Radicle Link

Monadic*

Revision rev1-draft, 2019-11-14, official/experimental

Contents

*team@monadic.xyz

1 Introduction

Radicle Link is a protocol for sharing of and collaborating around source code in a peer-to-peer network.

THIS IS A WORK IN PROGRESS. INFORMATION GIVEN HERE MIGHT BE INACCURATE, INCOMPLETE, OR OUTDATED

> tl;dr We adopt the SSB "social overlay" paradigm to establish a peer-to-peer replication layer on top of distributed version control systems. We model this largely based on an (unmodified) `git` backend, but strive for the protocol to be general enough to allow implementations on top of different paradigms, e.g. `pijul`. We assume a mostly non-adversarial setting. Features requiring strong sybil resistance (or are otherwise difficult to implement in a purely peer-to-peer way) are out of the scope of this specification. We refer to the `radicle-registry` project for solutions to those.

1.1 Overview

1.2 Conventions and Terminology

Data types (schemas) are defined using the Concise Data Definition Language (CDDL)[1], irrespective of the programming language types or the serialisation format they describe.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119[2] and RFC 8174[3] when, and only when, they appear in all capitals, as shown here.

2 Cryptographic Identities

- *DeviceKey* (K_d)

A Ed25519 keypair tied to a device `d`.

- *PeerID* ($I_d = BASE58(PK(K_d))$)

  Textual representation of the *DeviceKey*.

- *DHKey* ($S_d = PK(Curve25519(K_d))$)

  Used for authentication in a Diffie-Hellman key agreement (the static public key `s` in the Noise[4] protocol).

- *CodeSigningKey* (C)

  MAY be equal to K_d for any device `d` (i.e. as long as the key is not shared between multiple devices). Typically, PGP keys already present on a PKI are used for code signing, and shared across devices.

3 Resource Locators

3.1 URLs

Radicle Link uses standard URLs[5] for full-qualified references to resources. We introduce a new scheme, `rad`, for URLs within the Radicle network. The `rad` URL scheme takes the form:

```
rad://<ProjectID>[/<path>]
```

More formally, a `rad` URL is a URI[6] where

```
scheme = "rad"
authority = ProjectID (required)
```

See Metadata, Project for what the *ProjectID* denotes. The `path` component is generally specific to the implementation's repository format. Implementation-independent paths are specified in the relevant sections of this document.

3.2 Paths

Where Radicle interprets data stored in repository objects, path specifiers are interpreted as being relative to the file system tree corresponding to the current context. For example, a path `foo/bar/baz` occuring in a `git blob` reachable from `commit` C is interpreted as being relative to the root of the `tree` of the context C. A leading slash (`/foo/bar/baz`) has the same meaning. If there is no context, the meaning of the path is undefined, and it SHALL be treated as unresolvable.

Within context C, data blob D is itself a (sub-)context, which can be dereferenced using the dot ".". That is, the path `./img/x.png` is interpreted as being relative to D within C.

Note that these semantics are also applicable to path components of URLs, although establishing the context is implementation-specific.

3.3 Content Addresses

Where applicable, content may be referred to by its hash, making it location-independent. Such content addresses SHALL be encoded as "multihashes"[7], wrapped in an approprate "multibase"[8] encoding for textual representation. The choice of the approprate hash algorithm is up to the application.

3.4 CDDL types

In the context of CDDL schemas in this document, the following types shall be used:

```
; URL of scheme `s`
; A polymorphic `s` denotes "any applicable scheme"
URL<s>

; URL Template [RFC6570] of scheme `s`
URLTemplate <s>

; Context-relative file system path
Path

; Content address using hash algorithm `h`
; A polymorphic `h` denotes "any applicable algorithm"
CAddr<h>
```

4 Replication

Data in the Radicle network is disseminated via **replication**. That is, participants (semantically) **pull** the data and updates they're interested in from their peers.

The unit of replication is a **repository**, identified by a *PeerID* in the context of a project. The holder of the corresponding *DeviceKey* is referred to as the **owner** of the repository. Repositories belonging to the same project are represented locally as a single repository, identified by a *ProjectID*. In the context of a project, the owner of a repository may choose to **track** the repositories of other owners (this is called a **remote** in `git` terminology: a named reference to a remote repository). If the remote repository is found to track other remotes, the tracking repository SHALL also transitively track those, up to `n` degrees out.

A repository MUST therefore preserve the transitivity information (i.e. *via* which tracked *PeerID* another *PeerID* is tracked).

Participants in the network can choose to act as **seeds**. This is similar in concept to a "Pub" in SSB[9]: when a peer starts tracking a seed, the seed also tracks this peer, thereby increasing the availability of data on the network. In order to limit its resource usage, a seed MAY require authentication. Note, however, that any participant may act as a seed, and may choose to do so only temporarily.

Tracking a *project* means to implicitly track its maintainers (see Metadata). When initialising a local repository for a project (**cloning**) from a seed, however, it is beneficial to also track the seed (recall that $K_d \equiv S_d$). The transitivity rule above applies here as well.

Notice that a seed may track a large number of repositories for a given project, so cloning from a seed will greatly increase the connectedness of a tracking graph. Also note that, by tracking a seed, upstream maintainers can increase the number of paths leading back to them, such that contributions can flow back up even if they come from participants not within the set of tracked repositories of a maintainer. On the other hand, tracking a seed (or operating one) may increase disk and/or memory pressure on the tracking machine, and increases the risk to be exposed to malicious or otherwise unwanted data. We therefore devise that:

- It SHOULD be possible to selectively mark tracked remotes as "notifications only". This may be implemented using **shallow** tracking if the VCS backend supports it. Shallow remotes MUST NOT be accessible to other peers, except when providing update notifications to them.
- Seeds SHOULD untrack any *PeerIDs* they haven't heard from within a (generous) time window.
- Followers SHOULD take appropriate measures to protect themselves against pathological repositories. What measures to take is outside the scope of this document, but a simple measure is to abort transfers bigger than an upper bound, and **banning** the offending *PeerID* (i.e. stop replicating from it, and refuse network connections).

4.1 Metadata

In order to be replicated on the Radicle network, repositories must store metadata under conventional locations, and with cryptographic signatures as described below.

4.1.1 Format

Metadata is stored and transmitted in JSON Canonical Form[10]. While intended to be manipulated by tools rather than humans, a textual representation is favourable in the context of source code version control systems. Where a binary representation is more appropriate, e.g. for wire transfer, Canonical CBOR[11, Sec. 3.9] SHALL be used.

TBD: It seems like everyone is using OLPC's definition of canonical

*JSON (http://wiki.laptop.org/go/Canonical_JSON), or something else entirely (e.g. SSB). These are either not valid JSON, or it's not clear how canonical they actually are. The spec above **looks** saner, but is lacking implementations, so perhaps revisit this.*

4.1.2 Project

A **project** is a statement of intent: two **repositories** which share a prefix of their histories are *intended* to converge to a common state. It is equally valid for one of them to diverge, which is referred to as a **fork**.

Hence, in a fully peer-to-peer setting, we must rely on the respective **owner** of a repository to state their intent as to what project identity their repository shall have. This statement is made by storing a file `project.json` in the repository with the following structure:

```
project = {
    ; Radicle spec version
    rad-version: "2",

    ; Monotonically increasing version of this file
    revision: uint,

    ; Short name of the project, e.g. "linux"
    ? name: tstr .size 32,

    ; Brief description
    ? description: tstr .size 255,

    ; Sum of relations: tags, labels, external URLs, or
    ;labelled in-tree paths.
    ? rel: [1*32 {
        ? tag   => tstr .size 16,
        ? label => {tstr .size 16 => tstr .size 16},
        ? url   => {tstr .size 16 => URL<s>},

        ; Assumed to exist in `default-branch`
        ? path  => {tstr .size 16 => Path}
    }]

    ; Branch considered "upstream", equivalent to HEAD in git
    ? default-branch: tstr .default "master",

    ; Trust delegation
    maintainers: {
        ; Radicle peer IDs which may alter this file.
        ; Must include the founder key initially.
```

```
            keys: [+ peer_id],
        }
    }
```

A single **founder** of the project signs the initial version of the project metadata file with their *DeviceKey*. The identity of the initial version, as determined by the repository backend, becomes the *ProjectID*. For `git`, this would be the commit hash, while in other systems it might be the hash of the patch which introduced the file (and only the file). A *ProjectID* is thus represented in text form as:

```
<identity> '.' <backend>
```

The only currently defined value for `backend` is `git`, with `identity` being the SHA1 hash of the signed commit introducing the metadata file.

Initially, `maintainers` includes the founder's *DeviceKey*. Updates to the project metadata MUST be signed by a quorum $Q > M/2$, where M is the number of `maintainers`. It must be possible to retrieve and inspect the history of the metadata file, where the order of updates is established by incrementing `revision` by 1 on every update.

Non-maintainers simply copy the project metadata they currently believe is the latest without signing it (unless and until they become maintainers). This may be implemented by just moving the branch head pointer (e.g. `rad/project` in `git`).

In order for a non-maintainer to determine the latest agreed-upon project metadata, an algorithm similar to the root file update process from The Update Framework (TUF)[12] is used:

1. **The founder's trusted project metadata file is fetched.**

   The *ProjectID* serves as a content address for the founder's project metadata file. The file is fetched, up to some `X` number of bytes. If the transfer exceeds `X`, abort and report that the project is invalid.

2. **The co-maintainer's views of the project metadata file are fetched.**

   By inspecting the founder's `maintainers` field, the metadata branches for each maintainer (including the founder) are resolved. The history of each branch is walked back until the initial revision is found. Note that this will typically involve downloading the entire history, without knowing its size (in bytes) in advance. Hence, the validator downloads only up to some `Y` number of bytes, and reports the project as invalid if the transfer exceeds `Y`.

   If no quorum of maintainers can be resolved, the project is invalid.

3. **Check signatures.**

Verify that a quorum of the project metadata files are equivalent according to their normal form, and that each file of this set is signed by the *DeviceKey* corresponding its maintainer's *PeerID*.

If verification fais, abort and report that the project is invalid.

4. **Update to the latest revision.**

   Walk the histories of the quorum determined in step 3. *forward*, and resolve any new maintainers like in step 2.

   For every revision N, verify that

   a. a quorum of the project metadata files are equivalent according to their normal form
   b. the `revision` of N-1 is less than the `revision` of N
   c. a quorum of the files is signed with the maintainer keys from revision N
   d. a quorum of the files is signed with the maintainer keys from revision N-1

   If verification fails, revert to revision N-1, and **discard** all later updates (i.e. reset / untrack the respective branches). Otherwise, repeat step 4. until there are no more updates.

   As new updates are discovered on the network, repeat step 4., starting from the latest validated revision N.

   ***Note**: It is tempting to implement this scheme in `git` using merge commits to sign updates to the project metadata file. Since patches don't commute in `git`, this may lead to (false) merge conflicts, though. Therefore, it is RECOMMENDED to maintain independent histories for each maintainer.*

Maintainers perform the same validation, but are prompted to attest updates from co-maintainers instead of discarding them. It is assumed that maintainers coordinate out-of-band on proposals, such that no conflicts arise. If a maintainer is prompted to sign conflicting updates nevertheless, we expect them to sign only one, and discard the rest. Note that this may lead to an undecided state for an observer – since invalid updates are discarded by all verifiers, it is safe to reset maintainer histories in order to resolve such conflicts.

The project metadata file may be referred to using URLs of the form:

```
rad://<ProjectID>/<PeerID>/rad/project[.json]
```

4.1.3 Contributor

A **contributor** is someone who publishes updates to a project, which includes the maintainers. It is legitimate to publish updates which are not signed by the contributor, hence we require the contributor to attest (sign) the branch heads

to be published. Additionally, a contributor may wish to identify themselves in some way. We provide this by storing a file `contributor.json` in the repository with the following structure:

```
contributor = {
    ; Radicle spec version
    rad-version: "2",

    ; Monotonically increasing version of this file
    revision: uint,

    ; User profile.
    ? profile:
        ; Inline `user_profile`
          user_profile
        ; Web URL which yields a `user_profile`
        / URL<https / http>
        ; URL to another Radicle project (see Note)
        / URL<rad>,

    ; Note: Referring to another Radicle project may imply the
    ; same contributor key, or specify one explicitly. The target
    ; contributor metadata MUST specify an inline `user_profile`
    ; to prevent cycles.

    ; See "Large Objects"
    ? largefiles: URLTemplate<s>,

    ; The public CodeSigningKey.
    ; This can be used to coalesce multiple PeerIDs into a global
    ; "identity".
    ? signingkey: gpg_key,

    ; (Name, Hash) pairs of published branches, e.g.
    ;
    ; refs = [
    ;     "master" = "f620f64f57e5d81977be1ea6a59e131f5bacba3f",
    ;     "pr/123" = "7ac558676aef0408d5e732b080a72480bde1b991"
    ; ]
    ;
    ; Should always include project.default-branch. Removing an
    ; entry "unpublishes" the ref, causing tracking repositories
    ; to prune it.
    refs: {+ tstr => tstr}
}
```

```
user_profile = (
    nickname: tstr .size 32,
    ? img: Path / URL<https / http>,
    ? full-name: tstr .size 64,
    ? bio: tstr .size 255,
    ; <local-part .size 64>'@'<domain .size 255>
    ? email: tstr .size 320,
    ? geo: (float, float) / "earth",
    ? urls: {1*5 (tstr .size 16) => URL<s>}
)
```

TBD: require GPG key to be signed by itself?

Amendments to this file MUST be signed by the *DeviceKey* of the contributor only. Radicle clients are free to only fetch the latest revision of the contributor metadata file for each contributor.

Deferring creation of the contributor metadata file until an actual contribution is made is valid, and amounts to "anonymous tracking".

The contributor metadata file may be referred to using URLs of the form:

```
rad://<ProjectID>/<PeerID>/rad/contributor[.json]
```

4.1.4 Heuristics

TBD: How to determine upstream, how to resolve identities

4.2 Large Objects

In some circumstances it is beneficial to store (binary) large objects outside the source tree of a version control system. This trades local storage space against consistency and data availability – only the most "interesting" (i.e. recent) large files need to be downloaded, and only on demand. Examples of tooling which enables this are Git LFS[13] or Mercurial's Largefiles Extension[14].

Radicle Link does not provide reduced availability storage – it is up to the user to make aforementioned trade-off, and reason about the guarantees of the chosen large file storage system.

We enable projects to use out-of-tree large file storage in a distributed fashion by adopting the convention that it is the *contributor's* responsibility to provide access to the storage backend, as backends may be subject to access control restrictions. The contributor announces an endpoint for collaborators to fetch large files from in their contributor metadata file as a URL template[15], which tooling may inspect to configure large file extensions automatically.

Example `largefiles` values:

```
https://github.com/me/rad-mirror.git/info/lfs/objects/batch
```

```
ipfs://{SHA256_CID}
dat://778f8d955175c92e4ced5e4f5563f69bfec0c86cc6f670352c457943666fe639/{SHA256}
```

5 Collaboration

TBD

6 Network Model

Participants in the Radicle network are referred to as **Radicle clients** or **peers**.

6.1 Membership

In Radicle, data flows along the edges of the tracking graph. We model the membership protocol on the network layer accordingly.

Every Radicle client maintains two lists of `(PeerID, IP, Port)` triples: one containing the *PeerIDs* it is tracking directly, and the other containing transitively tracked *PeerIDs*. Both lists are kept sorted by the time a message was received from the respective peer. The list of transitively tracked *PeerIDs* is at most `k` elements in size.

In order to join the network, a Radicle client must know at least one such triple, for example that of a **seed**.

New triples are discovered either via incoming connections, or by asking connected peers for the `(IP, Port)` pairs of a random subset of already-known *PeerIDs* from either list. In the latter case, the liveness of the peer address is checked by sending it a `PING` RPC. If the *PeerID* is in the transitive set, and the list is already full, the Radicle client picks the most recently seen triple from that list, and sends it a `PING` RPC. If it fails to respond, its entry is replaced by the newly discovered peer triple. Otherwise, its last-seen time is updated and the new peer is discarded.

> *[Note]: This works like Kademlia, but without the XOR metric. There might be a meaningful way to arrange PeerIDs in a structure mirroring the tracking graph (instead of Kademlia's tree structure), so we get some notion of "distance". Not sure how useful that is, though.*
>
> *Should use UDP for efficiency, yet this requires authentication or at least echoing a challenge. Can we have QUIC with Noise, please?*

6.2 Syncing

A common pattern in interactive use of Radicle is to ask the client to `SYNC` with the network, and then disconnect. In order to do this, the Radicle client uses its

membership lists to select peers to connect to, up to a limit. For each successful connection, the initiator determines which of its local projects the remote has, and performs a `SYNC`. This process is repeated until all local projects have been `SYNC`ed with at least one peer.

> *TBD: `SYNC` is envisioned as kind of a combination of send-pack/receive-pack and fetch-pack/upload-pack, in order to minimise roundtrips*

6.3 Gossip

A stateful Radicle client starts by syncing, but does not disconnect afterwards. Whenever a local update is made, it announces `(ProjectID, ref, hash)` to its connected peers (multiple updates SHOULD be combined in one message). This triggers the peers to fetch the update, after which they forward the announcement to their connected peers (minus the originator). This process terminates if the receiver is already up-to-date.

> *TBD: Prove this! It's kinda like EBT, but with lazy push / graft only. A proper EBT may also be possible, but that'd mean the tree root may receive (potentially large) patches.*

The Radicle client periodically shuffles its active connections by picking other peers from its membership lists at random, and attempting to replace active connections with them.

> *[Note]: This is different from HyParView, in that peers cannot initiate shuffles (which is a bit of a concern in untrusted networks).*

6.4 Transport

TBD

6.5 RPC

TBD

7 Optional Features

7.1 Relaying

TBD

7.2 Alternate URLs

TBD

8 Security Considerations

TBD

9 References

[1] H. Birkholz, C. Vigano, and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures." RFC 8610; RFC Editor, Jun-2019. https://rfc-editor.org/rfc/rfc8610.txt

[2] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels." RFC Editor; RFC 2119 (Best Current Practice); RFC Editor, Fremont, CA, USA, Mar-1997. https://www.rfc-editor.org/rfc/rfc2119.txt

[3] B. Leiba, "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words." RFC Editor; RFC 8174 (Best Current Practice); RFC Editor, Fremont, CA, USA, May-2017. https://www.rfc-editor.org/rfc/rfc8174.txt

[4] Trevor Perrin, "The Noise Protocol Framework." 2017. https://noiseprotocol.org/noise.html

[5] T. Berners-Lee, L. M. Masinter, and M. P. McCahill, "Uniform Resource Locators (URL)." RFC 1738; RFC Editor, Dec-1994. https://rfc-editor.org/rfc/rfc1738.txt

[6] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, "Uniform Resource Identifier (URI): Generic Syntax." RFC 3986; RFC Editor, Jan-2005. https://rfc-editor.org/rfc/rfc3986.txt

[7] https://github.com/multiformats/multihash/blob/cde1aef8158d193d73012b7d730013f05c2f7063/README.md. [Accessed: 26-Jul-2019]

[8] https://github.com/multiformats/multibase/blob/f2d3c43f9d30d7dca178dc3220c5bf50963a1e70/README.md. [Accessed: 29-Jul-2019]

[9] https://www.scuttlebutt.nz. [Accessed: 30-Oct-2019]

[10] R. Gibson, "JSON canonical form," Apr-2019.. https://github.com/gibson042/canonicaljson-spec/blob/2d2228fef917a61b382c439fbf0d79e61dd3fdf9/README.md

[11] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)." RFC Editor; RFC 7049 (Proposed Standard); RFC Editor, Fremont, CA, USA, Oct-2013. https://www.rfc-editor.org/rfc/rfc7049.txt

[12] J. Cappos et.al., "The Update Framework Specification." 2019. https://github.com/theupdateframework/specification/blob/0cddec0a60f95f06d2e23ebadbd876eeb62c1df3/tuf-spec.md

[13] https://git-lfs.github.com. [Accessed: 11-Nov-2019]

[14] https://www.mercurial-scm.org/wiki/LargefilesExtension. [Accessed: 11-Nov-2019]

[15] R. T. Fielding, M. Nottingham, D. Orchard, J. Gregorio, and M. Hadley, "URI Template." RFC 6570; RFC Editor, Mar-2012. https://rfc-editor.org/rfc/rfc6570.txt

10 Appendix A: Implementation Notes `git`

TBD

11 Appendix B: Implementation Notes `pijul`

TBD