Lakat
A pluralistic base layer for academic publishing

Leonhard Horstmeyer
(Do not distribute)

May 20, 2023

Contents

Abstract

1 Introduction

With the vast amount of data structures, of query and storage systems, of versioning and networking tools and of large language models, one may engineer publishing systems by posing certain requirements that give rise to a different and arguably more collaborative, efficient and healthy academic culture. This approach can be contrasted with an incremental adjustment of the existing system, which in many quantitative sciences is called the greedy approach. We propose one such architecture around the available technology that we call *Lakat*. Lakat is a distributed database with a local peer-reviewed concensus layer. The system serves as a continuous integration and continuous development (CI/CD) solution for collaborative research and one may conceptionally think of Lakat as a peer-to-peer version of wikipedia with a branch structure similar to git and a peer review system. Our starting point is a set of eight core requirements, that we posit for a publishing system:

1. **Open** – Content and code base[1] should be accessible freely[2].
2. **Permissionless** – No one should be barred from contributing.
3. **Pluralistic** – No monopoly on research opinion.[3]
4. **Process-oriented** – Putting the curation on par with the content.
5. **Conflict-oriented** – Making irreconcilabilities manifest.
6. **Curatable** – Putting the curation on par with the content.
7. **Sustainable** – Data and compute resources should be low and reuse of fragments encouraged.
8. **AI friendly** – AI should be able to interact with the system.

The research paper as the gold standard of publisicing research output poses several threats to the overall scientific endeavour. It is a relict from the times where the printing press had been the latest innovation and where the channels for communicating had a large latency. We mention six issues associated with the paper-formatted research output that are addressed by Lakat:

- It incentivizes the creator(s) of scientific output to withhold preliminary results or results that are either not significant or at odds with a hypothesis. If there are significant results, they may not meet the eye or mind of other creators or consumers of scientific output until the entire paper has been published. Especially if there are significant and important results, it may then even take of the order of tens of months for the paper-formatted research output to be accessible. Thus the process of building on top of previous work and of critical engagement is hindered and in the best case deferred.
- It incentivize creator(s) to wrap minor changes into the costume of an entire research paper reusing a possibly templated introduction over and over again.
- It is but a polished snapshot of a process, an inorganic blob "data structure". The process of reaching a result or of not reaching it as well as the review process are generally not part of the output and not naturally representable in the rigid paper-format. The process often doesn't stop with the paper-publication, but continues thereafter and it requires awkward hacks in the form of corrections or second versions to account for changes.

[1] Here we refer to the code base of any client implementation.
[2] Internet service providers are not free. So we refer here to additional charges.
[3] This is not not necessarily the same as "No single source of truth".

- It creates rigid and isolated islands of content, disregarding potentially conflicting or agreeing intersections. Papers address these intersections with citations that are often placed in an unspecific context, and often reference an entire paper or body of work rather than a particular part. These intersections between different scientific outputs are not only constrained to citations, but entire paragraphs such as introduction or method sections are often simply replicated from previous papers. Thus, making conflicting or agreeing intersections a manifest part of the data structure can overcome the hacky fixes and shortcommings of the paper-format.

- The question of who contributed how much to a research output is often a source of conflict among researchers. A process-oriented publication system facilitates the tracking of contributions and may reduce the cases of unjust allocation of contributorship. In paper-formatted publications the contributorship is proxied by a negotiated ordered list of co-authors, which cannot capture contributions and inevitably leads to unjust allocations.

- The effective barring of potential contributors in paper-formatted research does not increase the level of scrutiny, creativity and quality of the output. On the contrary, maybe another set of eyes can add insights or expand on the results. Why should the self-declared co-authors be in the best position to conduct the research? The fear for the theft of ideas is mostly inherent to bulk-publications and less to process-based research output.

Apart from the abovementioned problems with paper-formatted research, Lakat may also be instrumental for solving other problems with scientific publishing such as the exploitation of scientists regarding their review services and production of output. Even though Lakat does not address this directly, it does provide a base layer to build a system of incentives on top of it.

There are various solutions that have been proposed to improve the process of science publishing with respect to transparency, review, ownership, decentralization, collaboration, openness or fairness. We exhibit proposed solutions and their benefits or shortcomings. Since Lakat sits at the intersection of branchable version control (think git), large collaborative enyclopedias (think wikipedia) and peer-to-peer (think humans), we will focus on solutions in that general triangle.

In 2006 the platform Scholarpedia [](find source) was launched. It is a wiki-based format with a peer review layer, where institutional affiliation is required to contribute. The latter is also one of the drawbacks of this solution. Furthermore, the authors of an article are either chosen or elected. This to our mind has two further problems. First, it raises the question who elects those that elect. Second, the collaborative dimension of wikipedia is lost. In contrast Lakat – like wikipedia – retains the permissionless so that no one is barred from editing or from proposing pull requests to change content (see Section 2 for details). In 2007 the Citizendium fork of the english wikipedia launched [] (find source) with the objective to add a quality assurance layer on top of wikipedia. The concept of approved aricles played an important role. However, who approves the articles. What happens to subsequent changes? Would they have to be approved again or does the approval yield a sort of finality for the manuscript?

There are also many attempts to put part of the existing publishing logic onto a cryptographically secured distributed ledger. Everipedia [] (find source) was a fork of wikipedia. They have also tried to build a quality assurance system on top of it using reputation tokens that can be staked and potentially lost in the process of edits. So instead of tokenizing ownership of edits, they tokenized reputation. Those tokens were deployed on a blockchain (EOS and later Polygon). The project has been archived. Orvium [](find whitepaper link) on the other hand aims to put submission of manuscripts, revisions and publications onto a blockchain or at least have them stored using some decentralized storage provider. Unfortunately it is not evident who stores what, how and where. There is for instance not much information about whether they are creating a dedicated blockchain or use an existing one.(They also launched a designated token to further raise captial for their undertakings.)

This is as close as it gets to Lakat [**?**]

DeSci Labs –

Lateral.io –

Publishing DAO

[**?**] [**?**] [**?**]

Git-ssb

https://www.sciencedirect.com/science/article/abs/pii/S0167739X2200440X

The entire architecture of Lakat is heavily inspired by concepts developed by the Hungarian philosopher Imre Lakatos. Probably in an attempt to contribute to the demarcation problem that was prominent in the field of philosophy of science during Lakatos' times and still these days, he developed the concept of a *research programme*, which is also called *Lakatosian research programme* to avoid confusion with the colloquial use of the former term. The demarcation problem asks about the criteria which tell science apart from "pseudo-science". Lakatos develops his theory on the grounds of a process-oriented account of science. So rather than saying that this or that monolitic bulk of work or set of statements is or is not scientific, he posits that this distinction can only be made on the grounds of processes of theoretical amendments to an existing corpus of statements. He distinguishes progressive

and degenerative amendments depending on whether they strengthen the programme's predictive power. For Lakatos a research programme consists of a *hard core*, which is a set of constituting assumptions, axioms as it were, that capture the essence of a research endeavour and a *protective belt* of auxiliary hypotheses. The key ideas that the Lakat-architecture takes from the concept of the Lakatosian research programmes are threefold: 1) The pluralism of various research undertakings. 2) The process-orientation 3) The distinction between a core and a protective belt. At the heart of these foundational concepts lies the idea that science lives through arguments, differences and discourse. The input of Lakatosian concepts into Lakat can then be described as follows. A research programme corresponds to a branch or a set of branches to which researchers contribute changes or ammendments. There is no single master branch, but rather every research programme has its own branch or set of branches. Conflicts with other branches or even within the same branch are an important aspect of Lakat and can be the source of progress (c.f. progressive ammendments in Lakatosian research programmes). A programme can maintain a set of branches amongst which there are many side or feature branches that support the core branch. These side branches behave like a protective belt.

2 Elements of the Publishing System

We propose a manifestly pluralistic, process and conflict-oriented architecture for the continuous integration and continuous development (CI/CD) of publications with a primary use case of research publications. At its core the architecture consists of a linked data structure that resembles a DAG, where the key objects are branches. This data structure facilitates collaborative work in much the same way as git does. Branches may be thought of as the analogue of a journal in traditional publishing (c.f. Subsection ???). The role of journal editors is covered largely by branch contributors. Branches are chains of blocks that contain submissions. Addition of another block happens via a proof of peer review, where the peers are the contributors to that branch. In that sense branches resemble block chains with blocks consisting of changes instead of transactions and the consensus is provided by a proof of review, a local (i.e. involving just branch-contributors) consensus rather than a global one. The review process follows a weak[4] form of a double-blind protocol, leveraging zero-knowlege proofs. Data is content-addressable and conforms to the ipld format. Storage may be delegated to a selection of storage providers, including decentralized storage networks such as ipfs, storj and others to improve resiliance and longevity. In the following we discuss the individual elements of the proposed system and highlight their interaction.

2.1 Data Structure

2.1.1 Bucket

The most elementary data object is the *bucket*. Each and every submitted item is submitted in a bucket: datasets, paragraphs, images and formulae are contained in buckets. These are examples of *atomic buckets*, expressing the fact that they are the building blocks of the system. Instead of a folder structure we solve the containment relation through designated buckets that we call *molecular buckets*. The data part of those buckets contains merely an arrangement of atomic buckets. One may think of them as the analogue of an article, a book or some other curated content.

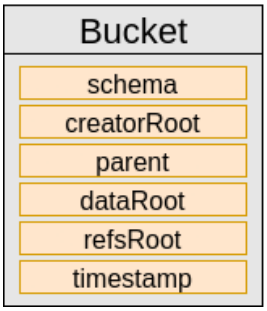

Figure 1: The most elementary type of data container is the bucket. It contains only immutable entries (orange), such as the *schema*, the *creatorRoot*, the *parent*, the *dataRoot*, the *refsRoot* and the *timestamp*.

Every bucket contains six entries: A *schema*, a *creatorRoot*, a *parent*, a *dataRoot*, a *refsRoot* and a *timestamp*. See Figure **??** for an illustration. Here and henceforth the word root refers to the root of a Merkle tree. We go through the entries in turn. The *schema* contains information about the format of the data. For instance we

[4]Weak is to be understood in the sense that both parties may choose to reveal their identity.

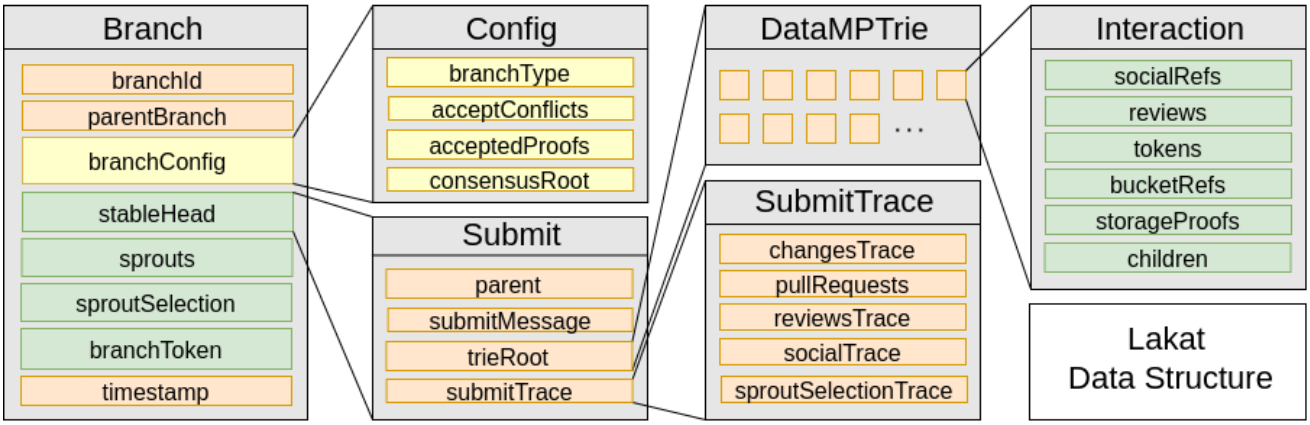

Figure 2: A schematic illustration of the branch object and its entries.

have already mentioned that the data in the molecular buckets is formatted as an arrangement[5]. The *creatorRoot* points to information about the creator of this bucket. Identity on *Lakat* is solved through proofs (see Section **??**). The *parent* is the *content identifier* of the parent bucket. For genesis buckets that would be 0. The *dataRoot* is a content identifier of the data contained in the bucket. In future versions the schema could be absorbed into the dataRoot using the ipld multihash format. This would require a Lakat-specific codec. The *refsRoot* points to all the references made to other buckets within the data. This is necessary, since references to other buckets might be obscured inside the data-encoding. This is an analogue of a list of citations. The *timestamp* records the time of inclusion of the bucket into the branch. It is important to note that we use ethereum block hashes as time stamps in our first version, since the local consensus is too weak to ensure that all participants are truthful to the time otherwise. Anticipating block hash is close to impossible. One cannot change the data inside the bucket. One would have to create a new bucket that points to the original bucket via its parent entry.

2.1.2 Branch

The central object type of Lakat is the *branch*. See Figure **??** for an illustration. Branches represent journals or research communities. They share some properties with *git*-branches and some with blockchains. Every branch contains an id, called *branchID*, that uniquely identifies it. The immutable entries of a branch and the initial head are hashed to produce the branch identifier. The branch also points to a parent branch from which it was branched off. This entry may however be empty for a certain type of branch, namely the sprout (see below). The corresponding entry is called *parentBranch*. This construction turns the set of branches into a linked data structure.[6] At creation time the branch receives a *timestamp*. The previous entries are all immutable. There are then four mutable entries, namely *stableHead*, then the two consensus entries *sprouts* and *sproutSelection* and finally *branchToken*. The stable head is a pointer to the latest stable submit. A *submit* is a set of changes (see Subsection **??** on submits). One may think of it as the Lakat version of an article submission. It has similiarities to a commit in git – not only phonetically – but also to a block in *ethereum*. The addition of new submits works through a consensus mechanism called *proof of review (PoR)* and *lignification* (see Subsection **??**). Also in this respect the branch behaves a lot like a blockchain. Every branch has it's own token, the *branchToken*. It allows funding bodies to fund a particular branch. Token logic is not handled by *Lakat*. Instead this entry essentially points to proofs of transactions on a blockchain where the respective token lives. The purpose of the integration of tokens is to create an incentive layer on top of Lakat, because (unfortunately) *humans* as well as *AI* do not work without incentives. The branch also carries configurational metadata, stored in *branchConfig*. It points to information about the branch type, whether merge conflicts are accepted (see Subsection **??**), the consensus rules and the proofs that are accepted, such as proofs of token transfer or proofs of time. The branchConfig's mutability is more constrained than that of the stable head (See Subsection **??**).

There are three types of branches: *proper branches*, *sprouts* and *twigs*. The branch type is stored in the branch config and can be changed under certain conditions. *Proper branches* can only be modified through the local consensus mechanism (see Subsection **??**). They point to a non-empty set of sprouts, which help with the process of producing stable heads in the proper branch. Proper branches cannot be changed to any other branch type. A *sprout* is a short-lived branch that is exclusively used to grow proper branches. Sprouts behave a bit like

[5]The purposefully vague formulation of an 'arrangement' is due to the intention to keep that format flexible. One may think of this as an ordered list, but one might also consider further directives or clustering of content in a directed hypergraph.

[6](Maybe check this) In *git* a branch is simply pointer to the head commit. In blockchains one often encounters ids attached to the chain (so-called *chainid*) to avoid issues when the consensus mechanism yields two differnt chains.

ommers in the ethereum protocol in the sense that they are contestors to produce the next stable head. They do not have an empty parent branch entry. Sprout branches point to an empty set of sprouts themselves. The sproutSelection contains all the sprouts that are rooted in it. The branchToken entries is empty. The stableHead is immutable. There is only one way to modify the sprout, namely indirectly when it turns into a proper branch during the lignification process (See Subsection **??**). Once a sprout turns into a proper branch the parent branch entry is filled with the id of the branch that it is rooted in. Finally, a *twig* can be thought of a little feature branch. Twigs can be modified through submits by *contributors* of the twig (See Subsection **??** for more information on contributors) or through merges. However, the process of merging into a twig does not need to go through the consensus mechanism of proper branches (See Subsection **??**).

In this paragraph we merely introduce some nomenclature. We distinguish between *core* and *belt* branches, which correspond to *this* and *other* in git. These are not intrinsic properties of branches, but denote the role they play during a merge. Lakat only has one type of merge. The core branch will be updated and the belt branch not (see Susbsection **??** for information on merges). A branch may be a core with respect to one merge and a belt with respect to another merge. This terminology originates in the core-belt dichotomy of Lakatosian research programmes. There is a further distinction that is purely conceptual and is not manifested in the technical specification, but in the nomenclature. We distinguish a *derived branch* from a *seedling branch* in that the seedling branch has a *singularity submit* without a parent (See Subsection **??** for information on submits). A singularity submit corresponds to the genesis block in a blockchain. We invoke here a cosmological metaphore rather than a biblical one. The seedling branch has no parent branch and the corresponding entry points to zero. A derived branch on the other hand has a parent branch that it points to. We say that the derived branch is *rooted* in the parent branch. The *root* of a derived branch is the last submit in the submit history that is also in the history of the parent branch.

We also not that there are various levels at which Lakat can be viewed as a graph, going from high level to low level. At the level of the branches one can form a graph $\mathcal{B}$, where a branch is a node and a directed link from one branch A to another branch B means that B is the parent of A or that B is merged into A (See Subsection **??** regarding merging). This directed graph is not necessarily a-cyclic, because A can be rooted in B and merge back into B, however if one excludes merges it is. At the level of the submits, a graph $\mathcal{S}$ can be created with the submits being nodes and a link can be drawn from a submit q to p when p is the parent of q. This yields a directed acyclic graph (DAG). Finally at the level of the data buckets there exists a graph structure $\mathcal{D}$ induced by the parent reference inside the bucket. There is a graph homomorphism from $\mathcal{S}$ to $\mathcal{B}$, but not vice versa and there are no homomorphisms between $\mathcal{S}$ and $\mathcal{D}$ or $\mathcal{B}$ and $\mathcal{D}$. The lack of a homomorphism between the submit structure and the data structure indicates that these are two separate layers. The relation between the elementary bucket object and the higher level branch object is not simply a many-to-one relation. Different branches may share the some data buckets. In practice one would expect that most of the data inside a branch is shared with at least one other branch. See Figure **??** for an illustration of this relation.

2.1.3 Submit

Submits bundle up changes to the data with some additional metadata. Every submit points to a previous submit, the *parent* submit. There exist *singularity* submits that have no parent. The parent entry of those submits is zero. Like in git or ethereum, there is a field reserved for submit-specific data that we call *submitMessage*. The change of the data within the submit is subsumed in *trieRoot*, which is the root of the *DataMPTrie*, a Merkle-Patricia-Trie that references the data state of Lakat (see Section **??**). The leafs of the trie are the data buckets. They have some resemblance with accounts in the ethereum state trie. Usually only a small part of the entire trie gets updated in a submit. Imagine the trie being all of wikipedia and a submit being just the creation of a new page or even just editing a page. Event though the bucket identifier is immutable it points to mutable entries. This is similar to ethereum, where the leave nodes are immutable account addresses that point to mutable entries like amount of ETH, the contract storage data or the account nonce. The mutable entries in the case of Lakat are made up of information that is attached by other users to the bucket. It is information that is not intrinsic to the bucket. This includes *socialRefs*, *reviews*, *tokens*, *bucketRefs* and *storageProofs*.

The socialRefs resolve to tokens of appreciation, such as thumbs up or down – the gold standard of social media user interaction. The reviews point to data buckets that contain a review or comments on the bucket in question. The tokens entry allow for the integration of tokens to data buckets. The bucketRefs are two collections of references to other buckets. The first collection is immutable and contains all those other buckets that are referenced inside the bucket data. This second collection is mutable and consists of all those molecular buckets that the atomic bucket is part of. This is a reverse registry that can be understood as how much a content has been reused. There is no analogue in classical publishing. StorageProofs are a ledger of timestamped proofs of storage for the bucket.

There are some submits with a specific structure. These are the *pull requests* (see Subsection **??**) and the merge submits (see Subsection **??**). The pull request contains at least one context bucket, called the *review container*, that references all the subsequent reviews. It also leaves a trace of the pull request in the submitTrace.

The merge submit contains all the data buckets of the belt branch and it points to the merged branch id in the submitTrace.

In Lakat conflicts are at the heart of the protocol. They are cherished as the source of progress and sets Lakat apart from conventional publishing systems. We provide a clear definition of a conflict. A *submit conflict* with respect to a branch $\mathfrak{B}$ is a set of three submits π, s_1 and s_2 where π is the parent of both s_1 and s_2 and all three are included in $\mathfrak{B}$. We denote this 4-tupel by $(\mathfrak{B}, \pi, s_1, s_2)$. A submit that creates a submit conflict is called a *conflicted submit* and a submit that does not create a submit conflict is called *conflictless submit*. A *merge conflict* is a submit conflict that arises from a merge submit. Depending on the branch configuration (see Subsection **??**) merge submits may or may not bring about merge conflicts.

2.1.4 Data–Trie

The data buckets as well as the mutable information attached to them can be looked up with the help of a Merkle-Patricia trie, called the *DataMPTrie*. This is cryptographically secure and very useful when resolving the information attached to buckets inside of an article. The keys that are stored in the trie are a truncated version of the content identifiers of the data buckets. And the values are the mutable entries attached to the buckets. To look up the bucket data itself one uses simply the content identifier of the bucket. Storage is handled separately (see Storage in Section). We propose to use a modified Merkle-Patricia trie – very similar to the one used in Ethereum – with four types of nodes: null nodes, leaf nodes, extension nodes and branch nodes []. The data at each node is serialized and hashed using a yet to be specified ipld-codec. The leaf-nodes are special in this respect, because the hashing uses a salt that equals the content identifier of the bucket. Why do we need a salt at all? When a data bucket is published it doesn't have any information attached to it, so without the salt all new data buckets would have the same hash, which is not desirable.

2.1.5 Storage

The data is stored in key-value databases and is content addressed in the sense that the key equals the multihash of the data. Inside the multihash there is information about the storage protocols used. So if one would use, say ipfs, for storage then a certain flag in the multihash would be raised. If also another protocol or system would be used then this would again be seen through the flag in the multihash. To proof that a certain data bucket has been stored, i.e. pinned, that proof is attached to the mutable information of the bucket in the *storageProofs* entry. There are a few more constraints about storage and pinning. It should be encouraged that every data bucket belongs to at least one molecular bucket so that there are no buckets without a context. Thus when a new data bucket is submitted the submission won't be accepted unless it is present in at least one context bucket.

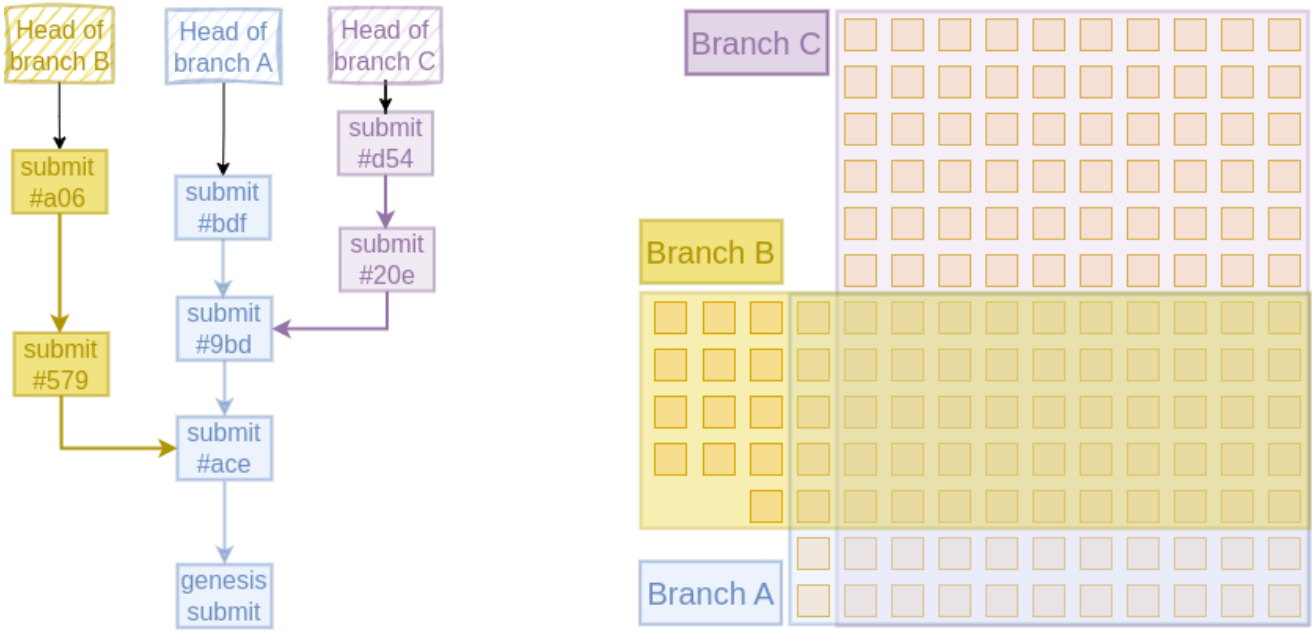

Figure 3:

2.1.6 Branch–Requests

Every branch has its own staging area, where any type of branch interactions are waiting to be included. This is called the *Branch–Requests* (or BR in short). It is similar to the mempool in ethereum. Everyone who is participating in the branch (See Protocol **??**) and who runs a light client may receive branch interactions from users and broadcast them to the network. Here we refer to a *client* as a piece of software that is yet to be written, which interacts with the network. A *light client* is a client that is not capable of doing branch operations, but is capable of receiving and broadcasting branch–requests. Inclusion of requests into the branch, however, requires more (see Protocol **??**). There are eight channels in the branch–requests (see Figure **??**): *submit requests*, *pull requests*, *review commits*, *review submit requests*, *social transactions*, *token transactions*, *storage updates* and *branch creation broadcast*. The requests inside the BR are not permanently stored. Requests are kept for as long as any of the branch contributors keeps track of them. That is where the similarity to the mempool stems from.

Every channel in the Branch Requests has a certain capacity. In particular this aims to prevent that one channel clutters the entire pool of requests, which might happen if the capacity was channel-independent. All requests or broadcasts are serialized. Submit requests contain a serialization of the data buckets that are requested to be added to the branch. Pull requests are simply pings from other branches that seek reviewers. Only by means of a pull request are contributors from the target branch allowed to make modifications to the requesting branch (see Proof of Revie **??**).

(One can submit interaction data to any branch that holds it. Once a branch processes it,)

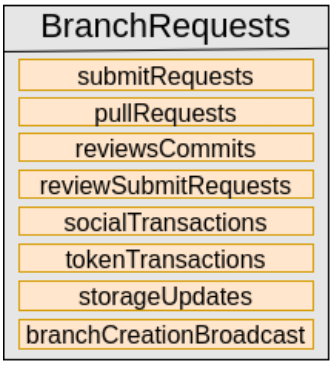

Figure 4:

2.2 Contributors and Contributions

2.2.1 Contributors

Every branch has *contributors*, or rather contributors have branches. A contributor is an account that can prove to have contributed to a given branch. There are four types of contributors for any given branch: *content contributors*, *review contributors*, *token contributors* and *storage contributors*. A content contributor can prove to have submitted to the branch. A review contributor is someone who can prove to have pushed reviews to the branch (see Subsection **??** for information on proof of review). A token contributor is someone who can prove to have deposited funds into the branch. A storage contributor is someone who can prove to store data of that branch. Being a contributor means that you have to prove your contribution for the submits from the root submit of the branch till the current stable head. How does the set of contributors change during a merge? What is the relation betwen the contributors of two branches before the marege and after? When the belt branch merged into the core following a pull request, then the new set of contributors is simply the union of the two branches (see Subsection **??** for the terms core and belt). That holds for all contributor types. When there is no pull request preceding the merge the contributors of this branch are unaltered. The main idea behind the concept of contributors is the mutability, governance and autonomy of branches. Branches can only be modified by their contributors. This attempts to preempt attacks on branches.

2.2.2 Contributions

2.2.3 Proof of Contribution

2.3 Local Consensus

Who decides which content will be added to a branch? In lakat there is no global notion of what counts as science and what does not. There is only a local notion, the detailment of which is the subject of this section. A global

consensus mechanism seems to be a good fit for a ledger that keeps track of values that are or ought to be globally agreed upon, i.e. for values that exist qua their global agreement. In contrast to money transactions, the global scope seems ill-fitted for the publication of research content. In our view this requires a local form of consensus. In the context of Lakat, the scope of the locality is at the branch-level.

What does a branch-level scope mean? This means that the scope is constrained to the *contributors* of a branch. Every branch has a history of submits and is *rooted* in some parent branch or is itself a *seedling* (See Subsection **??** regarding roots and seedlings). In either cases there is a set of contributors to every branch between its root and the current head. Any actor, human or AI, who can prove the contribution of content in any of the branch's submits counts as a contributor (See Subsection **??**).

Branch contributors form the basis of the consensus mechanism. We entrench this deep into the protocol by allowing only branch contributors to make changes to the branch that they are contributing to. This design choice also keeps potential attackers or crackpots from pushing unwanted content to a branch. Lakat does not make statements about what counts as science and what not. What counts as a legitimate scientific contribution purely emerges through the local consensus. One contribution that is viewed as being unfounded or unscientific for one branch might be viewed the opposite on another branch. In some sense this reflects a Feyerabendian approach []. It gives space for pluralism and allows for the organic selection of branches with possibly differing demarcation criteria. There is, however a positive momentum expected to emerge towards methodic agreements at least across branches that are close to each other. Branches that are close have branched off recently and possibly disagree more on technical grounds than on methodic grounds or they are simply feature branches that are to be merged back in. There is an overall incentive to merge branches, derived from the persistence of the data and the value of the token (see Subsection **??** for possible incentive structures).

The local consensus paradigm is governing amendments to branches, both to twigs and to proper branches. Sprouts on the other hand are just auxiliary objects that cannot be modified directly and are thus not amenable to a consensus mechanism. The consensus paradigm for twigs simply states that any branch contributor can push submits to the twig whereas merging into a twig requires a certain fraction of contributors to agree (See Subsection **??** for twigs and Subsection **??** for consensus on twigs). For proper branches the local consensus takes on a different form. It is divided into proof of review (See Subsection **??**), broadcasting and lignification (See Subsection **??**).

2.3.1 Feature Branches

Twigs are meant to be used for rather quick iteration. They behave like feature branches. Here is an example where twigs are expected to be used: If a contributor, human or AI, would like to add content to a target branch, say an article or some modifications or both, it creates a feature branch rooted in the target branch which subsequently goes through the proof-of-review consensus mechanism (see Subsection **??**). Typically the number of content contributors on a twig will be low. Maybe a single author or a small group of authors, as it is the case for article publications. In order to not compromise the momentum and the quick iteration both content contributors and review contributors (if there are any) can push to the branch directly. Merges can also be pushed, but require a fraction of approvals of the content contributors. The fraction is determined in the config of the twig.

2.3.2 Proof of Review (PoR)

Before a branch can be merged into a proper branch it needs to undergo a review. Table **??** summarizes the steps. To start the review process an *issuing branch* creates a pull request from a *requesting branch* to a *target branch*. The pull request is a submit with two properties: First it contains a newly created context data bucket, called the *review container*, that will hold all the forthcoming information of the review. The submit may of course contain other buckets besides that. Second, it leaves a trace of the information about the pull request in the pullRequests entry of the *submitTrace*, namely pointers to the review container, to the target branch and the requesting branch. In most cases the review happens on a twig, which acts as a feature branch. There the issuing branch and the requesting branch are identical, because the twig requests for itself to be pulled into the target branch. However, the requesting branch may also act as a proxy requester. This is the case when a proper branch rather than a twig seeks to be merged into a target branch. Since this intention itself must pass through the consensus rule of that proper branch, one would have to create a twig and include therein the proxy pull request. Once that twig is successfully merged into the actual requesting branch by passing the consensus, the review process can begin on that proper branch for it be mergeable into the target branch. We call a pull request *mature* once it is included in the requesting branch. In the most common scenario where the issuing and requesting branch coincide maturity is immediate. Once a pull request becomes mature a message will be send to the target branch where its contributors are invited to review the requesting branch. The message is simply a reference to the pull request sent to the pullRequests channel of the target's *branchRequests* (see Subsection **??** for branch requests). Any content contributor of the target branch who is not also contributing to the requesting branch can then become a *review contributor* of the requesting branch. They must first publish a review commitment on the

#	Step	Description
1	Create pull request	The issuing branch creates a request for the requesting branch (in most cases identical) to be merged into the target branch. A review container is created.
2	Maturity of the pull request	The pull request is included in the requesting branch (in most cases immediate)
3	Commitment	A content contributor of the target branch publishes a review commitment to the requesting branch. That makes them review contributors of the requesting branch.
4	Review	The review contributors create review submits that are referenced in the review bucket.
5	Completion	The number of review cycles and the coverage of the review meets the criteria of the target's branchConfig. The branch may be merged into the target.

Table 1: Overview of the Proof–of–Review (PoR) process

requesting branch. This makes them official contributors to that branch. It also helps to gauge general reviewers engangement prior to the actual review. This is helpful both for those who seek to merge and those who seek to review. It also increases accountability of the committing reviewer. Failing to supply a review after a commitment could be penalized via the social engagement (see Subsection **??**). Committers publish their commitment in the reviewsTrace of the submitTrace. They cannot submit reviews without a prior commitment. Also, the identity of the reviewer is not public in the sense that the commitment solely contains a zero-knowledge proof that the reviewer is a contributer to the target branch (see Subsection **??**). Of course the reviewer may decide to reveal their identity and this may or may not be in line with the configuration of the target branch.

Reviewers then push review submits to the requesting branch. The submits just contain a proof of contributorship in the target branch. A review submit consists of the following: A bucket with a review, called a *review item*. This bucket should reference all the data buckets that it has reviewed. In the respective interaction data (see Subsection **??**) of all those reviewed buckets a reference to the review item is stored within the reviews entry. Finally the review item gets referenced in the review container of the pull request. Updating the review bucket, as with any bucket update, consists of creating a new review bucket that points to the old one through the parent entry[7]. The branchConfig of the target branch specifies the pre-requisites for a merge. This consists of the minimum number of reviewers, a rule for acceptance and a minimum number of review rounds, which could be one by default. The rule of acceptance could be preset aswell. For instance one could reject requests when a certain fraction of reviewers reject and accept when there are no rejections and specify some rule for the middle ground. Once all the requirements of the target branch are satisfied the branch is ready to be merged.

How about merging branches that do not seek to be merged? This can be the case when trying to merge the newest developments from a remote branch. This case is in fact already covered by the respective consensus mechanisms of twigs and proper branches. Merging into twigs requires a fraction of content contributors to agree (see Subsection **??**). Merging into proper branches requires a pull request and subsequent reviews, so it is not possible to just merge other un-reviewed branches in the same way that one merges reviewed twigs or reviewed proper branches. Therefore, one would have to create a twig that merges the remote branch as a feature. It then requests to be merged und the merge undergoes a review. (clarify)

2.3.3 Broadcasting and Lignification

How are the reviewed pull requests bundled up and sequenced into a single proper branch? Why is the process important? How is the required attention bandwidth for this proess kept to a minimum? In order to explain the Lakat answer to this question we first contrast it to the case of blockchains: There transactions are bundled into blocks. They are then broadcasted across the network of nodes. When different blocks with the same parent are broadcasted, there will be conflicting versions of the blockchain state, which for a single source of truth is undesirable. In ethererum prior to the transition from the proof-of-work to the proof-of-stake these alternative versions were called ommers and were mostly the result of latency in the broadcasting, but of course also attacks or client-software issues. To make sure that a transaction has irrevocably been added into the blockchain one would have to wait for a few block confirmations.

In Lakat we solve the issue through a process that we call *lignification*. The idea is that amongst the potentially plentiful and conflicting versions of the new branch state eventually a new head will be chosen. This head is then called the stable head. The versions are stored as short-lived branches, called sprouts. The *sprouts* entry of the branch points to them. Why is the process of choosing a successor to the stable head important? Here is an

[7] In future versions of Lakat we wish to move to updates via deltas.

explanation: The branch is an object that is kept alive by an ecosystem of contributors. It could get hijacked by a group of bad actors who became branch contributors through a mal-reviewed pull-request. In principle, if this happened, the contributors that disagreed with this malicious onboarding could bail out by creating a new branch. However, this new branch would have to grow the reputation of its contributors anew, seek new storage providers, have a new branch token and would generally have to start from scratch. It might not even be an attack, but a disagreement in the community that leads to a branching. Even though the process of finding a new stable head constitutes an important security measure for the branch, it should not create an overload of attention demanded from the target branch contributors. In most cases there will be no action required. But it is precisely those rare cases, where such a security measure becomes valuable. So one of the requirements for this process is that the branch production continues unambiguously when there is no interference from the community of contributors. In the following we introduce the process of broadcasting and lignification in more detail.

Broadcasting. Henceforth we refer to our proper branch as *core*. Every proposed new merge submit could either become the stable head of core or become the first submit of a new (disagreeing) branch that is rooted in core. We refer to any of those new branches collectively as the *peri* branch. Merge submits carry in themselves the possibility of becoming the head of a new branch. Therefore we decided to "wrap" them into short-lived proto-branches, namely sprouts, whose respective heads are the merge submits. The process of broadcasting is as follows. A content contributor of core, let's call her Alice, creates a merge submit, which is a special kind of submit (see Subsection **??** (ToDo)). This submit is then wrapped into a sprout, which means that the head of the sprout is set to be the merge submit and the content contributors are set to be the union of Alice and all the contributors of the pull-requesting branch. Let's call this sprout S. The branch information of the sprout becomes relevant if it eventually turns into a proper branch, a process which is discussed in the lignification part of this Subsection. The parent of the merge submit is the head of a branch B, that is either the core or any of the sprouts upstream of the core[8]. Alice chooses B, so she decides where to root the new sprout. If she decides to point to a branch that is already pointed at, there will be a conflict. The new sprout S – or rather its branchId – is then added to the sproutSelection entries of B and the sprouts entry of the core (which might coincide with B). The new state of core is then broadcasted to all contributors of core. Note that the new state of core might have received more updates than just the modification of the sprouts or sproutsSelection entries. There can also be further modifications resulting from the lignification process (see next part). The changes, i.e. creation, of the sprout branch S are also broadcasted to its contributors.

Lignification. Once a given submit is the new stable head of core or of peri, it cannot be revoked. We say it is *lignified*. The conversion of a previously flexible object into a rigid amendment of a branch has similarities to the process of lignification in botany. The decision about the stable head is not made immediately, but there is a period of time where it can still be revoked and deferred. This time is called *lignification time*. As mentioned above, the objects that we make decisions about are not the merge submits themselves, but the sprouts that contain them. If there is only one sprout available after the lignification time, then the decision is clear, namely that the submit contained in that sprout becomes the new stable head of core and no action is required. However, there may be multiple sprout options. In this case, we propose to have a deterministic rule that singles out one sprout and we suggest the possibility of vetoing the default deterministic choice. This minimizes the need to vote each time multiple options arise, but more importantly it reduces the attack vector for people to bring branch growth to a halt by proposing alternate – yet still reviewed – merge submits. Vetoing is possible throughout the lignification time. Any branch contributor may register a veto to any of the vying sprouts and therefore against the default sprout. In case that a veto is registered the sprout in question has a chance to provide the next stable head. (how does that work in practice and where is the veto registered and every proposal of new merge submits can also advance the state of the stable head ...) Once a veto is registered, the content contributors can bring in their votes on the rivaling sprouts. After a period of time, called the *engagement time*, the winning sprout will provide the new stable head and the other sprouts can turn into peripheral proper branches rooted in core. Like with blockchains, the state of Lakat does not change by itself, but only through transactions (See Subsection **??** for transactions). This means that only when a new submit is broadcasted can the state of a branch be updated. Furthermore, a branch may only be updated if it is the target of a transaction. If the transaction is targeting core, then peripheral branches cannot be updated and vice versa. As a consequence those ousted rivalling sprouts do not turn into their own branches immediately, but only once a transaction targets them. Some of them may never turn into proper branches at all. Apart from the lignification time and the optional engagement time there is a time allowing for latency issues in broadcasting, called the *broadcastingBuffer*. This ensures that the timestamped vetos or votes are broadcasted and thus recorded before the stable head is irrevocably fixed.

Due to the time between successive transactions it is quite possible that the state of the core, in particular its

[8]The sprouts entry of a proper branch keeps track of all the upstream sprouts, but depending on the last branch update may also contain outdated sprouts. In order to retrieve all upstream sprouts one may "walk" upstream using the sproutSelection entry, which only contains the immediate offspring sprouts of a given branch.

stable head, needs to be updated. Maybe the veto time or the voting time between rivalling sprouts has passed or maybe there are no rivalling sprouts and the stable head simply needs to be advanced. The speudo-code in the Lignification Algorithm **??** outlines the iterative procedure that advances the stable head on each new transaction. It is worth noting that also the sprouts entry and the sproutSelection entry of core get updated by pruning and replacement respectively. An illustration of the lignification process is also shown in Figure **??**.

In practice the broadcasting and lignification can be automated by a script so that it requires less cognitive bandwidth. The script would choose a content contributor of core at random and broadcast collect all the pull requests that meet the merge-requirements from core, then create one or more merge submits from them, go through the lignification process and broadcast the result. Only in the case when there are disagreements would a manual interference be required.

Algorithm 1 Lignification – Advancing the stable head of the branch

```
checkChildSprouts ← TRUE
while checkChildSprouts do
  sprouts ← childSproutsOf(targetBranch::stableHead)
  if sprouts is empty then
    checkChildSprouts ← FALSE
  else
    if all sprouts are within veto time (plus broadcastingBuffer) then
      checkChildSprouts ← FALSE
    else
      if the default successor sprout has no vetoing sibling sprouts then
        targetBranch::stableHead ← defaultSprout::stableHead
        targetBranch::sproutSelection ← defaultSprout::sproutSelection
        prune all sprouts from targetBranch::sprouts that aren't upstream from targetBranch::stableHead.
        checkChildSprouts ← TRUE
      else
        /* one ore more siblings have vetoed */
        if voting has finished, i.e. engagementTime is over then
          targetBranch::stableHead ← winningSprout::stableHead
          targetBranch::sproutSelection ← winningSprout::sproutSelection
          prune all sprouts from targetBranch::sprouts that aren't upstream from targetBranch::stableHead.
          checkChildSprouts ← TRUE
        else
          checkChildSprouts ← FALSE
        end if
      end if
    end if
  end if
end while
return
```

2.3.4 Branch Config Changes

The branch config contains configurable metadata such as the branch type, a flag that can be set to allow only conflictless submits (see Subsection **??**), then the accepted proofs (e.g. proofs of storage, proofs of contributorship, proofs of token transfer) and finally the parameters that determine the consensus (e.g. the number of reviewers needed in the proof of review algorithm). These entries have constrained mutability. They require a merge rather than a plain commit to take effect. For a twig this means that the consensus mechanism for a twig needs to met, i.e. a config-specific fraction of contributors need to approve the merge. For a proper branch this means that the config change needs to go through the proof-of-review (PoR) consensus mechanism (see Subsection **??**).

2.3.5 Branch Operations

Creation. The first branch operation is the creation. There are *genesis creations* and *rooted creations*. As the name suggest the genesis creation is a branch that does not have ancestorial submits. This is similar to blockchains or git, which have a block or submit without a parent. However unlike those Lakat allows for multiple genesis creations. Anyone can at any time create a new genesis branch, which is either a twig or a proper branch. A genesis creation requires the creator to set the branch config. Optionally the creator may also specify a branch token. On the other hand a rooted creation is a branch creation whose initial submit has a parent submit. There

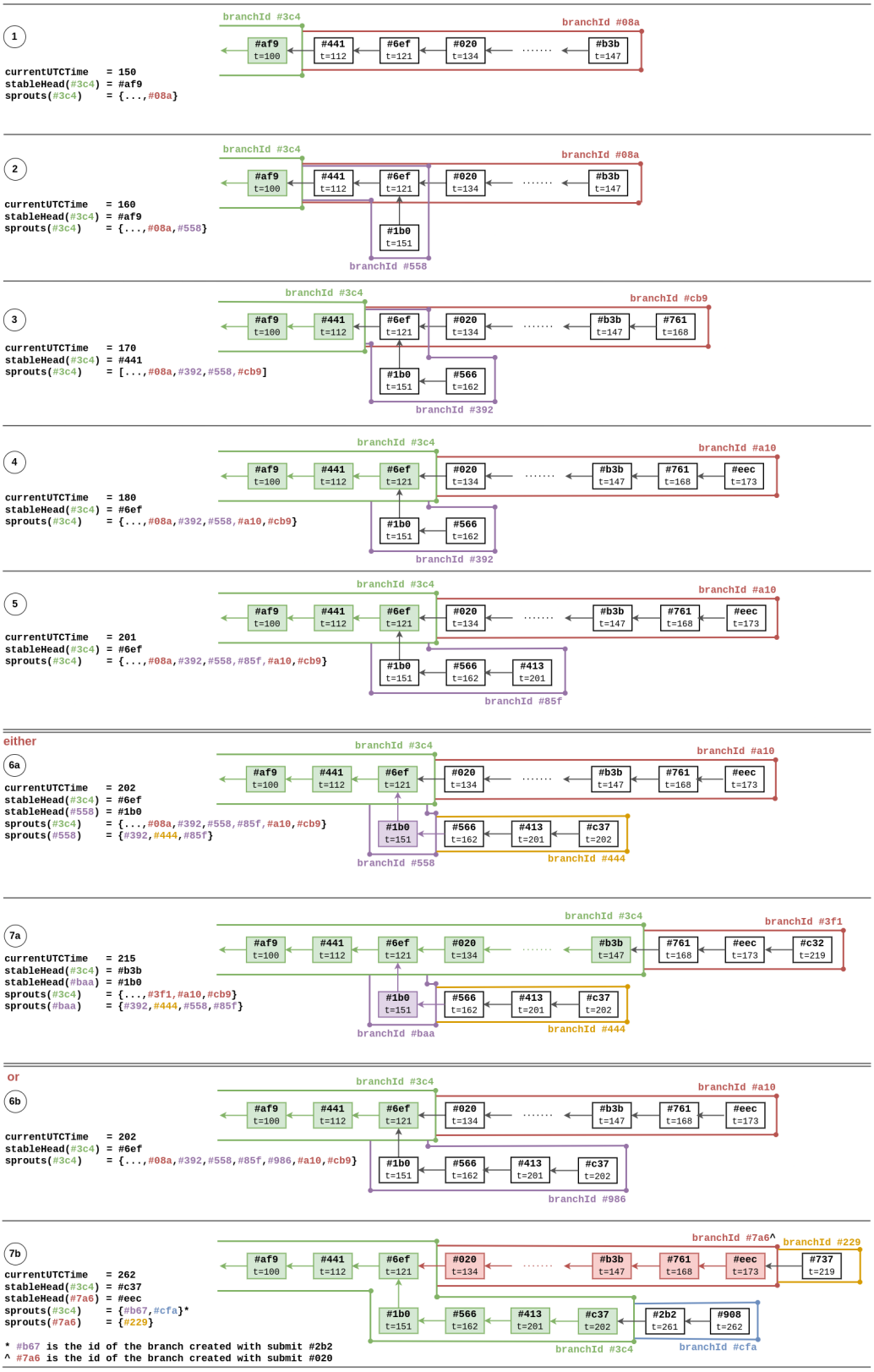

Figure 5: A hypothetical example of a branch lignification for the following example branch parameters: *broadcastingBuffer* = 1, *lignificationTime* = 50 and *engagementTime* = 60. Updates 1 to 5 are within the veto time. There are two optional further paths. Either no veto is given, which is the path from step 6a to 7a, or a veto is given at the last possible time of $t = 201$ ($= t(\text{contestor}) + \textit{lignificationTime}$) through the submit *#413* at Step 6b. Here the contestor is the submit *#1b0*. In this hypothetical scenario the contributors of the stable branch *#3c4* vote in favor of the contesting submit. The new stable part of the branch is thus decided in Step 7b.

are two ways that rooted creations come about. One possibility is that a creator starts a new branch and chooses a parent submit as a root. Anyone may do that for any root branch at any time. The config can then either be chosen anew or inherited. Another possibility involves an ousted sprout, namely one that has been attempting to provide the next stable head in a lignification process. If that ousted sprout receives another submit, it turns into a proper branch. This branch is rooted in the branch for which it was a sprout. It inherits the branch config, but not the token and the branch contributors are the creator of the sprout and the content contributors of the branch that has been attempted to be merged. Any of those contributors can create submits to that ousted sprout and with that submit it turns into a proper branch where the parent entry is set during that conversion. Thus this mechanism for a rooted branch creation is indirect and can only be executed by the respective content contributors.

Merge. In Lakat a merge is the inclusion of changes from one branch into another. There is a strict directionality in a Lakat-merge. Git distinguishes between this and other and in Lakat this corresponds to core and peri, where core is the pulling branch. Merging into a proper branch can only occur after a pull-request (see Proof-of-Review in Subsection **??**). Twigs on the other hand can pull other branches using an approval of a fraction of its content contributors. The fraction is specified in the branch config. After a merge the belt branch may become stale. A stale branch cannot receive submits. Whether a branch becomes stale after a merge depends on the branch config (see Subsection **??**). A merge requires a merge submit (see Subsection **??**) and a cryptographic validation of the branch that is merged. When the conditions for a merge are not met, the merge submit cannot pass the cryptographic validation. For instance if the config of the pulling branch only allows conflictless submits and the belt branch has conflicted submits, then the merge is invalid.

In order to discuss which data buckets are included in the merge submit we briefly introduce the set theoretic slang of A *minus* B for the set of elements in A that are not in B. The set of elements that are in A and B is called *intersection* of A and B and the set of elements that are in A or B is called the *union* of A and B. The respective notations are $A - B$, $A \cap B$ and $A \cup B$. We denote the set of submits of a branch $\mathfrak{B}$ by $\mathcal{S}_{\mathfrak{B}}$. We denote the set of data buckets in the data root of a submit s by $\mathcal{B}_s$. Thus the set of data buckets in a branch $\mathfrak{B}$ with stable head $head(\mathfrak{B})$ is $\mathcal{B}_{head(\mathfrak{B})}$.

We have already discussed that there is no many-to-one relation between buckets and branches (c.f. Figure **??**). There may be data buckets in core $\mathfrak{C}$ that are not in belt $\mathfrak{P}$ and there sure are data buckets in belt that are not in core, i.e. $\mathcal{S}_{\mathfrak{P}} - \mathcal{S}_{\mathfrak{C}} \neq \emptyset$ is not empty. One question that arises in the context of merges is how to combine disparate bucket sets and how to handle that on the level of submits. There are two possible design choices. Either all the submits of belt become submits of core and consequentially also the buckets in $\mathcal{S}_{\mathfrak{P}} - \mathcal{S}_{\mathfrak{C}}$. Alternatively they stay submits of belt and the beforementioned buckets are included in the merge-submit's merkle hash of the data trie. In the first scenario one is faced with the problem, that the submits of belt all have immutable timestamps and parents. Rebasing those would require to loosen those immutability conditions. In the latter scenario one needs to point to the submits that whose data was included. It suffices to point to the peri's last submit before the merge. We opt for the second scenario. Unlike the first scenario, the second scenario has the peculiar situation that belt may have data buckets in common with core even though they do not share any submits, i.e. $\mathcal{S}_{\mathfrak{P}} \cap \mathcal{S}_{\mathfrak{C}} = \emptyset$ yet $\mathcal{B}_{\mathfrak{P}} \cap \mathcal{B}_{\mathfrak{C}} \neq \emptyset$. The only way this can happen in Lakat is if core and belt have pulled from the same branch or from branches that have a common submit in their histories [9].

3 Technical Specifications, Integrations and User Flow

One of the objectives of Lakat is to transition academic publishing from a paper-formatted system to a cryptographically secure, collaborative and pluralistic system that allows for the continuous integration of research output. In order to achieve this objective, we believe that a transition should be as seemless as possible. The publishing system with isolated paper-formatted publications and intransparent review processes is an edge case of Lakat, an unsustainable and hacky one yet sufficient for onramping. We describe in which sense this is the case and how a transition could be achieved.

We can imagine a scenario for Lakat with a set of isolated branches. Each branch is controlled by a single legal entity, namely an academic journal. The academic journal is the content contributor, the storage contributor and the token contributor all in one. When a hypothetical researcher, say Alice (AI or human), wants to publish a paper, she has to send it to the journal. The branch that the journal controls is simply the indexed collection of articles that have been submitted, respectively chained together cryptographically. For a journal to transition its containt to such a branch state is simple. Each paper is stored on a journal-controlled server, thus making the journal the sole storage provider. By adding the submissoin to the journal branch, the journal becomse the sole content contributor and retains all the rights of the contribution. The contribution is no longer owned by Alice at all. In this hypothetical oligarchic aboration (check name) of Lakat, a contribution is a submit with a single

[9] Here we make the distinction between the history of a branch and the set of submits of a branch. A branch may be rooted in another branch, but its history can go beyond the root.

data bucket containing the paper. In summary, there is a way to map the classical publishing system into Lakat. Why is this unsustainable? Given the design of Lakat, this branch would quite naturally undergo diversification through forking. At some point a researcher may quite create a branch routed in that journal branch, which is but a click. Maybe the incentive structure provided by the journal is soo strong that authors are willing to transfer all the rights to the journal voluntarily, but given Lakat's inherent ease of branching it will be a matter of time until there is a diversification.

3.1 Interfaces with existing protocols

We envision Lakat as a base layer for a pluralistic, collaborative publication system that progresses through continuous integration. As a base layer we strive to rely only on a bare minimum of other software and aim to have interface with for existing software or protocols. Here we provide an overview of the so-called tech-stack and the interfaces.

We would like to interface with mediawiki. Mediawiki is an evolving database schema with a php frontend that allows for the creation of knowledge databases such as wikipedia. There are various ways how Lakat could interface with mediawiki. The weakest form requires the conversion of the data contributions in Lakat to database entries in mediawiki. A stronger form converts also contributions to mediawiki into Lakat contributions.

indexing

3.2 User Flow

Here we describe a hypothetical user flow.

4 Attack Vectors