Blockchain Cohomology*

Wyatt Meldman-Floch[1][0000−0001−6582−5925]

Constellation Labs, San Francisco CA 94108, USA

Abstract. The following is an exercise in constructing topological models of distributed system architectures for Blockchain technologies focused on scalability. Applications for these models are provided in terms of emerging Blockchain protocols and scalability approaches are provided, as well as programming models that allow for type-level verification of implementations.

Keywords: Distributed computing · Homology · Algebraic topology

1 Introduction

Existing models of blockchain technology need to be expanded to incorporate new advancements in scalability. These new approaches lack formal verification methods employed by existing scalability approaches, despite their compatibility. By connecting topological models of distributed computing to scalability approaches in big data through homotopy type theory, we create a hybrid topological model for both big data tools and scalability improvements common in emerging blockchain technology, which have adopted the same approaches used in big data. Applications are provided across common scalability approaches in distributed systems that are being adopted by new blockchain technology.

1.1 Related Work

The following is a literature overview from a historical perspective, grouped by overarching subjects amongst references used in formulating the main results of this work.

Topological Models in Distributed Computing Topology is a mathematical field focused on higher dimensional connective properties of algebraic objects. In the context of distributed computing, these objects are generalizations of graphs, and their connectivity properties related to the computability of distributed algorithms. Exploiting certain topological properties of higher dimensional geometric objects to prove results of distributed algorithms is referred to as the topological approach to distributed computing. Techniques from combinatorial and algebraic topology have advanced characterization of synchronous and asynchronous distributed algorithms as well as their solvability [1–4].

* Supported by Constellation Labs

The first applications of topology to distributed computing were in deriving the lower bounds for solving the distributed set agreement [12], which introduced a new paradigm based on algebraic topology for reasoning about asynchronous computations, where at-most one process can fail. The framework consisted of modeling tasks and protocols as simplicial complexes, and applied homology [5, 9] theory to reason about them. A key feature in this framework is that the exponential number of possible executions can be compactly represented using a static topological object in a model independent manner. In certain systems, such as those explored below relating to blockhain technology and scalability, the simplicial complexes are manifolds [6–8], a special class of simplicial complexes that are intuitive due to their geometric nature. The application of homology theory in these models forms a direct connection to Homotopy Type theory(HoTT) [9, 10], the basis for modern static analysis in software engineering as well as in programmatic proof assistants. This connection is realized by a result known as the Curry-Howard correspondence which allows us to know that a distributed system designed with topological models can be verified at compile time given a sufficient type system and typesafe (type preserving) implementation, as is enforced by the functional programming paradigm.

Homotopy Type theory and Verification Homotopy theory is an interpretation of the constructive type theory that connects logic and topology [14]. Specifically HoTT is a tool for constructing models of systems of logic and constructive type theory is a formal calculus for reasoning about homotopy. Computational implementation of type theory allows computer verified proofs in homotopy theory and is the formalism from which static analysis and functional programming emerged from [10].

The Curry-Howard correspondence between mathematics and programming is an isomorphism between mathematical proofs and programs. Proof assistants are tools for representing proofs and programs such that compilation verifies correctness. Proof assistants in math and computer science are increasingly common for managing increasingly complex proofs and programs. Examples of the success of proof assistants and type systems include the proof of the Four Color Theorem and the Feit-Thompson Odd-Order Theorem, as well as large pieces of software, such as a C compiler and the Standard ML programming language.

HoTT doesn't require a proof assistant, it is used in strongly typed programming languages like C or functional languages like Haskell and Scala to provide compile time verification. The ability to verify a program, especially all aspects of a distributed system before deployment has been a fundamental tool that allowed the creation of the Big Data space over the last ten years.

Functional Programming and Big Data Big Data [11, 15] refers to applications pertaining to data sets so big that distributing the processing is not an optimization but a fundamental requirement. It is widely held that the advent of the Big Data field stems from the MapReduce white paper [13], which introduced the first programming model strictly designed for processing and

generating large, inherently distributed, datasets. Given the exponentially increasing amount of data generated by human information systems, the Big Data field continues to grow.

A programming model is the style and interface used for development. The leading programming model, MapReduce, is fundamentally a functional programming model as it is just the combination of *.map*() and *.reduce*() operators, which are by definition declarative functional operations on collections. Functional, as opposed to procedural, programming is a paradigm in software engineering that simplifies verification and efficient distributed processes due to statelessness [16]. If something fails during state transformations developers have to recompile and redeploy their source code, restarting a workflow from scratch. The reason why functional languages encourage scalable software architectures is because of the fact that they eliminate as much shared state as possible from a language. This leads to scalable micro-services or components. These micro-services then scale naturally into larger components.

Blockchain Technology an Scalability A Blockchain is a fairly old concept dating as far back as the early days of information security. At its core a blockchain is an application of a distributed consensus algorithm operating on cryptographically signed data to form a secure append only log, which is secure in the sense that forgery is computationally intractable. Lately blockchain technology has become popular due to its application in creating cryptocurrencies like Bitcoin, but wider applications in verifiable decentralized code execution (provided by Smart Contract platforms like Etherem protocol) and open network security (provide by the Constellation protocol) have emerged.

A major limitation in adoption of blockchain applications is an inherent scalability issue due to the fact that traditional Byzantine Fault Tolerant consensus algorithms require serial execution across a distributed system, making concurrency difficult or impossible. New consensus algorithms and system architectures have emerged however, based on the design of highly concurrent and distributed data processing tools common in Big Data. These new approaches [19, 22] lack formal verification methods employed by similar tools in both blockchain and the big data space however, which is the focus of the models constructed below.

2 Consensus Protocols

Recent advancements in distributed computing adopt methods from algebraic topology to formally defining consensus protocols [1, 2]. First define an execution space as a topological space equipped with a discrete product topology [3]. Defining a distributed process in terms of topology only requires us to care about the structure of the set of possible schedules of a distributed system [4]. By defining an execution space in terms of the homology of protocol complexes [2], define a protocol complex $S_k : P_k\Delta^q$ as the q-dimensional standard simplex

$$\Delta^q = \{x \in \mathbb{R} | \Sigma x_j = 1, x_j \geq \forall j\} \tag{1}$$

at morphism k described by the following vertex set

$$S_k = \{v_{i,0} \dots v_{i,q}\} \tag{2}$$

where $P \subset S$ is the set of all admissible configurations and S is the set of all possible configurations.

Define a consensus protocol $P_*^\sigma(S) : \{S_k, \partial_k\}$ as the singular homology of a simplicial chain complex, carried by a group morphism implementing distributed consensus. Let S_k be a simplex configuration at step k and ∂_k be the differential of a distributed consensus morphism:

$$P_*^\sigma(S) : 0 \leftarrow \dots P^\sigma(S_{k-1}) \xleftarrow{\partial_{k-1}} P^\sigma(S_k) \xleftarrow{\partial_k} P^\sigma(S_{k+1}) \dots \tag{3}$$

where $P_k = ker\partial_k / im\partial_{k+1}$ and is also an abelian group. Thus, $P_* = (P_k) \,|k \in \mathbb{Z}$ is a graded abelian group which is referred to as the homology of a protocol complex S. We abuse our notation of P but rectify by noting that an admissible state k is required for anther step $k+1$, thus we define P as the functor carrying our consensus operator defined below.

Define a consensus operator σ as the group morphism on the singular q-simplex $\sigma : \Delta^q \to S$

$$\sigma_k : S_{k-1} \times P_k \to S_k \tag{4}$$

which are continuous on discrete topologies such as Δ^q [1]. Define homology between configurations as a measure of divergence given by the differential

$$\partial_k(\sigma) = \sum_{k=0}^{q} (-i)^{i-1} (\sigma \circ \delta_q^i) \tag{5}$$

for continuous functions $\delta_q^i : \Delta^{q-1} \to \Delta^q | 1 \leq i \leq q+1$ where

$$\delta_q^i(x_1, \dots, x_q) = (x_1, \dots x_{i-1}, 0, x_i, x_{i+1}, \dots, x_{q-1}, \dots, x_q) \tag{6}$$

As the graded abelian group of our protocol complex is the simplicial singular homology group and σ is our homology preserving map, it is trivial to note that homology holds $\forall k \in \mathbb{Z}$, i.e.

$$\partial_k \circ \partial_{k+1} = 0 \tag{7}$$

As a corollary of the fact that the geometric realization of a simplicial complex is dually a topological space, due to the vanishing homology up to k, $P_k \Delta^q$ is k-acyclic, or that there is a consistently forward moving "arrow of time".

2.1 Protocol Topologies

It's possible to layer protocol complexes defined as above with guarantees about consistency as long as continuity is preserved. Our definition of homology above is verification criteria for the ability to exchange configuration states between protocol complexes.

Specifically, a valid 'layering' as the existence of a functoral vertex map between singular homologies (defined equivalently here as the disjoint subset of protocol complexes) $l : \bigcup_k P_\pi \to \bigcup_k P_{\pi+1}$.

Making use of homotopy type theory allows us to focus on structure by treating topological characteristics called homotopy groups as primitives. Noting that simplicial complexes together with simplicial vertex maps form a category, if we redefine our k-acyclic distributed consensus protocol σ categorically as the functoral carrier Σ_* we can form a chain complex that adheres to the homology theory of homotopy types [5].

Let us define a 'layering' as a Protocol Topology $T_P^\Sigma : \Sigma_* P_\pi$, the singular homology of a chain complex of protocol complexes carried by a homotopy preserving functor Σ_*. The protocol topology is given by the following chain complex

$$T_P^\Sigma : 0 \leftarrow \Sigma_* P_\pi \xleftarrow{\partial} \Sigma P_0 \xleftarrow{\partial} \dots \Sigma P_i \mid i \leq \pi \in \mathbb{Z} \tag{8}$$

where $\Sigma_\pi : ker\partial_k^\pi / im\partial_{k+1}^\pi \to \partial_k^{\pi+1} / im\partial_{k+1}^{\pi+1}$

For protocol complex morphisms $\Sigma_\pi, \Sigma_{\pi+1}$ chain homotopy from Σ_π to $\Sigma_{\pi+1}$ is a homotopy preserving graded abelian group morphism $l : P_\pi \to P_{\pi+1}$ yielding a vanishing homology, i.e.

$$\begin{aligned} \Sigma_\pi - \Sigma_{\pi+1} &= \partial^\pi \circ l + l \circ \partial^{\pi+1} \\ &= \partial^\pi \circ \partial^{\pi+1} = 0 \end{aligned} \tag{9}$$

Noting that these conditions are met by the definitions of an acyclic carrier as above, it follows that a protocol topology as defined above is π-acyclic.

2.2 Applications: Scale out Networking

Traditional blockchains have come up short in their ability to support real world use cases, mainly due to scalability issues. In terms of scalability, sharding or partitioned approaches have taken form in improvements to Bitcoin with Lightning [20] as well as base layer protocols like Zilliqa [21]. In these approaches, relays or subnets are deployed to ferry larger amounts of transactions into a fixed-size configuration state (block). The key to their success is the application of Scale Out Networking [11], adding another layer to the network topology to buffer and compress data which results in faster routing and query times. If we consider either base layer protocol as a protocol complex, then a partitioned or sharding mechanism is described by our model of protocol topology. A type hierarchy is enough to verify a protocols equivalence to a Protocol Topology due to covariance, which is a valid null differential as detailed by R. Grahm above. Specifically, given a base layer protocol Σ_π with block type π, a type preserving operation $\Sigma_{\pi+1}$ such as a buffering service in a multi layered (in this case L2) topology, then if there exists covariance between their data types $\pi + 1$ — $\partial^\pi \circ \partial^{\pi+1} = 0$. Specifically, consider a protocol with an L2 topology:

$$T_P^\Sigma : 0 \leftarrow \Sigma_* P_\pi \xleftarrow{\partial} \Sigma P_0 \xleftarrow{\partial} \Sigma P_2 \tag{10}$$

This protocol topology is a corresponding model to an L2 system architecture in (see Fig. 1). A program implementing the above protocol is verifiable by ensuring functoral covariant state transitions which can be enforced by static analysis [10] (type checking at compile time).

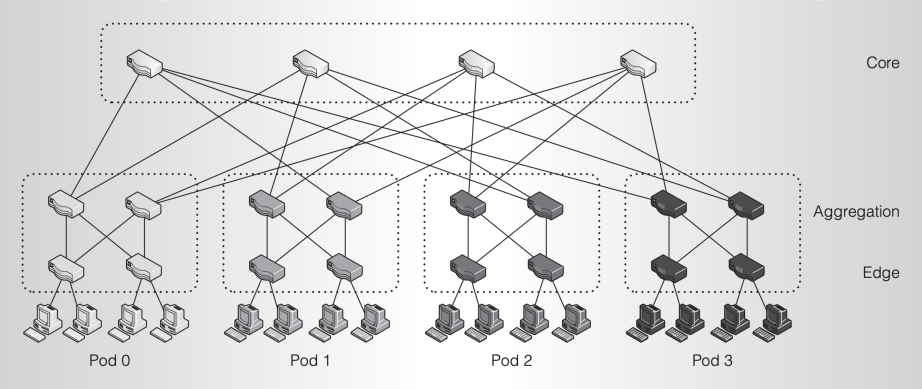

Fig. 1. System architecture diagram of an L2 system [11]

3 Blockchain Cohomology

Distributed architectures designed topologically can be verified at the type level. In order to model a differentiable state, we need to design our topologies such that data locality, or logic behind the distribution of data in a distributed database, are mathematically tractable. We introduce methods from Abstract Differential Geometry, namely finitary cech-deRham cohomology in order to define an orientable manifold from our definition of protocol topology.

3.1 Block Sheaves

First we need to introduce the dual of homology as described above, namely cohomology. In describing our protocol complex it only makes sense to have an arrow moving 'forward in time' as consensus itself is acyclic, with each iteration pointing 'backwards in time' to its previous state. In this sense our evolution was the compounding dimensionality of the space of all configurations, as implied by the discrete product topology of a protocol complex. In defining an orientable manifold, we need to move 'backwards' through our space, i.e. from higher to lower dimension. This is shown as the differential on an arrow going right instead of left.

In principle, Abstract Differential Geometry (ADT) admits any topological space as a base space on which to 'solder sheaves' for carrying out differential geometry [6]. i.e. constructing a manifold. By constructing the protocol topology

within a monoidal category, A. Malios et al. showed that the singular cohomology of a protocol topology is equivalent to an A-module of Z+-graded discrete differential forms, otherwise known as discrete differential manifolds. This forms an execution tree, a sequence of configurations a sequence of configurations in a poset (directed acyclic graph) topology. A decision tree can be assigned to any set of executions that captures the decision of choosing a successive configuration. A blockchain can be defined as an extension of an execution tree, where each block is formulated as a sheaf with a well defined tensor operation and each successive block verified by a decision tree. We define a sheaf ϵ as the 'enrichment' of any cochain $\mathbb{A}$-complex of positive degree/grade, corresponding to the $\mathbb{A}$-resolution of an abstract $\mathbb{A}$-module

$$S^* : 0 \to \epsilon \to S^0 \xrightarrow{d^0} S^1 \xrightarrow{d^1} \dots \tag{11}$$

and homomorphism given by Cartan-Kahler-type of nilpotent differential operator d. We will make use of the fact that an $\mathbb{A}$-module sheaf ϵ on any arbitrary topological space (shown above with an arbitrary simplicial cochain-complex) admits an injective resolution.

Blockchains are naturally equipped with a sheaf, known as a block hash, which contains topological state data about the configuration of the system. Every abelian unital ring (of which this sheaf theortic construction derives from) admits a derivation map [7], allowing us to 'unpack' data within a block recursively under the product operation. By noting the equivalence of Sorkin's fintoposets to simplicial complexes, A. Mallios et al. showed that the Gelfand duality implies that a manifold can be constructed from simplicial complexes [6]. Thus if we reformulate our definition of a consensus protocol above as a sheaf with semigroup operations carried by right derived functors with monadic bind, we can form a manifold.

For a fintoposet (the topological equivalent of a directed acyclic graph), it's incidence algebra can be broken down into a direct sum of vector subspaces

$$\Omega(P) = \bigoplus_{i \in \mathbb{Z}_+} \Omega^i = \Omega^0 \oplus \Omega^0 \cdots := A \oplus R \tag{12}$$

where $\Omega(P)$s are $\mathbb{Z}_+$ graded linear spaces, A is a commutative sub algebra of Ω and $R := \bigoplus_{i \geq 1} \Omega^i$ is a linear (ringed) subspace. It is trivial to notice that $\Omega(P)$ is an A-module of a Z+-graded discrete differential form.

A manifold can be constructed by organizing the incidence algebras of our protocol complexes into algebra sheaves. The n-th (singular) cohomolgy group $H_n(X, \epsilon)$ of an A-module sheaf $\epsilon(X)$ over topological space X, can be described by global sections $\Gamma_X(\epsilon) \equiv \Gamma(X, \epsilon)$

$$H_n(X, \epsilon) := R^n(\Gamma(C, \epsilon) := H^n[\Gamma(C, S^*)] := ker\Gamma_X(d^n)/im\Gamma_X(d^{n-1}) \tag{13}$$

where $R^n\Gamma$ is the right derived functor of the global section functor $\Gamma_x(.) \equiv \Gamma(X, .)$. Note that R^n is equivalent to the i^{th} linear ringed subspace above. These dual definitions of gamma correspond to out definitions of σ and Σ_* with respect to our functoral vertex map l in our definition of a protocol topology.

The sheaf cohomology of a topological space is the cohomology of any Γ_X-acyclic resolution of ϵ [14]. The corresponding abstract $\mathbb{A}$-complex S^* can be directly translated by the functor Γ_x to the 'global section $\mathbb{A}$-complex' $\Gamma_X(S^*)$

$$\Gamma_X(S^*) : 0 \to \Gamma_X(\epsilon) \xrightarrow{d^0} \Gamma_X(S^0) \xrightarrow{d^1} \dots \tag{14}$$

which is the abstract de Rham complex of a discrete manifold X. The action of d is to effect transitions between the linear subspaces Ω_i of $\Omega(P)$, as follows: d: $\Omega_i \to \Omega_{i+1}$.

The finitary de Rham theorem defines a finitary equivalent of the typical c^∞ smooth manifold. Noting $\Gamma_m^{P_m}$ is fine by construction, Mallios et al. show that finsheaf-cohomology differential tetrads

$$\tau := (P_m, \Omega_M, d, \Omega^M_{deR}) \tag{15}$$

is equivalent to the c^∞-smooth Cech-de Rham complex. In our definition of τ, Ω_M is the categorically dual finsheaf (finitary sheaf) of Sorkin's fintoposets P_m, d is an exterior product, and Ω^M_{deR} is the abstract de Rahm complex.

3.2 Protocol Manifold

We've shown how to create a manifold from the cohomology of a discrete topological space. We now show how to construct a manifold from of a protocol topology. Define a cochain-complex within the cohomology theory of homotopy types under the cup product. Making note of the existence of a tensor product in the cohomology theory of homotopy types by E. Cavallo [9] define the protocol manifold as the ringed vector space formed by the direct sum over all protocol sheaves

$$\Gamma^\epsilon_\Sigma = \bigoplus_{0 \le i \le \pi} \Sigma_* \epsilon_i \tag{16}$$

3.3 Applications: Data Locality and Dynamic Partitioning

Data locality refers to the ability to move a query close to node where actual data resides within a distributed system as opposed to moving data over network to the process executing the query. This minimizes network overhead, increases overall throughput and is a best practice in the development of big Data applications. A Protocol Manifold is a construction that forms the total state space of data across nodes implicitly respective of the network topology within a stateful distributed system such as a Consensus Protocol. For elastic or dynamically partitioned systems, where data storage needs to be rebalanced across the nodes in the system when nodes join or leave, it provides a convenient consistency check for the state of the network and an implicit way to organize the rebalancing of data across nodes. A prime example of a distributed data store that corresponds to the Protocol Manifold is the Hadoop Distributed File System (HDFS)(see Fig. 2)., which stores data across nodes with an optimal data locality and minimal redundancy to prevent data loss in the case of partial system failure. The

end result is an implicit orchestration of distributed data data storage with a high level strongly typed API and is key to the ability to form MapReduce APIs across stateful systems as demonstrated by the Poincare Complex below.

Consider an enrichment of a type hierarchy $\epsilon \dots \epsilon_n$, and a state transition

$$J : \Gamma^{\epsilon}_{\Sigma(t)} \to \Gamma^{\epsilon}_{\Sigma(t+1)} \tag{17}$$

where the number of nodes increase or decrease, if all ϵ commute and

$$\bigoplus_{0 \leq i \leq \pi} \Sigma(t)_* \epsilon_i = \bigoplus_{0 \leq i \leq \pi} \Sigma(t+1)_* \epsilon_i \tag{18}$$

then the total configuration space is consistent under the state transition of nodes joining or leaving. Such an ϵ implemented as a type with a product operation [18] would allow us to verify the state transition J by an 'unpacking' operation across block sheaves under the product operation, allowing for verification at the type level both at compile time and run time).

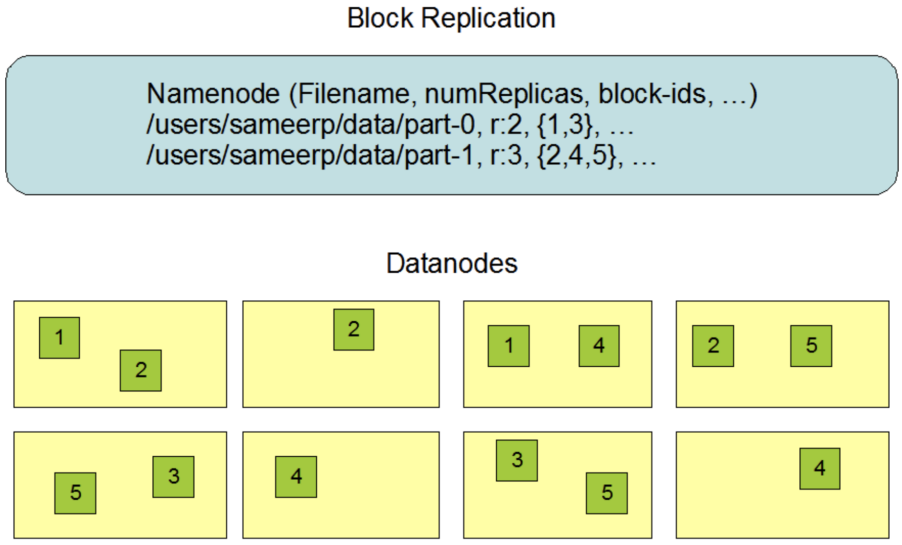

Fig. 2. HDFS replication system architecture [23]. A state transition adding or removing nodes maintains the total set of data.

4 Typesafe Poincare Duality

Up until now we have not explicitly defined functoral group homomorphisms that can construct the complexes described above. We show that the dual nature of the hylomorphic and metamorphic recursion schemes maintain vanishing differentials and thus poincare duality for all π.

If we define a catamorphism and anamorphism with the same f-algebra and f-coalgebra, we can show by construction that the resulting co/chain-complexes are valid definitions of protocol topologies/manifolds and that poincarre duality of the protocol manifold is maintained up to π isomorphism. We define in terms of Σ and ϵ, noting that our functor Σ is a valid f-algebra and sheaf ϵ a co-algebra.

Let us define a hylomorphism

$$\epsilon \leftarrow P \times \Sigma : \Omega^T(\epsilon, P) \tag{19}$$

and metamorphism

$$\Omega_\Gamma(P, \epsilon) : \Gamma_\Sigma \times \epsilon \to P \tag{20}$$

we formally verify by the construction of the following geometric cw-complex

$$\Omega_\Gamma^T : 0 \overset{\partial}{\hookleftarrow} \Omega_{\Gamma^*}^{T^*}(\epsilon) \overset{\partial}{\hookleftarrow} \Omega_\Gamma^T(\epsilon(P_0)) \dots \Omega_\Gamma^T(\epsilon(P_\pi)) \tag{21}$$

that T and Γ form a poincare complex, clearly satisfying the poincare duality as ∂ vanishes in our construction of T and Γ_Σ^ϵ. The fundamental class of our corresponding space is $\Omega_{\Gamma^*}^{T^*}$ which carries the type signatures of our hylo and metamorphisms. Formally define Ω_Γ^T as a Poincare protocol.

4.1 Applications: MapReduce APIs

The design of tools in the data engineering and data science space has been often employ principles from functional programming and type theory due to the benefits they have at verifying code at compile time. Network queries can be implemented with guarantees around correct access governed at the type level, as functoral operators such as .map() and .reduce(). This is largely a source of the origins of scalable data processing tools for analysis and modeling like Hadoop and Spark. Distributed data stores and microservice architectures employ monadic design patterns for improvements on concurrency, static type checking, testing, design patterns, cluster management, training models etc. One benefit is the development convenient API's with Map/Reduce operations simplifying integration by abstracting low level data locality management [15], a key example would be Apache Sparks RDD [17]. Monadic execution models allow complex data pipelines(see Fig. 3) to be implemented and governed declaratively, which allows distributed data stores to be constructed with high level API's. As such, if we follow their construction at the type level in architecture and code design we can develop increasingly complex distributed systems with greater guarantees.

5 Remarks

We have successfully constructed topological models of distributed system architectures for Blockchain technologies focused on scalability and applied them to emerging Blockchain protocols. The Protocol Manifold, Topology and Poincare

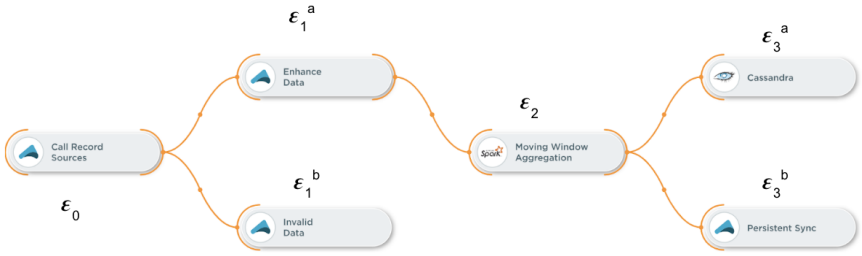

Fig. 3. Example of a data pipeline with a micro-service architecture. ϵ corresponds to indices within a Poincare Complex for each micro-service.

protocol as defined above can be expanded to other distributed systems that undergo finite state transitions. The key advantage of these models is that they can be designed within functional languages with strong type systems that allows for verification at compile time. Future work can apply these models to systems undergoing more complex and possibly differentiable state transitions.

References

1. Nowak T., Schmid U.: Topology in Distributed Computing, Master's Thesis, Vienna University of Technology, (2010)
2. Herlihy M., Rajsbaum S.: Algebraic topology and distributed computing a primer. In: van Leeuwen J. (eds) Computer Science Today. Lecture Notes in Computer Science, vol 1000. Springer, Berlin, Heidelberg. (1995)
3. Alpern, B., Schneider F.: Defining liveness. Information Processing Letters 21, 4 (October 1985), 181–185. Cornell University, (1985)
4. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: the topology of public knowledge. SIAM J. Comput. 29(5), 1449–1483 (2000)
5. Grahm R.: Synthetic Homology in Homotopy Type Theory Robert Graham. arXiv:1706.01540 (2017)
6. Mallios A., Raptis I.: Finitary Cech-de Rham Cohomology: much ado without smoothness. Int.J.Theor.Phys. 41 1857-1902 (2002)
7. Mallios, A.: Geometry of Vector Sheaves: An Axiomatic Approach to Differential Geometry, vols. 1-2, Kluwer Academic Publishers, Dordrecht (1998)
8. Mallios, A.: On an Axiomatic Treatment of Differential Geometry via Vector Sheaves. Applications, Mathematica Japonica(International Plaza), 48, 93. (1988)
9. Cavallo, E.: Synthetic cohomology in Homotopy Type Theory. Master's thesis, Carnegie-Mellon University (2015)
10. Nielson F. Nielson H. R.: Type and Effect Systems. In: Olderog ER., Steffen B. (eds) Correct System Design. Lecture Notes in Computer Science, vol 1710. Springer, Berlin, Heidelberg 114-136, (1999)
11. A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, Scale-out networking in the data center, IEEE Micro, vol. 30, no. 4, pp. 29–41, (2010) https://doi.org/10.1109/MM.2010.72

A Poincare Protocol is a consensus protocol with a monadic execution model like the big data tools discussed above. It describes a distributed data store of topological data formed of configuration data across blocks. An example of this is the Constellation protocol, the design of which is also its origin. Constellation was designed a monadic execution model [22] that can be layered like a Protocol Topology and has the consistency guarantees of a Protocol Manifold. In this model, multiple independent consensus operations can coexist across heterogeneous data types, allowing for complex queries to be performed across concurrent or composite services, like common data pipelines. Each of these consensus operations is referred to as a state channel, an operates on its own corresponding data type ϵ. By constructing a Poincare Protocol, Constellation was able to define a high level MapReduce API [19] for constructing new state channels out of API callbacks across existing state channels(see Fig. 4). Using an algebraic representation of these APIs in the callback tree, common in functional programming [16], each step in the call back tree is governed by the respective ϵ's f-algebra and f-coalgebra and we can verify correctness at the type level both at compile time and runtime. The end result is a consensus protocol with the same model and guarantees of big data tools, allowing for interoperability between the blockchain and big data ecosystems.

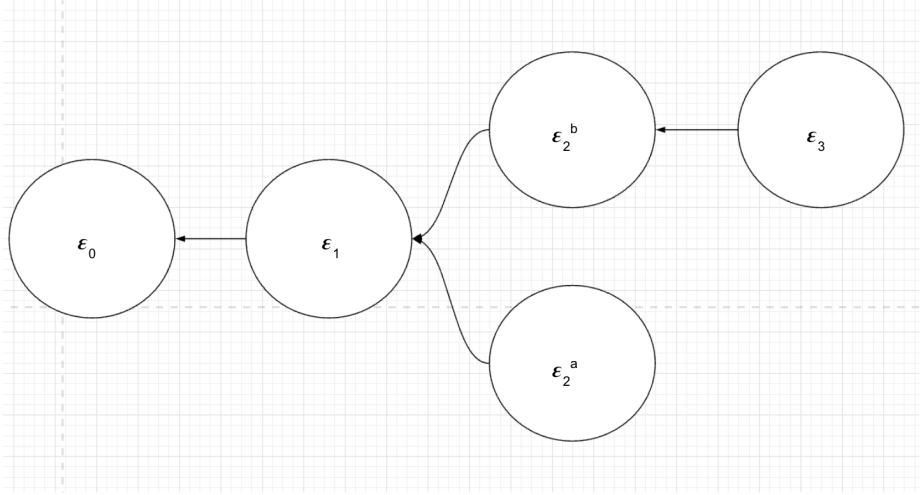

Fig. 4. Callback tree across Constellation state channels (denoted by their ϵ and indices within a Poincare Complex. The tree is defined by algebra/coalgebra of a recursion scheme and is transitively its own state channel.

12. Herlihy M. and Shavit. N.: The asynchronous computability theorem for t-resilient tasks. In: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, STOC, (1993)
13. Dean J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters, In: Proc. 6th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2004, San Francisco, USA, Dec. (2004)
14. Gallier J., Quaintance J.: A Gentle Introduction to Homology, Cohomology, and Sheaf Cohomology. Preprint (2016)
15. Wu D., Sakr S., Zhu L.: Big Data Programming Models. In: Zomaya A.Y, Sakr S. (eds) Handbook of Big Data Technologies, Springer International Publishing, AG, (2017) https://doi.org/10.1007/978-3-319-49340-4_2
16. Chuisano P., Bjarnason R.:Functional Programming in Scala, Manning Publications Co., Greenwich, CT, (2014)
17. Apache Spark RDD documentation, `https://spark.apache.org/docs/latest/rdd-programming-guide.html\#resilient-distributed-datasets-rdds`. Last accessed 15 Oct (2019)
18. Product operator from functional programming library Cats, `http://eed3si9n.com/herding-cats/Cartesian.html`. Last accessed 18 April (2019)
19. Constellation Protocol repository, MapReduce interface via recursion schemes `https://github.com/Constellation-Labs/constellation/blob/7aeaa786d42e0194e31cd8fd0c6b99462cb63f33/src/main/scala/org/constellation/Cell.scala`. Last accessed 18 April (2019)
20. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Joseph Poon, Thaddeus Dryja `https://lightning.network/lightning-network-paper.pdf` Last accessed 18 April (2019)
21. The ZILLIQA Technical Whitepaper, The Zilliqa Team. `https://whitepaper.io/document/16/zilliqa-whitepaper` Last accessed 18 April (2019)
22. Constellation Protocol repository, Validation Monad `https://github.com/Constellation-Labs/constellation/blob/dev/src/main/scala/org/constellation/util/Validation.scala`. Last accessed 18 April (2019)
23. Hadoop Distributed File System design documentation `https://hadoop.apache.org/docs/r1.2.1/hdfs\_design.html` Last accessed 18 April (2019)