev keeps about a run, and what its stores do when a write fails, comes twice, is cut off or meets a newer format. It ends with the receipt, the thing that the person holds at the end.

The run ledger

Each run has an id and its line of actions in the run ledger. The ledger is a file that Lev only appends to, and Lev replays the state of a run from it. The ledger keeps references to the decisions of checks. It computes no verdict itself: the verdict belongs to the evaluator (section 3), and the ledger only records it.

When the same source is admitted twice, Lev gives one run and one line of actions. A request that arrives again by mistake does not start the work again.

Who may finish an action

A worker that takes an action gets a ticket for it, with an id. Lev refuses to complete the action when the call gives an id that is not the id of the ticket on that action. The idea is close to the sequencer of a lock service, which lets a resource refuse an operation from a holder whose lock is stale [8].

A decision does not verify an effect

An effect is a change that a run makes outside itself, such as a sent message or a written file. When a run records a decision, its effect stays a candidate. The envelope of such a run says that the effect is a candidate and that no proof backs it. An effect becomes verified only through a proof of the effect itself.

The store of records under faults

Beside the run ledger, Lev keeps records in a store of rows. A change of a row writes its event in the same transaction. A write that fails after the row change and before the transaction ends leaves neither the new version of the row nor its event. A reader never sees a row whose event is missing, or an event whose row is missing.

A write can carry a key. When the store gets a write with a key that it saw before, it returns the earlier result and does not change the row, also after other writes and after the store was closed and opened again. This is the property that HTTP calls idempotent, where a request sent again has the effect of one request [9].

The run ledger guards its own file

A crash can cut the last line of the ledger. A last line with no newline is an append that crashed before it reached the disk, so no caller was told that it succeeded. Lev loads each whole record before such a torn line, and the next append leaves a ledger that a strict read accepts. Lev also refuses each append to a ledger whose format is newer than the one it knows, and writes nothing.

Finding records by their links

Records link to each other: a decision to the plan that it changes, a run to its task. A search for one type of record also raises the records that a found record links to. Lev follows each link through an index of links, and does not read every row of the source to find them.

The receipt

The receipt is the record that a run leaves: what was asked, what ran, under which authority, with which result, and what was checked. Figure 4 places the parts of this section on the path of one call: a surface, the admission, a run in the ledger, an effect and a receipt. The receipt is the end of that path and the part that leaves the machine. A person can hand it to another person, who reads the claims, finds the checks and runs them again.

Supply chain systems use records of the same type. In in-toto, a signed layout describes the expected steps, and each step records evidence that a consumer checks against that layout [10]. A transparency service of the SCITT architecture registers signed statements and returns receipts that a third party can verify [4]. A receipt of Lev records the claims about an outcome beside the steps that produced it.

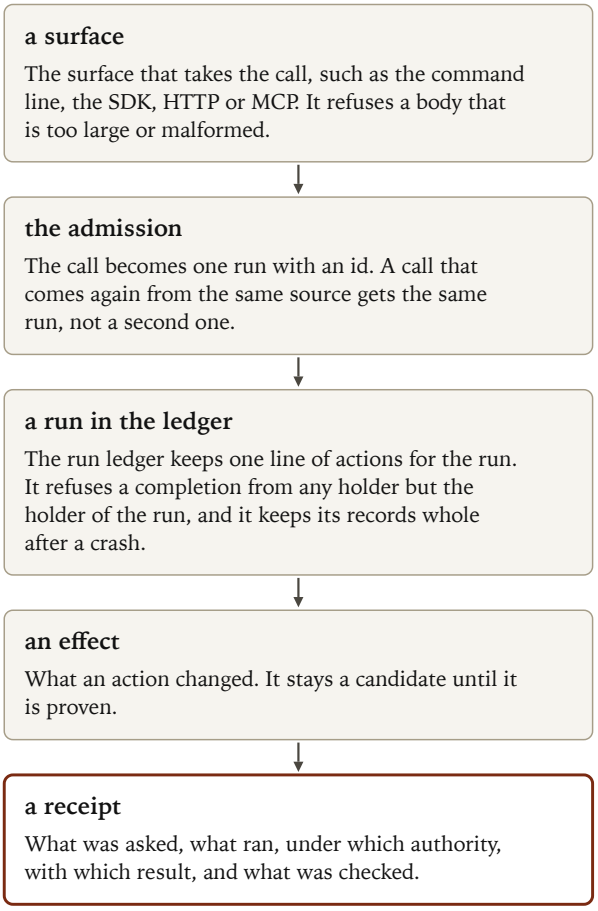


Figure 4. The path of one call. A call passes a surface, becomes one run, and leaves one line of actions in the run ledger. What the run changes stays a candidate until it is proven, and the run ends in a receipt that another person can check again.

6. Poly Protocol: one action, every surface

A person uses Lev from a phone, a chat, a terminal or another app. Each of these needs a way in. Poly Protocol is the rule that a capability is declared once, as an action, and that Lev makes every way in from that one declaration: the command line, the SDK, HTTP, MCP, gRPC and WebSocket. One action has one name everywhere. Figure 5 draws the declaration, the registry and the six surfaces.

The declaration and the registry

An action lives in the settings file of the module that owns it, under its list of actions. Lev compiles these declarations into a registry, and every surface reads the same registry. No surface holds a list of its own.

An action has a switch for each surface but the SDK. A surface that is switched off does not appear in the generated command names, HTTP documents, MCP tools, WebSocket handlers or gRPC services, and the SDK call stays. When an action turns every surface off, the registry makes no command, no HTTP path, no MCP tool, no UI action and no gRPC service for it, and it marks WebSocket and the surface for peers as off. A capability that only a schedule runs, such as a nightly job, declares its action with the command line off.

Each surface

For each action that has an HTTP path, the path is in the generated OpenAPI document, and a POST to that path on the HTTP listener reaches that action. The HTTP surface answers a health call with its state and the number of its bindings. It hands an action call to the action with the address, the binding and the body of the request, and it returns the answer of the action with its run id. It refuses a body that is malformed, is not an object or is too large, and the action does not run.

Each WebSocket path that the registry makes reaches its action on the WebSocket listener.

The generator makes a tool name for each action that has an MCP tool, and that list of names equals the list that the MCP transport resolves from its bindings. A tool can name a page as a resource. The MCP listener lists the resource and serves its exact HTML, and a call of the tool over MCP reaches the admitted action.

When the queue admits a run, the SDK opens one call, with one request event and one completion event, and the action runs once. The execution id on the completion event equals the id on the event with which the run engine started the run, so a caller can follow its call into the run ledger.

What the person gets

An action that a person starts from the terminal is the same action that an agent calls over MCP and that a phone calls over HTTP. It has one name, one check of its input, one run in the ledger and one receipt. A fix to the action fixes every surface at once, and a surface cannot drift away from the others, because no surface is written by hand.

Earlier work on surfaces

OpenAPI gives tools a machine-readable description of the paths, inputs and responses of an HTTP API [11]. MCP lets a client discover the tools and resources of a server: the client calls a tool, and it reads the data that a resource gives [12]. gRPC maps remote calls onto the streams of HTTP/2 [13], and WebSocket keeps a connection open for messages in both directions [14]. Poly Protocol makes each of these from one declaration of an action. In the terms of ports and adapters, the action is the port, and each surface is an adapter that connects one technology to it [15].

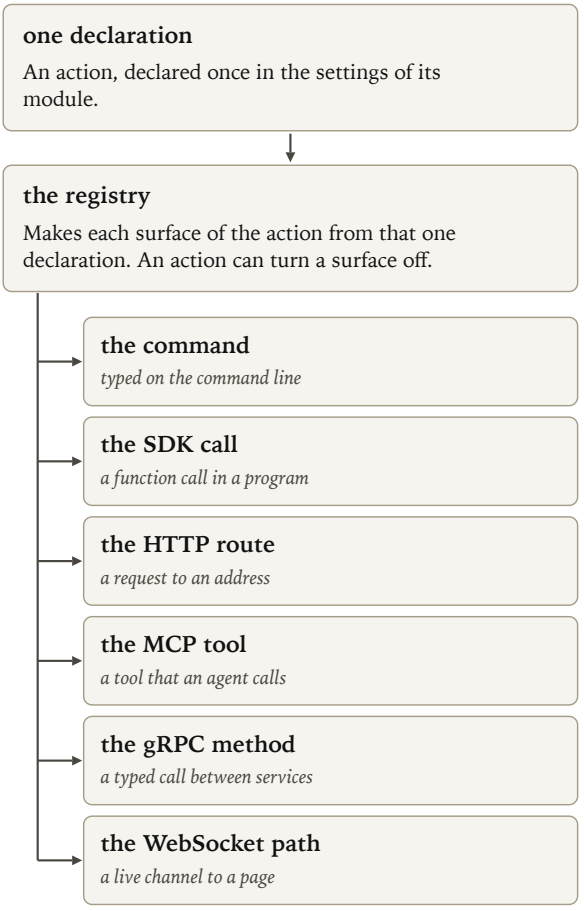


Figure 5. One action, six surfaces. A capability is declared once, as an action. The registry makes the command, the SDK call, the HTTP route, the MCP tool, the gRPC method and the WebSocket path from that one declaration, so each surface calls the same action by the same name.

7. Web MCP shims: an API layer to websites

Many websites have no API for agents. An agent that must use such a site has to read the page and press its buttons, and that is slow and easy to break. A proposed web standard, WebMCP, lets a website register tools for browser agents itself, with declared inputs and a description [16]. A Web MCP shim of Lev is for a website that has not done so: it gives the site a set of tools that an agent can call. Figure 6 draws the path from an agent through the daemon and the browser extension to the page.

The extension chooses the tools of a website

The browser extension of Lev holds tool definitions for the websites that it supports. Each definition declares the patterns of the addresses that it serves. When a page loads, the extension matches the address of the page against those patterns and gives the tools of that site, and only of that site. It refuses an origin that only looks like the origin of a supported website. Browsers use the origin to separate the authority of documents [17], so the choice of tools follows a boundary that the browser already draws.

The daemon passes a call only under a lease

The daemon of Lev runs on the person's machine. Its browser adapter takes the connection of the extension, passes commands to it and manages leases. A lease names the holder, the exact thing, the operations and an

end time. The adapter refuses a read when no lease is approved. After the person approves a lease, the adapter passes a read for the tab of that lease to the extension, and it holds no lease for any other tab. Under a read lease it refuses a script, a navigation, a read of cookies and a direct browser command. When the lease has ended, it refuses the read again.

This follows the principle of least privilege [18]: an agent that may read one tab may only read that tab, for that time.

What the person gets

The person keeps the browser, the sign-in and the cookies that they already have. The agent never sees a password or a cookie. It sees the tools of one site, under one lease, for a set time. The person approves the lease, and the lease ends at its end time or when the person revokes it. Each call through the shim is an action of Lev (section 6), so it leaves a run in the ledger and a receipt like any other call.

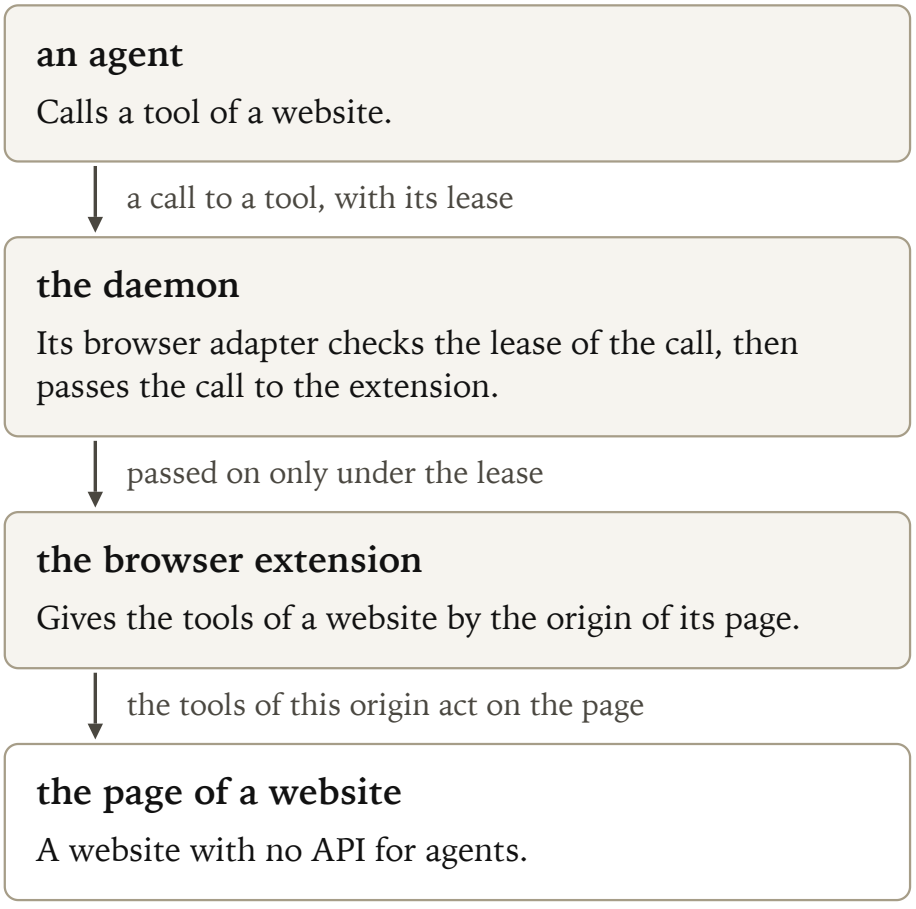


Figure 6. A website as tools for an agent. An agent calls a tool of a website through the daemon, which checks the lease of each call. The browser extension gives the tools of the site by the origin of its page, so a website with no API for agents still has tools.

8. Who may do what

An agent that works for a person needs some of that person's authority, and no more. This section says how Lev names a person, how a node decides whether an approval still gives authority, and how a keeper uses a

private thing for a caller without handing it over. Figure 7 draws the path from the log of a person through an approval and a lease to one use.

A person and a key

The first entry of a signed log fixes the id of a person. The entry names the method did:webvh 1.0, carries the id as its identifier, and holds a signed proof. The id is the hash of that first entry. It is not an email, the subject of a sign-in provider or a public key, so a person keeps it when any of these change. A later entry of the log can change the update key, move the host or end the log, and the first entry and the id stay as they were. Lev checks every entry of the log in order, from the log alone, so any holder of the log can check it.

The method did:webvh gives a web-hosted identifier a sequence of updates that a client can verify [19], inside the general model of decentralized identifiers [20]. KERI also records who controls an identifier as a signed sequence of key events [21].

An approval and its limits

A destination is the node that receives a call from a peer. When its owner approves a call, the destination saves the approval. An approval records one binding and one exact request, with its deadline, its principal and its lease. A revoke writes a separate record that lasts.

The destination compares each call with the approval that it saved. A call with a changed request gets no authority from it. A change of one field is enough: an added field, another caller, another peer, another key of the peer, another project or another binding. At the instant of its deadline an approval gives nothing, and a restart does not renew it. A revoked approval blocks a reader that held it before, it stays revoked after a restart, and the same request cannot be approved again.

The sender of a call

A destination maps the statement of a pinned adapter to the principal that sent the call, and it keeps the exact bytes of the signature. A sender can replace its transport key: the destination moves the principal to the new key, and it refuses to move a binding to another principal. A thief who has an old transport key cannot use it to take over a binding that belongs to someone else.

The keeper

A keeper is the one part of a node that reads a private thing: a key, a sign-in, the secret of a room or a private file. A caller, such as an agent or an app, asks the keeper for an operation under a lease, and gets the result of that operation. It never gets the private thing. An agent that must sign a message asks the keeper to sign it, and gets the signature, and never sees the key. Figure 8 draws a caller, a lease, the keeper and its checks.

The lease names the holder, the project, the exact thing, the operations and an end time. The keeper checks each of these, and that the lease is not revoked now, at the moment it runs the operation and not when the request arrives. One mismatch refuses the request. The validator of the keeper compares the holder, the runtime, the operation, the scope, the one-time number, the key against repeats and the number of the revocation with the lease. It refuses a use after the lease has ended, and a use from another boundary of the

runtime. A check at each access is an old principle of protection [18]. The keeper has a purpose near that of the delegates that manage private state in the current architecture of Freenet [22].

The keeper also limits what a caller can learn. No result and no error holds a part of the private thing. An operation whose answer depends on the private thing takes the same time for each answer, and has a limit of tries for each lease. A private thing leaves its keeper only as a sealed copy that one named keeper can open, and the owner approves each such copy. On disk, the private things of a node sit in an encrypted local vault, which refuses a key that does not open it and changes nothing.

The keeper states its own limits. A person with administrator rights on the machine of a keeper can read its key store. And no proof shows that an operation leaks nothing through its results, its errors or its time, so a person or a test reviews each operation that a keeper offers.

What the person gets

A person gives an agent a lease and not a key. The lease says who may do what, on which thing, until when. The person can revoke it at any moment, and the keeper refuses the next use. Every use leaves a receipt, so the person can read afterwards what was done in their name.

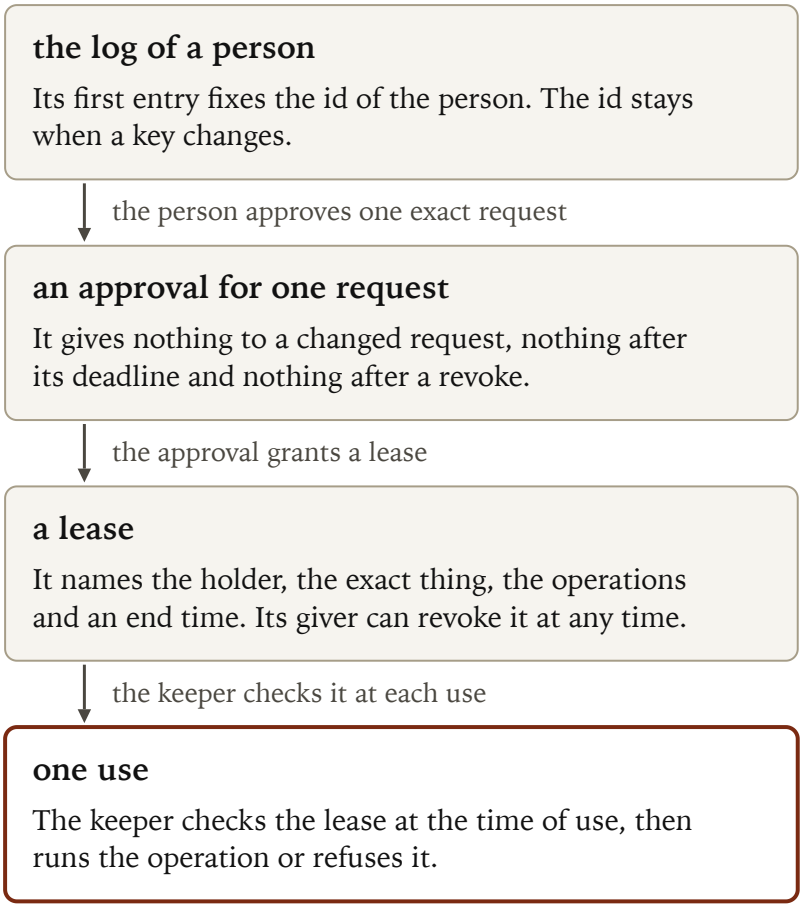


Figure 7. From the log of a person to the check of one use. A signed log fixes the id of a person. The person approves one exact request, the approval grants a lease, and the keeper checks that lease again at the time of each use.

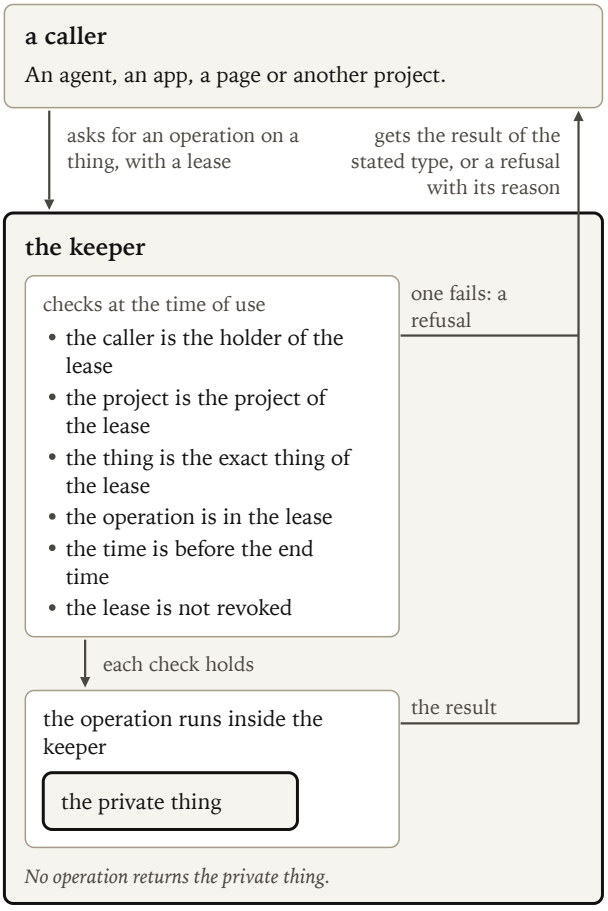


Figure 8. The keeper. A caller asks the keeper for an operation, with a lease. The keeper checks each field of the lease at the time of use, runs the operation on the private thing inside itself, and gives back only the result or a refusal with its reason.

9. The graph

Everything that this part has named so far needs a place to live: the claims, the flows, the runs, the receipts, the approvals and the decisions. In Lev they live in one graph that the owner holds. Each thing is an entry with a name and links.

One graph

An entry of the graph can be code, a spec, a decision, a session, a task, an idea, a run or a receipt. A decision links to the plan that it changes. A run links to its task and to its receipt. A receipt links to the checks that it records. A person who asks "how did we get here?" follows the links back from a result to the request and the decision behind it.

An entry holds its own spec. Lev makes the entry from that spec, and keeps checking that the entry and its spec still agree. When they drift apart, the check says so, the same way that a check of any other claim does (section 3).

Many views of one graph

Context, execution, skills, knowledge and code are layers over the same entries. A view is made from the graph, and nobody edits a view in place: a change goes into the graph, and every view that shows that entry changes with it. The screen that a person reads and the answer that an agent gets come from the same engine, so the two never disagree about what a thing is.

An agent starts with a small view: one entry and its neighbours. It asks for more detail where it needs it. It does not load a whole project to answer a question about one decision.

Earlier work on views

The idea of many views of one system is older than Lev. The 4+1 view model gives each concern of an architecture its own view and joins the views with scenarios [23]. The C4 model shows one system at several levels of detail [24]. Lev makes each view from one graph, keeps it true to that graph, and lets nobody edit a view in place.

The graph is where part I ends. One machine can now state an outcome, do the work, prove it, and keep the proof where a person and an agent can find it. The next question is what happens when the work needs more than one machine, and the machines belong to different people.

Part II. Many machines

Can my machines work with other people's machines, with no company in the middle?

10. The mesh

A person has more than one machine, and works with people who have machines of their own. The mesh is how the nodes of many owners find each other, reach each other and work together, with no company in the middle. Each node works for its owner. Figure 9 draws nodes of several owners and the paths between them.

Nodes find and reach each other

A node finds a node of another owner and reaches it hop by hop, directly when it can and through a relay when it cannot. It hands that node a request, and the destination acknowledges it. There is no centre that every request passes through, and no node that the others must trust.

Each request travels in an envelope. The envelope carries a digest of the input, the operation, a key against repeats and a time budget. A destination refuses an envelope in which one of these changed on the way. An envelope for a write also carries the version of the parent that the write expects, the version of the state, the number of its lease and a checksum. A route entry records which surfaces to prefer for a peer, where the peer should be, and its last known health.

Where a group of nodes must agree, Lev counts a strict majority. Raft commits new entries of its log by the agreement of a majority [25], and Lev uses the same rule for a vote among nodes.

One writer for each row

Each row of shared state has one writer: the node that made it. Other nodes hold copies and do not edit them. A newer version from the owner replaces a copy whole. To change a row that another node owns, a node sends a request to the owner of the row, and the owner decides. Lev needs no merge rule, because two writers never write the same row. Replicated data types take the other path and let any node edit and merge [26]; Lev keeps one writer, so each row has one owner who answers for it.

Between nodes, Lev sends a whole versioned document, and never a log of events or a delta. A reader asks the origin for the rows after its cursor. The origin keeps a set number of its last changes. When the cursor is older than the changes that the origin keeps, is of another generation, or is ahead of the origin, the answer is that the reader needs a new snapshot.

Share one thing with one person, for a set time

A person shares one thing with one other person for a set time, and can take it back. The share is a lease (section 8): it names the holder, the exact thing, the operations and an end time, and the giver can revoke it. The other node gets that one thing, and nothing else of the person's graph. When the lease ends, the next use is refused.

Nobody is believed on their word

A node does not believe a claim because a known node sent it. Each node checks each claim by its own rules, and the node that a request comes to decides it, never the sender. A change to the graph from another node comes in as evidence, and does not change the local graph by itself. When one node checks a claim and signs its view, that signed view is only a claim at the next node, which checks the evidence again. Two nodes may apply different rules to the same claim, and both are right by their own policy.

The shared thing

A shared thing is state that more than one node holds, with the rules that say which state is valid. A holder checks its copy against the rules that travel with it, with no call to another node. Holochain has peers check shared entries with the rules of the application [27].

A shared thing has two addresses (Figure 10). The address of one instance is a hash over the hash of its rules and its parameters, and it never changes. For a thing with an owner, the parameters hold the id of the owner's log and a label, so two owners never share an instance. The name is that id and that label. It points at the current instance in a head record that the writer key of the owner signs. Each head holds a generation and the digest of the head before it. IPFS keeps an address by content apart from a name that points at data that changes [28]. Lev has a pointer of the same shape: the name stays, and the head that it points at moves. The address of an instance differs from an address by content, because it is a hash of the rules and the parameters and not of the state.

So an address does not prove whose state a copy holds or which version it is. A state record that the writer key signs says that, and only a live answer of the owner's writer says which state record is the newest. A reader keeps the highest generation that it checked and refuses a lower one. A first read has no such floor: a reader that has never seen a thing has no generation to compare its first answer with. Section 16 keeps this as an open question.

What the person gets

A person's laptop, phone and home server join one mesh of that person. That mesh works with the mesh of another person one shared thing and one lease at a time. No company holds the data in the middle, and no node can act for another owner.

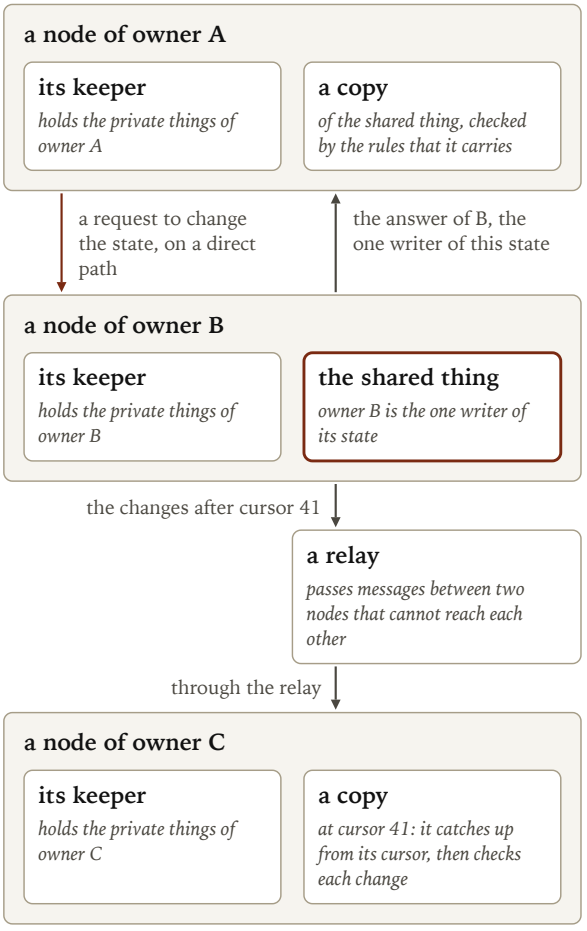


Figure 9. Nodes of many owners. Three nodes of three owners, each with its keeper and a copy of one shared thing. A request goes on a direct path to the one writer of the state, which answers it. A copy on a node that the writer cannot reach directly catches up from its cursor through a relay.

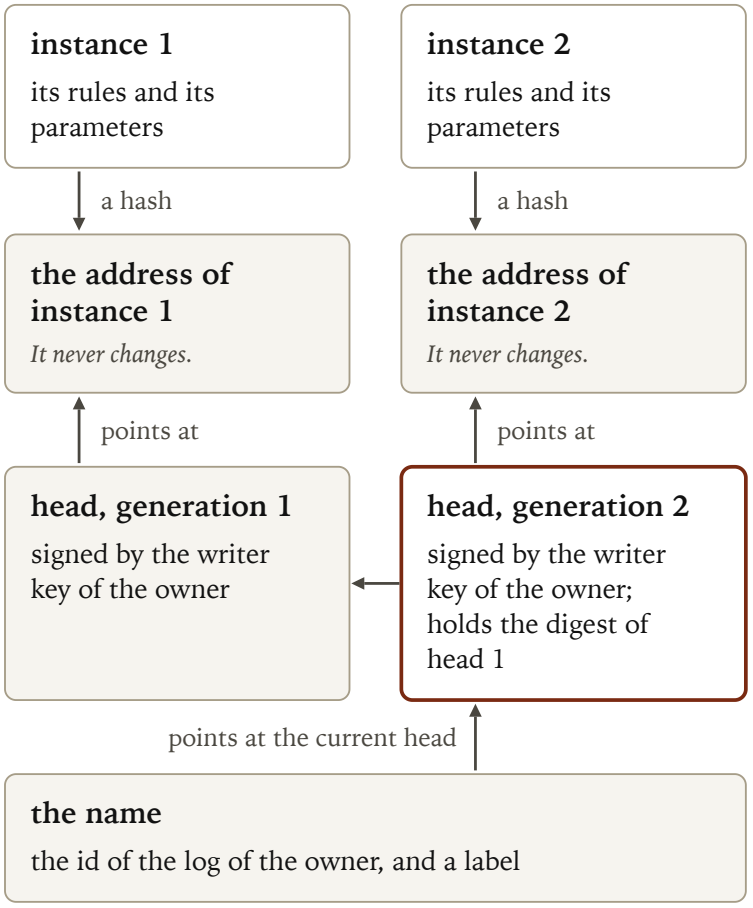


Figure 10. The two addresses of a shared thing. A shared thing has two addresses. The address of an instance is a hash of its rules and its parameters and never changes. The name stays the same, and a chain of heads signed by the owner says which instance the name means now.

11. Lev beside Freenet

Freenet is the nearest neighbour of the mesh of section 10. Lev takes two patterns from it and leaves the rest. Lev takes no code of Freenet's and relies on no program of Freenet's.

What Freenet solves

Freenet began as a network in which peers pool their storage, keep and find files, and adapt their routing from experience, with the aim to protect the anonymity of the people who publish and read [29]. Its current design has contracts that say how shared state merges, delegates that manage private state, and routing between peers that adapts [22].

A contract is a small program together with parameters that are fixed when the contract is made. The program says which state is valid and how two states merge. Each peer that holds a copy of the state runs the same program, so it needs no trust in the peer that gave it the copy. It drops a whole state that is invalid before any merge. For a delta, an update that holds only what changed, it checks the merged result and refuses the delta when that result is invalid [22]. The key of a contract is a hash of the hash of the code joined with the parameters, so a change of either gives a new key. The design asks of a merge that it gives one state when merges come in any order and any number of times, which is the shape of a state-based replicated data type [26]. Its paper gives one case in which this does not hold for each grouping. For a log that keeps a set number

of records, two groupings of the same merges can give different states. A merge of the two results removes the difference [22]. Any peer that can reach a contract can read its state, so an app that wants privacy encrypts what it stores [22].

A delegate is a private program on the node of a user. It holds secrets, and other code only sends it messages and reads its answers, so a caller can use a secret without getting it. A delegate cannot open the network or a file itself; it asks the node, which carries its requests to contracts [22].

The network puts each peer and each contract at a place on one ring, and copies of the state of a contract sit on the peers near its place. A peer that has enough links aims a new link at the largest gap between its neighbours, on a scale of the logarithm of the distance. The aim is the shape of a small world, in which a local search can find short paths [30]. A router predicts the delay, the chance of failure and the rate of each neighbour from past results, and it uses the predictions when it chooses where to send a request. A request to connect does not use them [22].

What Lev takes, and what it leaves

Lev takes two patterns from Freenet. The keeper (section 8) takes the shape of a delegate: one part of a node holds a private thing, and a caller gets the result of an operation and never the thing. Lev adds a lease that the keeper checks at each use. The address of a shared thing (section 10) takes the address of a contract. The address of an instance is a hash over the hash of its rules and its parameters. The parameters of a thing with an owner hold the id of the owner's log.

Lev leaves out other parts. Freenet lets any two valid states merge [22]. Lev gives each row one writer and has no merge. Freenet places state on its ring by a hash [22]. In Lev a reader asks the node of the owner or a node that holds a copy, and there is no lookup by hash across the network. Before it sends a whole state, Freenet can send only a summary toward the hosts of a contract, so that the sender then sends only what a host lacks [22]. Lev sends a whole versioned document and no deltas. Freenet writes the rules of a contract as a program in WebAssembly [22]. Whether the rules of a shared thing in Lev run as such a program or as a Lev record type with its checks is an open question of section 16.

Part III. People

What can a group of people do with it?

12. The double

Parts I and II describe what a machine does and how machines work together. This part is about what that gives people. It starts with one person and the agent that acts for them: the double.

A double is an agent that acts for one person in one part of that person's life, under a lease. It is the person's own agent. It runs on the person's own node, or on a machine of the person's commune while that node is off (section 13). When it posts or acts, it shows as the person, with a mark that says an agent did it. Others can see which posts came from the person and which from the double.

One double for each room

A life has rooms: the family, the work, the street, the town. Your double goes into each room with a lease of its own, and the lease says what it may do there and until when. Figure 11 draws one person, four rooms and a double in each. Here is one task in each room.

- Family. "Who picks up on Thursday?" The double reads your calendar and proposes a name. You tap yes.
- Work. It drafts the weekly update and waits for you. Nothing goes out until you send it.
- The street. It takes your turn on the co-op order.
- The town. It holds your place in a queue for an appointment.

For each room you choose one of three settings: stay quiet, propose, or act. A double that stays quiet works for you alone: it watches, keeps notes and writes drafts, and it sends nothing to anyone. A double that proposes sends you a proposal and does nothing until you answer. A double that acts does the work inside its lease, and tells you afterwards. You can change the setting of a room at any time, and the next run of the double reads the new one.

A wall stands between the rooms. The double in the town room knows nothing from the family room. When you join a shared project, the others in it do not see what your double saw elsewhere, and they cannot act for you. Each double comes home with one line of receipt: what it was asked, what it did, and what it checked.

How a double decides

A double looks after a responsibility that you give it: a goal, the standards that guide its judgment, the grants it may use, and how often it reviews. When a review is due, Lev admits one bounded run of the double. The run reads your current grants, your notes and its earlier decisions. It ends in one of three ways: it stays quiet, it proposes, or it acts. Each end leaves a record: a decision to stay quiet, a proposal, or an action with its receipt.

A double can start a smaller goal inside its responsibility. It cannot widen its own grants by proposing a useful goal. An interpretation of what you said gives it no authority: if you say "I want to lose weight" as an example in a conversation, your double does not open a health goal. A wish that you mean comes back later as a question, and becomes a goal only when you say so.

A double also meets other people's doubles. Your double can make friends with the double of another person first, and ask you before it introduces the two of you.

A day with your double

Here is one day, as an example of what a double does and does not do.

- 07:30. The night's messages are in three piles: done, needs you, can wait.
- 09:00. The car needs brakes. The double asks three garages for a price.
- 12:30. Three prices are back. It proposes the middle one and says why. You tap yes.
- 13:00. A form asks for your home address. Your rule says no. The double refuses, and the refusal is on the list.
- 15:00. A friend's double and yours find one free evening for both. Neither sees the other's calendar.
- 18:00. One thing it never does alone is pay. It asks, and you tap yes.
- 22:00. One page: what it did, what it asked, what it refused. Each line opens into its receipt.

The page at the end of the day is the same thing as the receipt of part I, made for a person. A line that says "done" opens into what was asked, what ran, under which lease, and what was checked.

What the double stands on

Each part of the double comes from part I. Its lease is the lease of section 8, and each use of it is bound to a receipt. When the double must use a private thing, such as a sign-in to a garage's website, it asks your keeper for one operation and never holds the sign-in itself. Its runs have the limits of section 4, and its receipts link into your graph.

The first principle of section 14 holds for every double: it is never creepy. Every piece of data that a double collects or shares is collected with consent, for a scope, open to revoke, and readable by the person it is about.

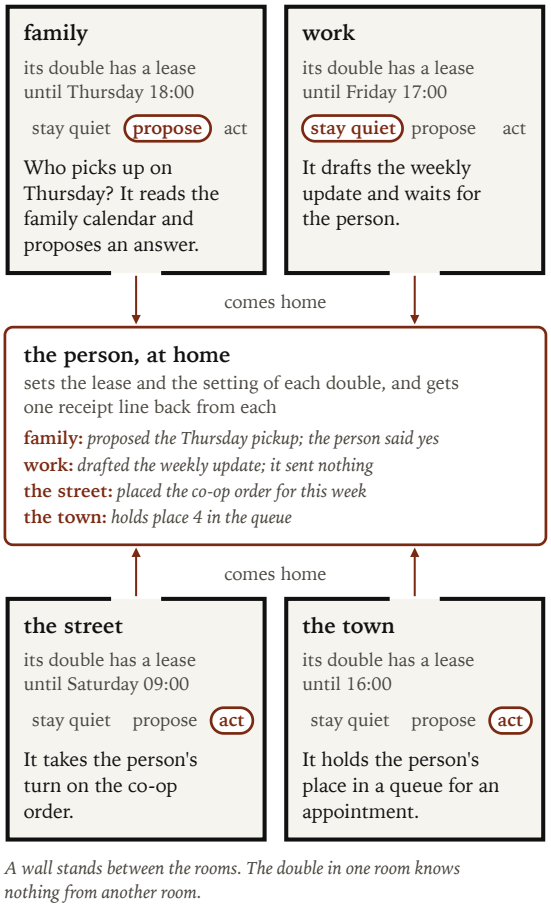


Figure 11. The double and its rooms. A person sends a double into four rooms: family, work, the street and the town. Each double has a lease with an end time, a setting of stay quiet, propose or act, and one task. A wall stands between the rooms, and each double comes home with one receipt line.

13. The commune and the spare machine

One person with a double gets their own work done. A group of people can do more. A commune is a group of people and their machines round one goal, with rules of its own and a count of what each member gives. Communes can be small, and there can be many of them across the world. This section goes down into one of them.

One street, one goal

Take one street of six homes. Its commune has one goal, written as one sentence: "Every child on our street gets to school on time." The goal is an outcome in the sense of section 3. It has claims, and each claim has its check: each child has a ride on each school morning, and each ride arrives before the bell. On Friday, when every morning of the week passed its check, the goal is proven, and the street gets a receipt like any other.

Members give different things. One gives a seat in a car on two mornings. One gives a spare machine. One gives an hour of a double's time. Each gift leaves a receipt, and each receipt adds to the count beside the member who gave it. Figure 12 draws the goal, the members, their gifts and the count.

A member agrees to give something. That is a commitment. A member can also let a double act for them inside a scope. That is a delegation. The two have different results, and Lev keeps them apart: a commitment

says what you will give, and a delegation says what your double may do. Joining the commune shows the others nothing that you have not chosen to share, and lets nobody act for you. When you leave the commune, your commitment ends, and your own goals stay yours.

Each commune keeps its own rules

A commune keeps its own rules, its own words and its own count. You join by an invite, and you can leave at any time. One commune may count an hour of driving the same as an hour of a machine, and the next may not; each is right by its own rules. No single score covers the world, and no one sees your count unless you show it.

A problem comes up. One family on the street needs the school's notices in another language, and no member speaks it. The commune next door has a member who translates. A wall stands between two communes, and the wall belongs to both. The two communes agree, one after the other, to open one shared border. Only then does that one border open, and one piece of work crosses it: the translation of the notices. Every other border of both communes stays shut, and a piece of work that meets a shut border goes back.

The spare machine

One member has an old laptop in a drawer. The member takes it out, plugs it in and accepts one invite. It joins the commune.

At night, when the house sleeps, the machine does three jobs, one after the other.

1. It keeps a sealed copy of the street's shared photos. Sealed means that this machine cannot read them. Only the keepers of the members can open them.
2. It runs a neighbour's double while the neighbour's own machine is off.
3. It adds up the co-op's order for the morning.

In the morning, a tally stands beside the member's name: seven hours given. At noon the member's own laptop is closed, and the member's double runs on a neighbour's machine, with the hours that the spare machine earned. A double that runs on another member's machine carries no private thing with it. When it needs one, it asks the keeper of its owner for one operation under a lease, as in section 8. The machine that runs it gets the use of the thing by reference, and never a copy.

One day the member unplugs the spare machine. The member's things go home with it, and the sealed copies that it held move to another machine of the commune. Nothing is lost.

What a commune stands on

A commune is the mesh of part II with a goal. Its gifts and its counts are receipts, and each node checks them by its own rules, so no member can add to their own count by saying so. Its border is a lease between two communes: it names the work that may cross, either side can close it, and each crossing leaves a receipt.

Lev works at every scale, so the commune does too. A node runs on a small, old machine with no network, keeps working when the connection is gone, and catches up from its cursor when the connection comes back.

Many communes, each with its own rules, can work together across the world, one shared border at a time.

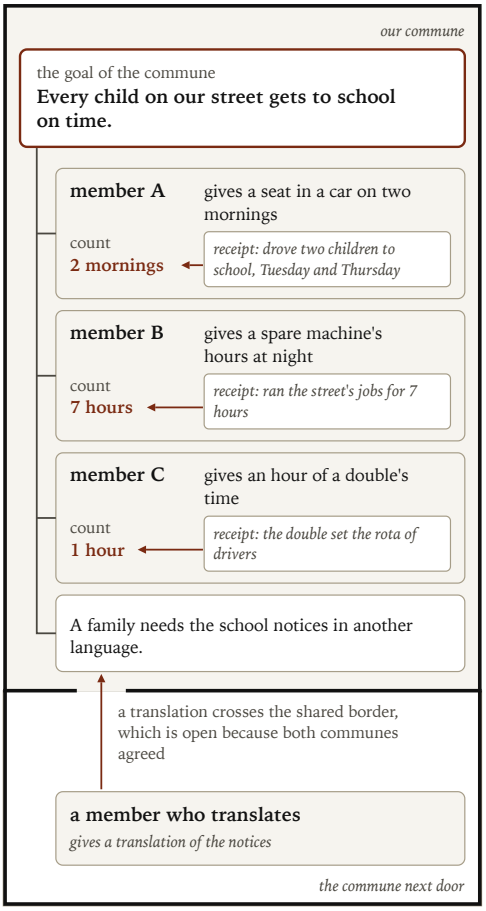


Figure 12. What each member gives, and how it is counted. A commune forms round one goal. Each member gives something of a different type, each gift leaves a receipt, and each receipt adds to the count beside the member who gave it. The border with the commune next door is open because both communes agreed, and a translation crosses it.

14. Teaching, credit and the free society

A commune grows when its members learn from each other. This section shows how a member teaches a skill once, how every double in the commune can then use it, and how the credit comes back to the teacher. It ends with the stance of Lev on a free society of people and their agents.

Teach it once

One member knows how to appeal a parking fine that is wrong. The member does it once, with their double beside them, and says what done means as three checks: the letter cites the rule, it has the date, and it has the photo. The double writes down what was done as a flow (section 4), and the three checks become the checks of that flow. The skill is now a small map that anyone can read: its steps, its checks and its limits.

The member shares the flow into the commune. Now each of the six doubles on the street can do it. When one of them runs the flow for its person, the flow's checks decide whether the letter is done, exactly as they did the first time. The skill carries what is needed to check it, which is the thesis of section 2 applied to what a person knows.

Each run of the flow names the flow and its teacher in its receipt. Each time a double runs it and its checks pass, one credit goes back to the teacher, and the teacher sees the count. The street's own sentence appears: "Nobody on our street pays a fine they do not owe." A letter that passes its checks is one step, and the outcome is another, because the town can still refuse an appeal. So the sentence has two claims for each wrong fine: the appeal went out with its three checks passed, and the answer of the town cancels the fine. When both hold for each fine, the goal is proven. An appeal that the town refuses stays open, and the street sees it. Figure 13 draws the skill taught once, the doubles that use it, and the credit that comes back.

Grow the pie by teaching

In many trades, a person who knows a skill guards it, because the skill is scarce. Lev counts teaching as a gift like any other, because more capable people make more work possible for everyone. A teacher's credit grows with each use of what they taught, and a skill, a tool and the history of who trained whom are records in the graph.

What a credit is worth

A credit is a count. One credit says that one gift was used once and that its checks passed, and the receipt of that use names the giver. Every node and every commune counts in this one way, so a credit on one street means what it means on the next.

What a credit is worth is a rule of each commune. A commune sets it as a project sets any other rule in Lev: in its own settings, which hold for its members and for nobody else. The street of this section uses no currency. There a credit is thanks that the street can see, and it buys nothing. Another commune puts a currency on top of its credits, its own or a cryptocurrency that it picks, and gives a credit a price in it. Under the currency the credit stays the same count. So a commune can change its currency, or drop it, and lose no history of who gave what.

The free society

Lev holds eight principles for a society of people and their agents. They are the stance of Lev, and each one shapes a mechanism of this paper.

1. **Do not be creepy.** People give their best data to a system that they trust, and none to one that watches them. So data is shared with consent, for a scope, open to revoke, and readable by the person it is about, and the person controls what is shown and to whom.
2. **Work at every level.** The same system works for a community that is falling apart, for an ordinary business and for a society of advanced agents. So a node works offline, runs on a small machine, and joins larger networks when they are there.
3. **A voluntary civilization.** Groups form by choice round shared words, principles and commitments, with no central control. Membership can be ended. Each group keeps its own rules, its own ledger and, when it wants one, its own currency. Groups meet through the mesh, never through one global social graph.
4. **Hierarchy only by consent.** A leader is legitimate when the members chose to give that leader authority and can take it back. In Lev a delegation is a lease: it has a scope and an end, it can be revoked, and each use of it leaves a receipt. No rank is fixed forever.

5. **Credit that each peer keeps.** A credit counts the same way everywhere, and no central book holds the counts. Each node keeps its own count of its peers, made from receipts that it checked, and the person decides what of that count to show, and to whom. No central score exists.
6. **Agents that represent people.** A double works for its person's interests, learns, shares tools and builds relationships with other doubles, and introduces the people only when the doubles trust each other (section 12).
7. **Grow the pie.** The best help is to train others in your skills, and Lev counts it as a gift.
8. **A group is its shared words.** Every group forms by making its own language: its terms and what they mean. Lev keeps the words of each group in its rules and translates at the borders, so that two communes work together and each keeps its own words.

On the street, these principles are mechanisms. The goal is one sentence with its checks. Each member gives what they can, and each gift is counted for its giver by receipts that each node checks. The goal of the group is proven by its own checks, in the same way as the work of one agent in part I.

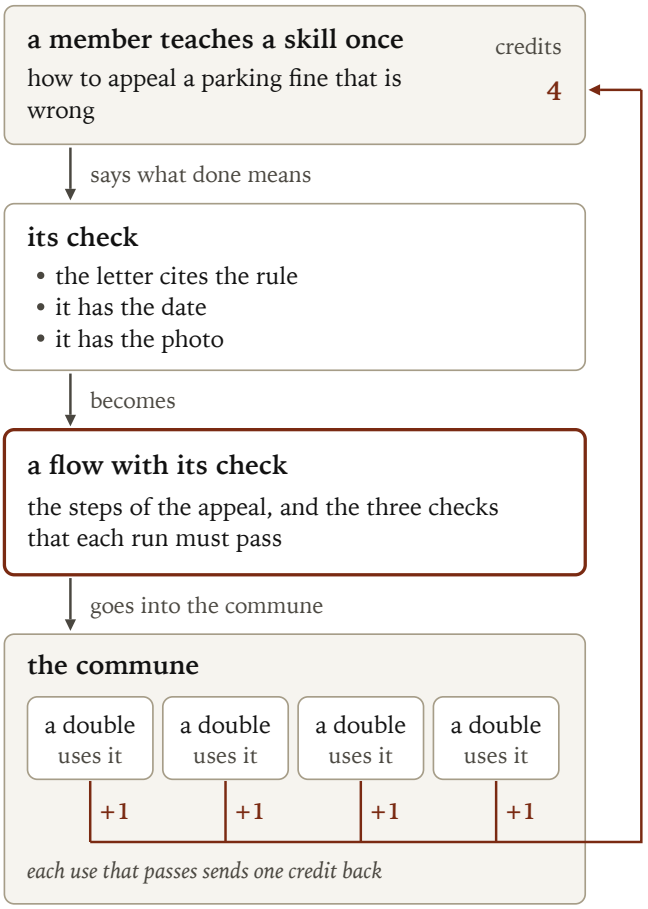


Figure 13. A skill taught once, used by many. A member teaches how to appeal a wrong parking fine once, and says what done means as three checks. The skill becomes a flow with its check and goes into the commune, where each double can use it. Each use that passes its checks sends one credit back to the count beside the teacher.

15. Threats, and how Lev answers each

A system like Lev meets threats of five groups. For each threat, the text says how Lev answers it, and the section that holds the mechanism stands in parentheses.

A false done

- A builder says done with no proof. The claims come before the work, and each has its proof. The evaluator gives no pass on evidence that it cannot read, and could not check is never a pass (section 3). Software supply chains face the same threat, and in-toto answers it with evidence that each step records and that a consumer checks against a signed layout [10].
- A check cannot fail. Each check has a control that it must fail, and a check that exits 1 fails, whatever it prints (section 3).
- A loop never ends. Each flow stops at its limits on attempts, failures in a row, time and commands (section 4).
- A reader agrees with the builder. The reader took no part in the work, gives the path and the line of its evidence, and writes in a file that the builder cannot edit (section 3).

A request or a record that comes again, changed or broken

- The same request comes twice. The same source gives one run, and a write with a key that the store saw before is applied once (section 5).
- A request is changed on its way. An approval gives no authority to a changed request (section 8), and a destination refuses an envelope whose input digest, operation, key against repeats or time budget changed (section 10).
- A body is made to break a surface. The HTTP surface refuses a body that is malformed, is not an object or is too large, and runs nothing (section 6).
- A ledger file is torn or newer. Lev loads each whole record before a torn last line, and an older program refuses to write a newer ledger (section 5).

Authority that is stolen, stale or too wide

- A grant outlives its purpose. An approval gives nothing at its deadline, a restart does not renew it, and a revoked approval stays revoked (section 8).
- A transport key is stolen. A destination moves a sender to a new key, and refuses to move a binding to another principal (section 8).
- The update key of a person's log is stolen. Any update key of the log can sign an entry by itself, and an entry can replace the whole list of keys. Lev answers with keys named in advance: each entry names the next key, and the next key sits on another holder than the key that signs now, so a thief who holds the key that signs now cannot sign the next rotation. KERI binds keys named in advance in its log of key events [21], and The Update Framework uses separate roles with thresholds to limit what a thief of some keys can do [31].
- A site looks like another. For an https address, the scheme, the host and the port determine the origin [17], so a site that only looks like another has another origin, and the extension gives it no tools (section 7).
- A lease is used for another tab or tool. The browser adapter holds a lease for one tab, and under a read lease it refuses a script, a navigation, a read of cookies and a direct browser command (section 7).

A private thing that reaches a caller

This is the threat that the keeper is for (section 8). The keeper checks the lease when it runs the operation, and not when the request arrives, which is the principle of checking each access where it is used [18]. The caller gets a result of a stated type and never the private thing, and the private thing leaves its keeper only as a sealed copy that the owner approves for a named keeper. A lease for one thing that asks for another thing on the same keeper is refused, like any other mismatch.

The keeper names what it does not stop: a person with administrator rights on its machine can read its key store, and a caller sees errors and time as well as results. Lev answers the second with a review of each operation, with operations that take the same time for each answer, and with a limit of tries.

Threats that people bring

- A double oversteps. Each room has its setting, a double cannot widen its own grants, it never pays alone, and a wall stands between the rooms (section 12).
- A system becomes creepy. Data is shared with consent, for a scope, open to revoke, and readable by the person it is about (section 14).
- A member lies about what they gave. A gift counts only as a receipt, and each node checks each receipt by its own rules. A signed claim from another node is only a claim at the next node (section 10).
- A machine reads what it holds for others. A spare machine keeps sealed copies that it cannot open, and a double on another member's machine uses a private thing only through its owner's keeper (section 13).
- A leader keeps power. Authority given to a leader is a lease, with a scope and an end, and the members can revoke it (section 14).
- One score rules everyone. No central score exists. Each node keeps its own count of its peers, and each commune keeps its own rules (section 14).

16. Open questions

This work leaves questions of research open. Some are about proof itself, and some are about the mesh and the groups that use it.

Questions about proof

Two readers do not always give one answer. Two fresh readers can read the same claim against the same evidence and disagree. How many readers, and of which types, make a claim settled? When should a third reader break a tie, and when should the claim itself be written again?

How fresh is fresh? A reader that took no part in the work may still share a model, a prompt or a habit with the builder, and with them the same blind spots. What makes a reader independent enough for a claim of a given cost?

A passing test may not ask enough. A check is as strong as its test. Mutation testing changes small parts of the code and sees whether the tests notice [5]. A survey of that field compares how the changes are made,

and how it handles a change that alters nothing a test can observe [32]. Which faults should Lev plant for each type of claim, and how many are enough?

A proof belongs to the bytes it ran on. When the code moves, a proof can go stale. Who runs it again, when, and who pays for the run?

Questions about the mesh and its groups

State with many writers. Lev keeps one writer for each row. Some state may need more than one writer: a list that a whole commune edits, for example. Replicated data types converge with no coordination when their merge rules meet stated conditions [26]. Which state needs many writers, which merge it takes, and where its rules run, are open. So is the form of those rules: a program in WebAssembly, as Freenet writes its contracts [22], or a Lev record type with its checks.

What a caller learns from a keeper. A caller sees results, errors and time. Lev reviews each operation of a keeper one at a time. A general method that shows an operation leaks nothing through these is open.

A key for a group. A commune whose members join and leave needs a key that follows them. MLS updates the shared keys of a group as members join or leave [33]. How a commune holds such a key, and who may add a member to it, is open.

A page that a host serves. A key in a browser cannot hide data from the node that serves the code of the page. How a person can use a page served by another node without trusting that node is open.

The first read. A reader keeps the highest generation of a shared thing that it checked and refuses a lower one. A reader that has never seen a thing has no such floor. How a first reader can tell a current head from an old one is open.

How to count gifts of different types. A seat in a car, an hour of a machine and an hour of a double's time are different gifts. Each commune decides by its own rules how they compare, and whether a currency stands on its credits (section 14). Which rules stay fair as a commune grows is open. So is how two communes settle work that crosses their border when one gives its credits a price and the other does not. So is what stops a member from earning credits with runs that helped nobody.

17. Related work

Lev builds on earlier work. For each family of work below, the text says what the family gives and where Lev differs.

Proof that travels with the work

Proof-carrying code ships a proof with the code, and the consumer checks it against its own safety policy before it runs the code [2]. Work on the supply chain does the same for software as it is made. In-toto records evidence for the steps of a build against a signed layout [10]. Sigstore lets a user check a signature and the identity that made it against a public log [34]. The Update Framework limits what a thief of some update keys can do [31]. Reproducible builds take byte-for-byte equality as their test [35]. In Lev the thing that is checked

is a claim about a piece of work, and its proof can be the answer of a fresh reader, which the evaluator turns into pass, fail or could not check.

Logs that anyone can check

Certificate Transparency lets a monitor check that a certificate is in a public log and that a later state of the log extends an earlier one [36]. did:webvh gives an identifier on the web a history of updates that a client can verify [19], and KERI records who controls an identifier in a signed sequence of key events, with commitments to the keys of later rotations [21]. Lev uses such a log for a person: the first entry of a signed log fixes the id of the person and names the method did:webvh 1.0, and a later entry can change the update key while the id stays the same.

Authority

Capability systems can express delegation, confinement and revocation when their design provides for it [37]. Macaroons let a holder narrow a grant with caveats that the receiving service checks [38], and UCAN lets a recipient verify a signed chain of delegation [39]. Zanzibar decides access from relations between objects [40]. OCapN drafts how references that carry capabilities move between processes [41]. In Lev, a node saves an approval for one exact request and compares each call with it, and a keeper checks a lease at each use (section 8).

Data that its owner holds

Local-first software keeps the working copy on the device of the user and still lets people work together [42]. Replicated data types converge with no coordination when their merge rules meet stated conditions [26], and Automerge syncs the changes of a document and keeps a compact history of it [43]. In Solid the person chooses where the store is, and access goes by identity [44]; in remoteStorage the app and the store are separate, and any app works with any store under a scoped permission [45]. In both, an app with access reads the data; in Lev a caller gets the result of an operation on a private thing and never the thing. Range-based set reconciliation finds the entries that two peers lack by comparing fingerprints of ranges [46]. Lev gives each row one writer, and a copy catches up from a cursor (section 10).

Networks with no centre

The first design of Freenet pools the storage of peers and adapts its routing from experience, to fetch and copy files while it aims to protect the anonymity of those who publish and read [29]. Its current design describes contracts that say how a piece of state merges, delegates that manage private state, and routing between peers that adapts [22]. Kademlia routes a lookup by the XOR distance between ids [47], and work on small worlds shows when local routing finds the short paths that exist [30]. Secure Scuttlebutt copies a signed log for each author [48]. Nostr clients sign events and publish them through relays [49]. The AT Protocol keeps the signed repositories of accounts apart from hosting, relays and indexes [50]. Matrix copies the history of a room between servers, with rules for changes of state that compete [51]. Holochain has peers check shared entries against the rules of the application [27]. In Lev, each node decides the requests that come to it, and a shared thing carries its rules (section 10). Section 11 compares Freenet and Lev.

Protocols that Lev speaks

The Model Context Protocol lets a client find and use the tools, resources and prompts of a server [12]. OpenAPI describes the paths, inputs and responses of an HTTP API [11], gRPC maps calls onto the streams of HTTP/2 [13], and WebSocket lets either end send messages after a handshake [14]. Lev makes each of these surfaces from the declaration of an action (section 6). TLS 1.3 [52] and QUIC [53] protect a transport, and MLS protects the keys of a group [33].

Testing a test

Mutation testing changes code in small ways and asks whether the tests notice [5]. QuickCheck tries generated inputs against a stated property [54]. FoundationDB runs its code in a deterministic simulator with planted faults, so that a failure replays from its seed [55]. Lev gives each check a control that it must fail (section 3).

Writing and architecture

ASD-STE100 sets a controlled vocabulary and rules of writing to make technical instructions easier to read [56]. The C4 model draws a system at the levels of its context, its running units, its components and its code [24]. Hexagonal architecture puts the application behind ports, with an adapter for each technology [15]. Lev takes it as its primary pattern.

18. What the person holds

Section 1 began with a report that says the work is done, and a person who cannot tell whether it is. With Lev that person holds more than a report. They hold the sentence that they asked for and the claims under it. For each claim they hold the answer of a check that the builder did not control: pass, fail or could not check. They hold a receipt that another person can check again. When the work stopped at a limit, the receipt says so, and no part of it claims a pass.

The same rule holds on more than one machine. A node takes no other node at its word, and a commune counts a gift only from its receipt. A double that acts for a person comes home with a record of what it did, what it asked and what it refused.

The request was: make this outcome true, and prove it. The proof is what lets another person rely on the work with no need to trust the worker.

ACKNOWLEDGMENT

We thank Joshua Eisenhart for the principles of a voluntary society of people and their agents on which part III builds.

REFERENCES

[1] *Reflections on Trusting Trust*. Ken Thompson. Communications of the ACM. 1984. https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf

[2] *Safe Kernel Extensions Without Run-Time Checking*. George C. Necula, Peter Lee. USENIX Symposium on Operating Systems Design and Implementation. 1996. https://www.usenix.org/conference/osdi-96/safe-kernel-extensions-without-run-time-checking

[3] *in-toto Attestation Framework Spec*. in-toto Project. in-toto Project. 2024. https://github.com/in-toto/attestation/tree/v1.1.0/spec

[4] *An Architecture for Trustworthy and Transparent Digital Supply Chains*. Henk Birkholz, Antoine Delignat-Lavaud, Cédric Fournet, Yogesh Deshpande, Steve Lasker. RFC Editor, RFC 9943. 2026. https://www.rfc-editor.org/info/rfc9943/

[5] *Hints on Test Data Selection: Help for the Practicing Programmer*. Richard A. DeMillo, Richard J. Lipton, Frederick G. Sayward. IEEE Computer. 1978. https://web.archive.org/web/20250621131639id_/https://gse.ufsc.br/bezerra/disciplinas/Confiabilidade/docs/demillo-mutants.pdf

[6] *Amazon States Language*. Amazon Web Services. Amazon States Language Specification. 2024. https://states-language.net/spec.html

[7] *Common Workflow Language (CWL) Workflow Description, v1.2.1*. Peter Amstutz, Michael R. Crusoe, Kaushik Ghose, John Chilton, Michael Franklin, Bogdan Gavrilovic, Stian Soiland-Reyes. Common Workflow Language Project. 2024. https://www.commonwl.org/v1.2/Workflow.html

[8] *The Chubby lock service for loosely-coupled distributed systems*. Mike Burrows. USENIX Symposium on Operating Systems Design and Implementation. 2006. https://research.google/pubs/the-chubby-lock-service-for-loosely-coupled-distributed-systems/

[9] *HTTP Semantics*. Roy T. Fielding, Mark Nottingham, Julian Reschke. RFC Editor, RFC 9110. 2022. https://www.rfc-editor.org/rfc/rfc9110.html

[10] *in-toto: Providing farm-to-table guarantees for bits and bytes*. Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, Justin Cappos. USENIX Security Symposium. 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[11] *OpenAPI Specification v3.1.1*. OpenAPI Initiative. OpenAPI Initiative. 2024. https://spec.openapis.org/oas/v3.1.1.html

[12] *Specification - Model Context Protocol*. Model Context Protocol Contributors. Model Context Protocol, version 2025-06-18. 2025. https://modelcontextprotocol.io/specification/2025-06-18

[13] *gRPC over HTTP2*. gRPC Authors. gRPC Project. 2016. https://github.com/grpc/grpc/blob/v1.0.0/doc/PROTOCOL-HTTP2.md

[14] *The WebSocket Protocol*. Ian Fette, Alexey Melnikov. RFC Editor, RFC 6455. 2011. https://www.rfc-editor.org/info/rfc6455/

[15] *Hexagonal Architecture*. Alistair Cockburn. Alistair Cockburn's architecture articles. 2005. https://alistair.cockburn.us/hexagonal-architecture

[16] *WebMCP*. Web Machine Learning Community Group. W3C Web Machine Learning Community Group Draft Report. 2026. https://webmachinelearning.github.io/webmcp/

[17] *The Web Origin Concept*. Adam Barth. RFC Editor, RFC 6454. 2011. https://www.rfc-editor.org/info/rfc6454/

[18] *The Protection of Information in Computer Systems*. Jerome H. Saltzer, Michael D. Schroeder. Proceedings of the IEEE. 1975. https://web.mit.edu/Saltzer/www/publications/protection/

[19] *The did:webvh DID Method*. Stephen Curran, John Jordan, Andrew Whitehead, Brian Richter, Michel Sahli, Martina Kolpondinos. Decentralized Identity Foundation. 2025. https://identity.foundation/didwebvh/v1.0/

[20] *Decentralized Identifiers (DIDs) v1.0*. Manu Sporny, Amy Guy, Markus Sabadello, Drummond Reed. W3C Recommendation. 2022. https://www.w3.org/TR/2022/REC-did-core-20220719/

[21] *Key Event Receipt Infrastructure (KERI)*. Samuel M. Smith. arXiv preprint. 2019. https://arxiv.org/abs/1907.02143

[22] *Freenet: A Peer-to-Peer Platform for Real-Time Decentralized Applications*. Ian Clarke. Freenet Project. 2026. https://freenet.org/pdf/freenet-whitepaper.pdf

[23] *Architectural Blueprints—The "4+1" View Model of Software Architecture*. Philippe Kruchten. IEEE Software. 1995. https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

[24] *The C4 Model for Software Architecture*. Simon Brown. InfoQ. 2018. https://www.infoq.com/articles/C4-architecture-model/

[25] *In Search of an Understandable Consensus Algorithm*. Diego Ongaro, John Ousterhout. USENIX Annual Technical Conference. 2014. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[26] *A comprehensive study of Convergent and Commutative Replicated Data Types*. Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. INRIA Research Report RR-7506. 2011. https://perso.lip6.fr/Marc.Shapiro/papers/2011/Comprehensive-CRDTs-RR7506-2011-01.pdf

[27] *Holochain: Distributed Coordination by Scaled Consent, not Global Consensus*. Eric Harris-Braun, Arthur Brock, Paul d'Aoust. Holochain white paper v2.0. 2024. https://www.holochain.org/documents/holochain-white-paper-2.0.pdf

[28] *IPFS - Content Addressed, Versioned, P2P File System*. Juan Benet. arXiv preprint. 2014. https://arxiv.org/abs/1407.3561

[29] *Freenet: A Distributed Anonymous Information Storage and Retrieval System*. Ian Clarke, Oskar Sandberg, Brandon Wiley, Theodore W. Hong. Designing Privacy Enhancing Technologies, proceedings of the 2000 workshop. 2001. https://www.cs.cornell.edu/people/egs/615/freenet.pdf

[30] *The Small-World Phenomenon: An Algorithmic Perspective*. Jon M. Kleinberg. ACM Symposium on Theory of Computing. 2000. https://www.stat.berkeley.edu/~aldous/Networks/swn-1.pdf

[31] *Survivable Key Compromise in Software Update Systems*. Justin Samuel, Nick Mathewson, Justin Cappos, Roger Dingledine. ACM Conference on Computer and Communications Security. 2010. https://theupdateframework.io/papers/survivable-key-compromise-ccs2010.pdf

[32] *An Analysis and Survey of the Development of Mutation Testing*. Yue Jia, Mark Harman. IEEE Transactions on Software Engineering. 2011. https://web.archive.org/web/20250517065032id_/https://web.eecs.umich.edu/~weimerw/2024-481F/readings/mutation-testing.pdf

[33] *The Messaging Layer Security (MLS) Protocol*. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, Katriel Cohn-Gordon. RFC Editor, RFC 9420. 2023. https://www.rfc-editor.org/rfc/rfc9420.html

[34] *Sigstore Announces General Availability at SigstoreCon*. Open Source Security Foundation. Open Source Security Foundation. 2022. https://openssf.org/press-release/2022/10/25/sigstore-announces-general-availability-at-sigstorecon/

[35] *Collaborative Working Sessions - Defining our definitions*. Reproducible Builds Project. Reproducible Builds Hamburg Summit. 2023. https://reproducible-builds.org/events/hamburg2023/definitions/

[36] *Certificate Transparency Version 2.0*. Ben Laurie, Eran Messeri, Rob Stradling. RFC Editor, RFC 9162. 2021. https://www.rfc-editor.org/rfc/rfc9162.html

[37] *Capability Myths Demolished*. Mark S. Miller, Ka-Ping Yee, Jonathan Shapiro. Johns Hopkins University Systems Research Laboratory Technical Report SRL2003-02. 2003. https://cgi.cse.unsw.edu.au/~cs9242/papers/Miller_YS_03.pdf

[38] *Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud*. Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, Mark Lentczner. Network and Distributed System Security Symposium. 2014. https://research.google/pubs/macaroons-cookies-with-contextual-caveats-for-decentralized-authorization-in-the-cloud/

[39] *User Controlled Authorization Network (UCAN) Specification*. Irakli Gozalishvili, Daniel Holmgren, Philipp Krüger, Brooklyn Zelenka. UCAN Working Group. 2026. https://github.com/ucan-wg/spec

[40] *Zanzibar: Google's Consistent, Global Authorization System*. Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, Mengzhi Wang. USENIX Annual Technical Conference. 2019. https://www.usenix.org/conference/atc19/presentation/pang

[41] *CapTP Specification*. Object Capability Network Pre-standardization Group. Object Capability Network Pre-standardization Group. 2026. https://github.com/ocapn/ocapn/blob/main/draft-specifications/CapTP%20Specification.md

[42] *Local-First Software: You Own Your Data, in spite of the Cloud*. Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, Mark McGranaghan. ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 2019. https://martin.kleppmann.com/papers/local-first.pdf

[43] *Introducing Automerge 2.0*. Automerge Project. Automerge Project. 2023. https://automerge.org/blog/automerge-2/

[44] *Solid Protocol*. Sarven Capadisli, Tim Berners-Lee, Kjetil Kjernsmo. W3C Solid Community Group, Editor's Draft. 2024. https://solid.github.io/specification/protocol

[45] *remoteStorage*. Michiel B. de Jong, Franciscus Kooman, Sebastian Kippe. Independent Internet-Draft, IETF Datatracker. 2023. https://datatracker.ietf.org/doc/html/draft-dejong-remotestorage-22

[46] *Range-Based Set Reconciliation*. Aljoscha Meyer. arXiv preprint. 2022. https://arxiv.org/abs/2212.13567

[47] *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. Petar Maymounkov, David Mazières. International Workshop on Peer-to-Peer Systems. 2002. https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia.pdf

[48] *Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications*. Dominic Tarr, Erick Lavoie, Aljoscha Meyer, Christian Tschudin. ACM Conference on Information-Centric Networking. 2019. https://conferences.sigcomm.org/acm-icn/2019/proceedings/icn19-19.pdf

[49] *NIP-01: Basic protocol flow description*. Nostr Protocol Contributors. Nostr Implementation Possibilities. 2026. https://raw.githubusercontent.com/nostr-protocol/nips/master/01.md

[50] *Bluesky and the AT Protocol: Usable Decentralized Social Media*. Martin Kleppmann, Paul Frazee, Jake Gold, Jay Graber, Daniel Holmgren, Devin Ivy, Jeromy Johnson, Bryan Newbold, Jaz Volpert. arXiv preprint. 2024. https://arxiv.org/abs/2402.03239

[51] *Matrix Specification v1.13*. Matrix.org Foundation. Matrix.org Foundation. 2024. https://spec.matrix.org/v1.13/

[52] *The Transport Layer Security (TLS) Protocol Version 1.3*. Eric Rescorla. RFC Editor, RFC 8446. 2018. https://www.rfc-editor.org/info/rfc8446/

[53] *QUIC: A UDP-Based Multiplexed and Secure Transport*. Jana Iyengar, Martin Thomson. RFC Editor, RFC 9000. 2021. https://www.rfc-editor.org/rfc/rfc9000.html

[54] *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. Koen Claessen, John Hughes. ACM SIGPLAN International Conference on Functional Programming. 2000. https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf

[55] *FoundationDB: A Distributed Unbundled Transactional Key Value Store*. Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, Vishesh Yadav. ACM SIGMOD International Conference on Management of Data. 2021. https://www.foundationdb.org/files/fdb-paper.pdf

[56] *About ASD-STE100*. Aerospace, Security and Defence Industries Association of Europe. ASD Simplified Technical English Maintenance Group (the specification, Issue 9, is given out on request). 2025. https://asd-ste100.org/about.html

---