FUTURE OF RADICLE: TRANSPARENT MACHINES

JAMES HAYDON

Abstract. This document presents a vision for a possible future of Radicle. The main goal is to transform Radicle into a technology that powers an ecosystem of *transparent state machines*. This is a network of replicated state machines running the Radicle interpreter which are monitored by their peers, to ensure they abide by the Radicle semantics.

Contents

1. Current state of Radicle

Currently Radicle is composed of two main parts:

- A deterministic programming language that is designed in order to be able to specify state machines whose semantics may evolve over time.
- A daemon which allows hosting and reading such machines on IPFS, using:
  - The IPFS DAG to store a linked list of blocks containing the Radicle expressions which determine the machine,
  - IPNS as a mutable reference to the head of the list,
  - IPFS Pubsub for communication of suggested inputs from peers to the owner of a machine.

This setup has a number of drawbacks.

- *Unweildy reprogrammability.* The current design of the language has not proved itself to allow for easy reprogrammability of machines.
- *No guarantee of Radicle semantics.* The machine owner has complete control over what the IPNS link points to. In particular there are no guarantees that the owner will update the pointer in such a way that, over time, the sequence

†Monadic, james@monadic.xyz.

of Radicle expressions respects the *append-only* property. In particular the machine owner can rewrite history, change semantics, remove and include new data, etc., all without detection by other peers. In short there is no guarantee the view of the machine evolves according to the rules of a Radicle state machine, as defined in the Radicle whitepaper [1]. This voids a lot of the purported benefits of Radicle ("rule-based governance").

- Because all mutations to the state of the machine are made visible through updates to the IPNS link, updates are associated with a single cryptographic key. In other words, Radicle machines are *single-writer*. This is partially mitigated by the mechanism of suggesting inputs via IPFS Pubsub, but this still leaves two problems:
  - *Owner availability burden:* The owner of the machine must be online at the same time as the request is sent.
  - *Isolation:* Each machine is its own little universe. Even fundamental things like establishing pairings between usernames and public keys must be restarted from scratch on each machine.
  - *Potential censorship:* The owner has the opportunity to reject certain inputs if this might benefit them.

This document aims to tackle these problems by proposing:

- A P2P system for *monitoring* Radicle machines which, while giving no guarantee that owners will follow the Radicle semantics, makes the detection of (some forms of) misbehaviour evident at the protocol level. Peers simply ignore misbehaving machines. This also indirectly alleviates the owner availability burden by making third-party, highly-available hosting services a viable option, since they can be monitored by the original owner.
- A system for inter-machine communication, backed by proofs of inclusion of outputs or state on other machines.
- A simpler mechanism for restricting inputs at the language level, based on an updatable transactor function, rather than eval-redefinition, along with a non-hyper static environment to allow for better reprogrammability.
- Finally the potential censorship problem is partially tackled by making censorship as detectable as possible by clients, so that users may then gossip (off-protocol) about misbehaving machines and stop using them. We also discuss a web-of-trust type solution which may be implemented at a later stage.
- New multi-writer Radicle reference types, backed by CRDTs, which mitigate the owner-availability burden.
- Finally we discuss how we can extend the confidence in the transparency of a system to the UI presented to end-users through the use of *gateways*.

2. Transparent machines

2.1. **Motivation.** The main goal of this proposal is to fix the most important problem highlighted above: currently machines can misbehave in serious ways, and not respect the Radicle semantics at all. By creating an ecosystem of machines and monitors, allowing for the detection and broadcasting of misbehaviour, we aim to make machines *transparent*: owners may still misbehave but it will be obvious to everyone else. Machines are hosted by people/organisations which want to expose transparency in the ways they operate, so they have an incentive to stick to the rules.

The primary application for Radicle will still be the hosting/management/governing of important open source projects, but other applications we have in mind are:

- Certificate authorities, Name registries, etc.
- Package managers,
- Adding transparency to the governance systems used by e.g. political organisations,
- Adding transparency to the interactions between organisations (e.g. NGOs, private corporations, etc.) end-users.

Essentially any organisation that wants to make transparent to the world one of the systems it uses to operate should be able to use Radicle to specify and host a machine to realise this goal. Even governments have been exploring ways to increase transparency in their operations in an effort to combat corruption and increase the faith their citizens have in them (e.g. [2]), so there could be wide uses and demand for such an ecosystem.

There are already several distributed systems in production in which certain entities publish secure logs, so that third parties may monitor their activities and check they are behaving according to rules everyone has agreed to. For example, Certificate Transparency is:

> an Internet security standard and open source framework for monitoring and auditing digital certificates. The standard creates a system of public logs that seek to eventually record all certificates issued by publicly trusted certificate authorities, allowing efficient identification of mistakenly or maliciously issued certificates.

Each such distributed system must define afresh the protocol and set of rules that are to be followed. The *Radicle network* proposes a single unified protocol and monitoring system for all such "transparent machines". Such a unification would have several benefits:

- *Protocol re-use:* Having many systems use the same underlying protocol makes that protocol safer, more resources can be spent in auditing its code, fixing vulnerabilities, etc.
- *Code re-use:* domain-logic specifying rules is likely to be similar across domains and organisations. For example many domains will require processes for holding votes, electing leaders, managing user permissions, etc. By creating a set of ready-to-use packages each new organisation may launch a tailored transparent machine with minimal effort.
- *Larger, homogeneous network of monitors:* Once a machine is launched it immediately benefits from a large network on monitors. Different domains can now share monitor nodes, because each monitor just needs to understand the base Radicle semantics, not specific domain requirements.

The ideas presented in this document take inspiration from:

- This insightful blogpost: Secure ledgers don't require proof-of-work.
- *Transparent Certificate* in general, including:
  - How it works.
  - The latest RFC.
  - Some of the docs of the Trillian repository.
  - A paper: Aggregation-Based Gossip for Certificate Transparency.

It might help to review these documents to better understand this one.

The implementation may use pre-existing P2P technologies, e.g. the secure ledger might use an IPLD [7] linked list (as Radicle currently does), a DAT hypercore feed [5] or a Trillian log [6]. We base the following on a hash-linked list for simplicity, but note that Trillian in particular offers a similar API with respect to fetching of data and proving inclusion/forks, but uses a *dense Merkle tree* instead, making

many operations much more efficient. The theoretical properties are essentially the same though.

2.2. **Protocol.** Preliminary definitions:

- *Radicle-lang* refers to the Radicle programming language [1]: a syntax of valid expressions and an deterministic *interpreter* which maintains state (or produces an error) when ran over a sequence of expressions.
- A (valid) *Radicle sequence* is a sequence of Radicle-lang expressions which does not throw an error when evaluated in order by the Radicle-lang interpreter.
- A *secure ledger* (also called a "blockchain", but without the implied proof-of-work consensus protocol) is a list of records, called *blocks*, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, and transaction data. In our case the transaction data will be a list of Radicle expressions. Each block is signed by the same private key, whose corresponding public p key identifies the ledger as a whole. We will simply say *ledger* to mean secure ledger in this document, and *ledger for* p when we wish to specify the corresponding public key.
- A ledger x is a *prefix* of another y, written $x \leq y$, if y is obtainable from x by appending zero or more blocks to x. In particular this implies that x and y both use the same public key. This forms a partial order on the set of ledgers.
- Two secure ledger are *consistent* if one is a prefix of the other, that is, if they are comparable for $\leq$.

2.2.1. *Radicle ledgers.* The point of the ecosystem is to maintain Radicle state machines which are determined by *Radicle ledgers*. A block of a Radicle ledger for p is a record with the following fields:

- `expressions`: a hash for the list of Radicle expressions,
- `outputs`: a Merkle tree hash for the list of outputs for all expressions of the machine from the start. This allows for compact proofs that an output was seen on a machine, and is used in §2.4.
- `previousHash`: hash of the previous block, or $\perp$,
- `hash`: hash of this block,
- `signature`: a signature for the whole record (minus the signature field), for public key p.

Note that in order to keep the contents of the blocks lightweight, the expressions are not present directly, only a hash of them. Replication of the actual expressions is left to another layer of the Radicle network. For example the hash could be an IPFS CID, and the lists of expressions replicated through IPFS. In the following we will assume that a hash uniquely determines a list of Radicle expressions.

The set R_p of Radicle ledgers for the public key p is the smallest subset of the set of non-empty lists of blocks such that:

- The ledger with a single block a, singed for p, with $a_{\texttt{previousHash}} = \perp$ and $a_{\texttt{expressions}} = []$ is a member of R_p.
- If $x \in R_p$ with last block a, and b is a block such that:
  - b is signed for p,
  - $b_{\texttt{previousHash}} = a_{\texttt{hash}}$,
  - Appending the list of expressions corresponding to $b_{\texttt{expressions}}$, to the concatenation of all the lists of expressions corresponding to the hashes in x produces a valid Radicle sequence, and furthermore, $b_{\texttt{outputs}}$ is the root of the Merkle tree of all the outputs obtained by running through this list of expressions,

then appending b to x produces a member of R_p.

2.2.2. *Machines.* The *Radicle network* is composed of two sorts of nodes: *machines* and *monitors*.

A *Radicle machine* (or simply *machine*), is a node (identified by a public key) which maintains a Radicle ledger. The machine responds to requests for inclusion of expressions in the ledger, and also makes the ledger accessible for monitoring by monitor nodes. To be considered *valid* a machine must:

(M_1) Respect the *append-only* property:
- All the ledgers disseminated by the machine are consistent with one another.
- If ledger s_0 is made available before ledger s_1, then s_0 must be a prefix of s_1.

(M_2) Respect Radicle semantics: a ledger is invalid if the underlying sequence of expressions throws an error, or the output hashes do not match the `outputHash`'s contained in the blocks. In general all ledgers disseminated must conform to the definition of a Radicle ledger given above.

(M_3) Not censor: The machine should respond to all valid requests. In particular the machine should respond to requests made by monitors for the ledger, and try to include any valid expressions that are sent to it by other peers.

Property (M_3) is hard to state precisely for various reasons that are discussed below.

A ledger that outputs a ledger that doesn't respect (M_2) is *corrupt*. Note that because ledgers are meant to start with the same dummy block, any two non-corrupt ledgers for the same key must have a common prefix.

2.2.3. *Monitors.* Monitor nodes observe the ledgers output by certain machines and gossip between themselves in order to detect misbehaviour.

A machine might misbehave in one of the following ways:

(1) Fork the ledger, so as to present different views to different clients, or in order to re-write history.
(2) Output an invalid ledger, for example to be able to make invalid claims on other machines.
(3) Refuse to respond to monitoring requests, in order to cover up some other misbehaviour.
(4) Refuse to respond to expression inclusion requests (censoring).

In the case of (1) or (2), it will disseminate a proof of the misbehaviour as widely as possible. Monitors counter-act (3) by gossipping amongst themselves, so that inconsistencies in the responses from the machine can be detected. Monitors also have to be careful about unmonitoring and remonitoring machines. For example if a machine can't parse a log output by a machine it might decide to stop monitoring it, but then pick up monitoring again a few weeks later after the machine starts outputting valid logs again. But this sort of behaviour presents a way for machines to cheat: should they wish to re-write history they may start outputting slightly corrupt logs in the hope that most monitors will stop monitoring, only to start again once history has been re-written. For this reason monitors should treat even mild corruptions of the log quite seriously, and when unmonitoring a machine should persist the machine's ID so as to make sure to never start monitoring it again.

Provable misbehaviours of a machine:

- A machine has *forked* if it has disseminated two inconsistent ledgers. A *fork-proof* consists of two blocks b_0 and b_1 (signed by the machine's key) which contain identical hashes for a previous block b. Because of the assumed

properties of cryptographic hashes and signatures, it is a assumed that the existence of a fork-proof proves that a machine has forked.
- A machine is *invalid* if it has disseminated a ledger whose contents is not a valid Radicle sequence. In this case a (minimal) *invalidity-proof* is a block b such that the previous block is still valid.

Taken together, fork-proofs and invalidity-proofs form the set of *misbehaviour-proofs*.

Some other behaviours can not be proven to other nodes:

- A monitor may detect that a machine is not desseminating the ledgers monotonically. For example it might encounter a ledger x_0 via gossiping only after this make a request for the ledger directly from the machine, for which it responds x_1. If $x_1 < x_0$ then the machine has not respected $(M1)$.
- A monitor may detect that a machine is selectively applying some expressions and not others. See §2.3 on censorship detection.

In cases that a monitor has detected unprovable misbheaviour, it creates a *misbehaviour-description*, including for example signed Pubsub messages, HTTP responses, etc., and it should make this known to other monitors and stop monitoring the machine, and never restart monitoring it.

More formally, to be a valid monitor, a node must:

- Maintain and make available to peers its *dead-set* D; a set of machine IDs paired with either a misbehaviour-proof or a misbehaviour-description. Whenever it connects to another peer it should make this dead-set available to the peer, and ask for that peer's dead-set D'. Once checking the proofs, all elements of the dead-set that are accompanied by misbehaviour proofs should be added to its own dead-set. It is free to add the other elements to its dead-set too, depending on how much it trusts that monitor. A monitor should never remove elements from the dead-set.
- Maintain and make available to peers its *monitor-set* M; a set of IDs of machines it is currently monitoring. It is free to add or remove machines from this set as it sees fit, as long as $M \cap D = \varnothing$.
- For each machine $m \in M$ in its monitor set, to:
  (1) Subscribe to the ledger updates from m.
  (2) Subscribe to ledger updates for m broadcast by other monitors.
  (3) Maintain the current most up to date ledger for m:
    * In normal operation all ledgers it receives from (1) and (2) should be consistent, and hence there is always a unique maximum. This maximum should be broadcast to all other monitors of m.
    * Should it receive a ledger, either through (1) or (2), which is not consistent with the one it is currently maintaining, then the monitor has detected a fork. In that case it should construct the minimal fork-proof and broadcast it to all other monitors of m.
    * Should it receive a corrupt ledger, it should construct a minimal invalidity-proof and broadcast this to all other monitors of m.
  (4) If at any point it constructs a misbehaviour-proof for m, it must remove m from M, and add it to D.
  (5) Detecting non-provable misbehaviours is more involved and optional. Possible techniques are discussed in the next section. If at any point it detects non-provable misbehaviour it should add m to D and broadcast a misbehavior-description to other peers.

2.3. **Censorship detection.** Checking that machines respect the last rule (M_3), no-censoring, is harder since there isn't a proof which can be disseminated among the peers. As a first step, censorship will not be blocked at the protocol level.

Instead we will focus of censorship being as *detectable* as possible to the end-users. Users are then encouraged to gossip amongst themselves (out-of-band), and simply stop using any machines that seem to be censoring inputs, or have done so in the past.

Deciding if an input is being censored can by difficult. Given a ledger, an expression e may be valid as the next input. However the machine might have already decided to include other expressions in the next block, some of which invalidate e. Even if e is insertable at any position in the next block that is formed by the machine, and the machine received the expression e before starting that block, it is unclear that the machine tried to censor e. Indeed it might have received another input which was incompatible with e, and also incompatible with another input it decided to include in the next block, but was otherwise valid. It is even plausible that such inputs may be crafted for this very reason by a malicious actor in an attempt to block a certain input. To counter-act this we make (M_3) more precise in the following way:

(M_3') Machines consider potential expressions for inclusion into the ledger in such a way that an expression is rejected only if there exists some index in the next block at which inserting that expression would form an invalid sequence.

One algorithm that achieves this is the naive greedy one: expressions are considered one at a time in a random order, creating an increasing valid sequence. If appending the next item to the currently valid sequence does not create a valid sequence then it is rejected and never reconsidered.

The disadvantage of this options is that checking censorship is quite expensive: one must try to insert an input at each location of a certain number of blocks, so possibly this check would only kick in at certain monitor nodes when other nodes have started to get many rejected expressions.

In cases where censorship would be particularly bad (e.g. voting on something), specialised Radicle data-structures could be used to make the tracking of potential expressions more transparent. For example a special set could be initialised with a validation function to specify exactly what sort of values are to be accepted. Expressions submitted for inclusion bypass the root validator and go straight to the set, as long as it remains "open". A special command, or another predicate, "seals" the set, at which point it can be processed, e.g. to determine the result of the vote. Such a system removes any ambiguity on why an expression might not be accepted according to the Radicle semantics.

2.4. **Inter-machine communication.** Since Radicle ledgers record a Merkle tree hash of the output list at each block, it is possible to create a compact proof witnessing that an output was seen on some machine. Radicle-lang will be equipped with a new primitive (`assert-on-machine` k o p) which takes two arguments:

- A public key k identifiying a machine,
- An output value o,
- A proof p. This is composed of the signed block in which the output appears, and the Merkle proof for the specific output of that block.

The function returns a Boolean value, indicating if the proof is valid.

A machine can then depend on another by validating such proofs. For example, let's assume a machine R has come to be trusted as a username registration service. R accepts all requests to assign a username u (a string) to a public key k, as long as that username has not already been assigned to another. For each such valid request it will output $[u\ k]$, a vector of length 2. Another machine P, representing an important open source project, may then allow users to use certain usernames

as long as they have been claimed on R. To claim a username on P, a user can include a proof p of the presence of the output $[u\ k]$ on R, and the code defining the validation process on P would make a call (`assert-on-machine` R_{id} $[u\ k]$ p) in order to verify that the user may claim this username.

For cases when the state of a machine is more complex, Radicle can be augmented with primitives for creating and checking authenticated datatypes[1]. A machine can then create an authenticated piece of data representing a part of its domain state, and return the root hash as the output. Other machines can then use this to create proofs stating various properties of this state. Alternatively, an extra field could be added to the blocks for this specific purpose, linked to a new special effectful primitive function, `update-domain-state`, and this would also have to be checked by the monitors.

2.4.1. *Linear time.* Using a Merkle hash of the whole domain state actually has another benefit over referencing outputs directly: one can require that one uses a relatively "fresh" version of the referenced machine's state. For example, a more sophisticated version of R might have various means of revocating usernames for various reasons. The function `assert-on-machine` could optionally also check that the output being referenced is not strictly before any previous output from the same machine that was also referenced. This prevents users using very old inputs as part of their proofs. While the machine P may experience a "lag" in its view of R, time still flows in the sime direction.

2.4.2. *RPC.* It is even possible that a machine offer a sort of RPC functionality, by accepting some special inputs corresponding to specific queries, and outputting the result. Machines can even evaluate quite arbitrary expressions e with the guarantee that state is not modified by using the following pattern[2]:

```
(catch 'rpc-result
  (do (def r e)
      (throw 'rpc-result r))
  (fn [r] [:rpc-query e r]))
```

Since `catch` always restores the entire state as it was before entering the body, the machine's state will not be modified. The output is `[:rpc-query e r]`, which the calling machine can then use as proof that `e` evaluates to `r` on this machine.

3. Radicle-lang simplification/improvements

3.1. **Simpler input restriction.** Eval redefinition is being used to restrict the allowed inputs of a chain. But in fact most of the logic for rejecting inputs is done by validators, at the moment placed at the start of each "command" function.

Futhermore once an eval-redefinition has taken place, this heavily restricts the programming available in the machine (that's the point), but this is annoying for experimentation at the REPL, crafting custom queries, etc. For example we have the idea that users who wanted JSON responses could just query `(to-json (list-patches))` instead of `(list-patches)`, but this is not valid after eval-redefinition.

Here we propose that instead there is a special purpose function which holds the machines root validation function `tx`; a validation function that is used for all inputs to the machine. When an expression e is submitted to the machine, the atom `tx` is evaluated, and it is expected to be a function. If so, this function is invoked on e. Whatever this invocation returns is what's finally evaluated. Furthermore,

[1]The work for this has already been done in an unmerged pull-request.

[2]Of course the machine should add a timeout in this case to prevent expressions which take too long to evaluate.

the Radicle executable can be configured to bypass the root-validator using a simple flag, for when one wants to experiment with the machine at the REPL or craft specialised queries. Daemons which are owner-mode for a machine should obviously never bypass the root validator when accepting new inputs, but the `query` endpoint on a local daemon would just execute expressions normally, without calling `tx`.

Even just for a simple counter-machine, the eval-redefinition spec of the state-machine is not beginner friendly (omitting the prelude):

```
(def i (ref 0))

(def eval
  (fn [input rad-state]
    (match input
      :increment
        (eval (quote (modify-ref! i (fn [x] (+ 1 x)))) rad-state)
      :getCounter
        (list (read-ref i) rad-state))))
```

This is what it would look like with a root-validator:

```
(def i (ref 0))

(def tx
  (fn [expr]
    (if (eq? expr :increment)
      (modify-ref! i (fn [x] (+ 1 x)))
      (throw :invalid-input "Can␣only␣:inc!"))))
```

Note that querying has now been pushed out of the machine code altogether; if one wants to query the machine for the current value one can just use the expression `(read-ref! i)`.

The function `tx` is expected to perform most state-updates itself, and return a value which evaluates to itself, a simple log of what sort of transaction was processed. Only when new *code* is to be added to the machine, for example redefinition of `tx` itself, would the function `tx` return that quoted code, to then be executed on the machine. One of the advantages of this is that it removes the need for the `eval` function altogether, which makes the semantics and static-analysis of Radicle much easier. However without eval-redefinition, it might be necessary to add macro-definition functionality.

Such validation functions can be layered, higher-order, etc., and this makes for more flexible code-reuse between machines. In particular, because they can be stateful, this allows for the creation of interesting "middleware", for example for conducting votes, authenticating users, moderating comments, etc.

For example if the participants of a machine decide all inputs should be signed by some public key, then they could import a standard higher-order validation function, designed as follows. The function (`signed-input` f) expects all inputs to be of the form:

```
[:signed-input
  {:nonce "123"
   :machine-id "abc"
   :public-key pk
   :signature s
   :input i}]
```

It would check that the signature `s` is indeed a signature for the rest of the dict, and then pass

```
{:input i
 :author pk}
```

to the sub-validator f.

Higher-order validator can also be stateful. For example, a validator can manage a vote for some binary decision. It would look for specific sorts of inputs, *proposal inputs*, and when it encounters one, start a vote on that proposal (other sorts of inputs are simply passed through to some sub-handler). At that point it then also listens for *vote inputs* for that specific proposal, maintaining in state how much votes each outcome has accumulated. When it successfully processes a vote, it simply returns the metadata:

```
[:vote-accepted
  {:vote-for "123"
   :user "user-id"}]
```

When enough votes have come through for there to be a winner, that expression is passed through to another sub-handler.

Such a system can for example enable most inputs (new issues, comments, etc.) to go through a standard transactor, but important code changes to go through a voting process before being evaluated.

3.2. **Non-hyper static environment.** We have created a few remote machines and they all seem to have the following:

- A big chunk of "domain" state (e.g. a dict of issues).
- Smaller "metadata" state (e.g. used up nonces, next ID, admin set, etc.).
- A set of commands, doing a lot of validation and a little bit of mutation.

Some ways people might "reprogram" a machine:

1. Fix some of the data (add a default field).
2. Tweak validation of an existing command (add more conditions). Maybe also tweak the mutation this command does.
3. Add a whole new command.

Scenario (1) is probably quite common as part of an update which does (2) or (3), because new features of changes to old features might demand a modified data format. In any case it is probably quite easy, just a `(modify-ref~..)`.

The first step of scenario (3) is quite easy: add a new function for the new command. However you must then mutate the input-validator to accept this new invocation. This might be tricky: the pertinent validator might be under a few layers of higher order validators. If you are lucky the validtor you want to mutate was put in a ref (to be dereferenced by other validators) and you can mutate that directly.

In any case it seems that scenario (2) (tweaking existing functionality) is the tricky one. Relaxing the hyperstatic environment seems like one way of making this easier, since now you can just resubmit definitions for the functions you want to change.

Non-hyperstatic would also allow for old definitions to get garbage collected, whereas before this wasn't possible because an old definition could still be in use in some deeply nested closure. Machine participants would then be much more encouraged to update the modules in use on a machine.

4. CRDT data types

4.0.1. *OrbitDB and IPFS log.* Here we will quickly describe how IPFS-log works, which is what all the datatypes of OrbitDB are based on. The ideas of Multi-Writer DAT are quite similar.

IPFS-log is a conflict-free replicated data structure (CRDT) that uses IPFS for storage and replication. The log is formed of *entries* which are documents on IPFS, identified by CID. Each entry contains a set of CIDs of previous entries, usually the latest ones the node was aware of when creating the new entry. This forms a DAG of entries. By being given an entry (a *head*), a node can recursively resolve the pointers to other entries in order to access the whole log. The entries contain CRDT log-operations and so despite the ordering on the entries being only partial, the view that nodes have on the order of the items in the log is eventually consistent.

IPFS-log allows restricting the write-access to a set of public keys. For the moment this set can only grow (this is also the case with Multi-Writer DAT), but the maintainers state that access revocation is currently on the development roadmap.

4.1. **Input queue.** If the owner is offline then participants of a machine may pool the inputs they would like to submit in a queue, which the owner can read from once they are back online. If inputs *always* go through such a decentralised data-structure, then this also makes it easier to monitor which inputs are being potentially censored by the machine.

IPFS-log could be used directly for this, but it would probably be more efficient if each peer drops any inputs which are not valid according to its current view of the machine.

4.2. **Radicle data types.** More ambitiously, we can imagine new Radicle reference types for storing data that isn't as sensitive to ordering as the list of expressions defining the semantics of the machine. For example an an "append-only log" could be initiated as follows:

```
(def valid-comment
  (fn [c]
    ...))

(def comments-crdt
  (new-crdt-log! valid-comment))
```

A validator is used to discard invalid comments. By default a log would not restrict write-access, but this can be controlled by other functions.

Behind the scenes this would refer to a CRDT log either backed by IPFS-log, or something similar. The name of the log is generated deterministically using the machine's ID and a counter. *Writing* to the log happens off-machine, with special effectful functions added to Radicle. First one must obtain the name of the log, and then one may append inputs to it:

REPL

```
> (crdt-log-append! comments {:author "james" :comment "This is so rad!"})
=> :ok
```

Even after people have added entries to the log, the machine will not see them, that is `(read-crdt-log! comments)` will return `[]`. This is because the machine's view must be updated explicitely by checkpointing a head: (`crdt-log-checkpoint!` h) where h is a head of the log. After such a checkpoint `(read-crdt-log! comments)` will return the list of comments as of that checkpoint. Of course this means that the machine's data doesn't actually change until a head is submitted, and this can only be finalised by the owner. However query functions can use the latest head known by the user:

REPL

```
> (some-query-fn!)
=> [...]
> (crdt-log-checkpoint! comments latest-head)
=> :ok
> (some-query-fn!)
=> [...] ;; more comments
```

A more sophisticated query function would make sure to query for the view of the machine and the current view of the CRDT seperately, and then merge them in such a way that e.g. comments which are not yet visible to the machine are styled differently or labelled as "pending", to make it apparent that these haven't yet been checkpointed on the machine. Of course it is expected that the machine owner checkpoints appropriate heads every once in a while, and if this isn't the case then users may complain out of band, etc.

5. UI gateways

It's all well and good to have a transparent state machines but if the only way to access it is through a complex network protocol end-users (especially non-tech-savy ones) have no way of interacting with them. For this we would use trusted *portals*, servers which present a UI as specified in the machine itself.

As a simple example, a machine can record and maintain an HTML template paired with a machine query for populating the template placeholders. Each head of the machine would then determine a unique piece of HTML, which can be hashed. This pair of hashes can be sent to a monitor node for verification. This way the user can be sure that the portal has not tampered with the UI in any way. Such functionality could be implemented as a browser extension. The extension would send the pair of hashes to known monitors of the machine (as specified in the machine itself, or by the portal), and display an icon if they all confirm the hash-pair, or block the page entirely if any one of them does not.

6. Rough implementation and roadmap notes

- The improvements to Radicle-lang are mostly independent of all the other implementation requirements.
- One possible implementation for transparent machines would use IPFS as much as possible. Unfortunately IPFS PubSub messages are not signed at this time, so a Radicle machine would be identified by an IPNS name which points to a record containing, among other things a Radicle key and a pointer to the latest signed head. The Radicle key should never change, and is also used for signing PubSub messages. Care should be taken to create simple interfaces for:
  - Discovery,
  - Ledger replication,
  - PubSub.

  IPFS would initially be used for all three, but at a later stage one item could be swapped out. For example at one point we might implement things on top of libp2p, and use for example DAT for the ledger replication.
  - The machine uses the IPFS DAG for hash linking and replication of the blocks. It is expected that the machine and all its monitors pin all the machine's blocks. Furthermore blocks should contain a pointer to a tree of inputs on IPFS (e.g. branching factor of 32, 100 expressions

stored at each leaf) for faster cold starts. If present, this field also need to be checked by monitors.
  - The blocks contains some extra optional data: the Merkle hash for a dense left-biased Merkle tree of all the inputs of the machine. If this is present, it must be checked by the monitors.
  - IPFS pubsub for all communication between machines and monitors:
    * A machine would publish the current head using IPNS, but also disseminate signed heads using a specific topic.
    * Another topic is used for suggesting new inputs.
    * Monitors use a specific topic for gossiping signed heads. A monitor should subscribe to the topic and try to ensure that the latest signed head it knows about is published on that topic at least once every n minutes.
      · If it encounters a (consistent) signed head it has not seen published on the topic yet, it publishes it immediately.
      · If the latest signed head published to the topic hasn't been republished in the last n minutes then it publishes it.
      · Also hopefully, IPFS PubSub might already have some form of message reduplication.
  - If a machine wants to offer low-latency requests, it can publish a traditional DNS URL which can be used for:
    * Requesting the latest signed head. Obviously any signed head returned by this method should also be published to the PubSub topic.
    * Requesting a range of inputs along with the Merkle proof of validity.
    * Requesting that an input be included. The requester should also publish this request to the relevant PubSub topic to allow monitoring of censoring.
- Simple inter-machine communication only requires basic proofs of inclusion. The more elaborate version would need authenticated Radicle datatypes, which could be added later.
- It would be ideal if Radicle2 launch with some nice story for allowing inputs to be accepted in some form (even in a "pending" state) with high availability. There are a few options for this:
  (1) We also launch a "pending inputs" queue service. Potential inputs can accumulate here when the machine is offline. Clients may query this service in order to create a combined view of the machine.
  (2) We also launch a Radicle machine hosting service, to allow people to spin up trusted Radicle machines at the press of a button.
  (3) We implement the Radicle CRDT datatypes.

  Since (2) seems desirable in any case, it would probably be best to start with this and add (1) and (3) at a later stage.
- Code collaboration infra:
  - Username registry,
  - Project registry,
  - Merge-requests,
  - Better CLI,
  - More frontends.
  - Git replication in-machine,
  - Some story for CI (webhooks)
- Better FFI for common langs,

References

[1] Monadic. Radicle. October 2018
[2] http://www.ua.undp.org/content/ukraine/en/home/blog/2018/the-expectations-and-reality-of-e-declarations.html
[3] Blog post: Secure ledgers don't require proof-of-work
[4] Ceritficte Transparency: How it works
[5] Ogden et. al., Code for Science. 2008. "Dat – Distributed Dataset Synchronization And Versioning".
[6] A transparent, highly scalable and cryptographically verifiable data store.
[7] IPLD – InterPlanetary Linked Data
[8] OrbitDB – A Peer-to-Peer Databases for the Decentralized Web
[9] IPFS-log – Append-only log CRDT on IPFS
[10] Dat Protocol – DEP-0008: Multi-Writer
[11] Aggregation-Based Gossip for Certificate Transparency