An Open-Source **D**ecentralized **A**pplication **P**rogramming **T**oolkit (ADAPT)

Preliminary White Paper

Aleksandr Bulkin

Abstract

ADAPT enables decentralized software in the same way compilers enable software in general and TCP/IP enables connectivity – as a tool that qualitatively simplifies certain complex tasks. It is a software toolkit developers can use to design, implement, and launch their own fully independent decentralized networks. To provide the most design freedom possible, ADAPT imposes no a priori choices of architecture, economics, or governance. ADAPT: (1) provides a programming model for building decentralized databases; (2) offers a flexible data permissioning model; (3) gives you a choice of storage, distribution, and decentralization approaches and timelines; (4) enables convenient programming of persistent multi-party interactions; (5) is developed around a simple, uniform, and extensible programming language; (6) provides an appropriate programming model for implementing cryptographic assets, contracts, and ledgers; (7) enables a broad range of use cases, both centralized, trusted, hybrid, and fully decentralized; (8) uses a generic future-proof data model for interoperability of decentralized databases; (9) does not have a single-platform cryptocurrency or hold an ICO, rather relying on its first applications to create economic value in the ecosystem. ADAPT is the next generation of decentralization technology, promoting scale, complexity, interoperability, and ease of experimentation.

Develop and launch your own decentralized network...

... unconstrained by existing platforms' one-size-fits-all design decisions ...

... in an interoperable ecosystem ...

... and build the system that is right for your users.

Contents

0.1 Terminology

Some terminology associated with blockchain today is ambiguous and misleading. Consequently, in this document we use alternative terms that better correspond to the underlying concepts than the currently accepted terms.

To disambiguate "blockchain," we use the terms (1) **consensus database** to mean an instance of a blockchain system (e.g. Ethereum and Ethereum Classic are two distinct consensus databases); (2) **decentralization platform** to mean the codebase of a decentralized network that may be shared between different databases (e.g. Ethereum and Ethereum Classic are instances of the single Ethereum platform); and (3) **decentralized ecosystem** to mean multiple consensus databases that are somehow connected or related (e.g. Ethereum and Bitcoin are both part of the larger decentralized ecosystem, as they are connected via exchanges, oracles, and interoperability channels).

We use the term **economically motivated consensus** instead of "proof of stake."

We use the term **economic resources** to mean, collectively, all limited cryptographic tokens, rights, badges, and votes. In other words, a resource is data, the mutation of which requires strong guarantees of validity under the network's rules.

0.2 Document Structure

This document combines both technical information about ADAPT and a non-technical discussion of the motivations behind it. If you are not technically inclined, you should focus on the first two sections, Introduction and Motivations. If you have a clear understanding of decentralized engineering, you should explore an abbreviated presentation of the ADAPT's programming language in Section 4 and its overall architecture in Section 5. Section 4.7 presents a simplified implementation of a token wallet contract. Section 6 is a brief illustration of using our methodology to developing and launching a decentralized network that may be interesting to readers familiar with decentralized computing. Our development roadmap and the donation-based funding structure is presented in Section 7, followed by our conclusion.

Those with shorter attention span are welcome to follow the margin notes for the shortest white paper ever.

1 Introduction

1.1 I Want My Own Network

The claim we are making is simple: applications should design and launch their own decentralized networks.

When we make this statement to passive observers, we usually get blank stares followed by a barrage of questions, such as: "How can a small network secure itself?" or "How do you ensure interoperability between applications?" To be sure, these are important topics to consider, but with today's state-of-the-art thinking in decentralization, they do not present significant roadblocks. Recent advances in decentralized software, such as proof-of-stake consensus and atomic swaps, offer a reasonable path to solution.

Active dApp developers, however, increasingly ask for solutions that cannot be provided by existing platforms. They struggle to implement frictionless user experiences, only to realize that the requirements artificially imposed by the host platforms will make things needlessly difficult. Consider the fact that to interact with any Ethereum application, users must hold and spend Ether on every transaction. Who will custody this Ether? If it's the user, then private key management is a problem for the user experience; if it's the application developer, then trust and regulation become a problem.

Developers need flexibility that cannot be offered by existing decentralized platforms, which impose their economic design on everyone.

To be sure, platforms can and will improve significantly. Ethereum may start accepting transaction fees in tokens and implement a scalability solution preventing high-throughput applications from slowing down the entire network. Cosmos and RChain will create zones that separate applications from each other, and even have their own community of miners. But in the end, every platform in existence will follow the set scenario: make architectural choices on behalf of all future dApps and channel value into its own token.

Platforms suck value out of their ecosystems for no good reason.

Provided the security and interoperability problems can be solved, there is no reason whatsoever for an application to share a network with any other application. Applications need to design networks that are right for them. They also need to run their own networks, because sharing resources (both technical and economic) constitutes a risk both significant and unnecessary.

Today's platforms impose one-size-fits-all solution on dApps and create technical and economic contagion risks.

Lacking the imperative to share a network, value extraction into some fundamental platform token is impossible. Yet, it constitutes the primary incentive of a platform developer. So how are we different from Facebook and Google, if we create platform incentives that are ultimately misaligned with those of our users?

Value extraction is the defining motivation of platform developers.

We want to design our applications on their own blockchain networks. This is because we don't want others to make design decisions that affect the experience of our users. You may think that this is a complex undertaking fraught with complexity and requiring specialized expertise. Yet, this is exactly what people thought of computing before PC, software before compilers, connectivity before TCP/IP, and radio communication before cell phones. This paper presents ADAPT, a way to build your own decentralized systems, which will make full-stack decentralized design a commodity.

Decentralized engineering today is akin to software development before compilers.

The ADAPT framework is the layer that commoditizes decentralized software development.

1.2 What Is Blockchain?

Today's blockchain networks are distributed database systems that use a consensus protocol to commit database transactions. Understanding blockchain through the prism of database science offers a grounded view on concepts that otherwise sound exotic. Take, for example, the term "two-way peg" to describe the ability of some blockchains to perform atomic multi-chain transactions. What if we used the established database term "two-phase commit" instead?

Adding a consensus protocol to a database offers a clear advantage over existing frameworks: the ability to implement extremely strong constraints on data integrity by removing agency from human operators and developers. The main use case for strong constraints is building networks that operate on economic value. Certainly, no centralized architecture can host value at the scale that is now customary for blockchain, while also remaining open and accessible. Only consensus-based systems operated by open communities are suitable for this.

Blockchain is a distributed database, and should be seen as such. Blockchain programming is database programming.

At the same time, blockchain systems sharply deviate from traditional databases in their features and programming approaches, mainly because existing centralized databases never developed an approach to permissions management that is sufficiently strong and versatile for public blockchain networks. Servers in traditional database applications run behind the scenes, with the middleware layer of the application responsible for permissions and access management on behalf of individual users. Database security is then mostly concerned with access security, not with managing permissions associated with individual data items.

Decentralized database programming doesn't fit into existing database frameworks, because of security and permissioning needs.

Yet, blockchain systems went too far with their belief that an underlying economic model is fundamental. As a result, this economic model (e.g. the Ether token economics of the Ethereum blockchain) is built to dictate the approach to all further capabilities of the blockchain. Resulting blockchains don't look or feel like databases: they neglect the data part as incidental and unimportant.

Current blockchain programming over-emphasizes payments and economics and undervalues data-oriented features.

Furthermore, basic economic models of blockchain systems are often created to serve the goal of capturing value for their founders and early supporters. While a worthy cause on its own, this drives the market of solutions that tend to be impractically generic for most specific use cases. The goal of directing value to the network's main token leads to many frictions and complexities imposed on adopters, developers, and users (e.g. the requirement to hold Ether in order to send transactions to any dApp in the Ethereum ecosystem).[1]

The economics of the platform unnecessarily complicates dApp user experience.

1.3 Self-Hosted Decentralized Databases

Simultaneously with growth and development of blockchain networks, an altogether different approach to decentralization is arising, best described as self-hosted application server architectures[2]. Platforms of this type offer a unique solution to developers of decentralized applications: a connected network of small, self-hosted,

[1] To be fair, the Ethereum team is working on a solution where you can use tokens to pay for gas. My profound respect to the team for doing this, for this will surely diminish the value of Ether down the road.

[2] A good representative of this approach is Urbit.

feature-rich data servers, in conjunction with a thin user-facing frontend and a blockchain backend managing the economic layer.

Self-hosted application servers are databases with additional functionality to build application logic, but with a trust profile different from a blockchain. They are designed to be trusted by a small group of actors: you and your trusted friend networks. Such servers cannot hold economic value[3], but can serve to build decentralized data applications far superior to large centralized applications of today in terms of privacy, reliability, resilience, and resistance to coercion, while being significantly cheaper and more performant.

Networks of self-hosted non-consensus databases are an important branch of decentralized programming.

It then makes sense to consider all decentralized architectures in the scope of one framework. Such a framework must enable all types of decentralization that we know of today, as well as a number of other architectures[4].

ADAPT is designed to be such a framework. It offers a consistent programming approach to decentralized databases by separating the transaction layer (consensus, trusted, or local) from the database programming layer. ADAPT even offers client-size programming tools, the power of which is derived from the consistency of its programming model in multiple different contexts.

ADAPT offers a consistent programming model for databases, independent of their place within the decentralization spectrum.

1.4 The Unlearning

It is surprisingly difficult to communicate the main premise of ADAPT to decentralization enthusiasts. Surprising, because the mere mention an open toolkit in the world of pre-2010 solutions unequivocally evokes open-source systems, consisting of code, tools, and libraries used to develop something as efficiently as possible. Linux, GCC, Wiki and Git are toolkits. It would not occur to open-source developers of that time that a platform like Linux has to have a currency.

Today, when one mentions a decentralization solution, people think of a blockchain network with a base economic layer. So let me state the main thesis behind the ADAPT framework: ADAPT is a platform in the sense of Linux, Git, GCC, and Wiki; not in the sense of Ethereum, Tezos, Cosmos, and RChain. This document aims to contradict the premise that a base economic layer is a necessity. In other words, ADAPT eschews the need for platform economics as both unnecessary and harmful. It instead strives to enable unprecedented level of freedom on part of individual applications to design their own highly tailored economic frameworks.

A toolkit is a toolkit is a toolkit. Not an economic system. Not a token. Not a network. Not a host.

ADAPT is an open-source ecosystem that offers no token of its own. It doesn't host an investible asset, because doing so diminishes its utility. The initial efforts will be crowdfunded solely by donations or resource contributions (see Section 7.3). Groups developing decentralized applications are likely to want to contribute their efforts to build something that will benefit their products, and it is our hope that this will be the primary mechanism to incentivize the creation of ADAPT – on its merits, not on its profits.

No platform token.

To be clear, this is not to say that ADAPT will not host tokens. Any network

Tokens may be implemented as part of ADAPT applications.

[3] That's not entirely true. Trusted loan networks may operate on such servers without using global consensus at all. We are yet to see them come to market, but promising examples are available.

[4] Section 6 provides a clear example of a hybrid public/private decentralized network.

developed with ADAPT will likely build its own economics, complete with its token and asset model. It is also possible that the currency of one of the first such applications may become a de-facto standard currency for the entire ecosystem. Yet, none of these assets will be ADAPT's platform assets per se.

2 Motivations

2.1 Data and Resource Storage and Organizing

Decentralized applications require full-featured, economically enabled resource- and database systems. It is possible to store large immutable objects such as pictures and videos in an external storage system, such as IPFS. However, such architecture would not suit other types of data objects within more complex dApps. A good example would be a social network or a decentralized marketplace, in which posts, comments, users, orders, and groups comprise a complex network of data relations and must be mutated under the network's consensus rules and security protocol. This example shows that the tools that enable decentralized data architectures must also enable easy modification and querying of datasets of significant complexity.

dApps often require their entire database, not just payments, to operate under full consensus.

Economic resources can also be viewed as data, with the key feature that such data must only be mutated under strong consensus-based constraints. A simple example of this is the representation of cryptocurrency ownership as a database table that associates a number of tokens with every wallet in the system. One must not only ensure that access to wallets is protected by strong cryptographic primitives that control individual permissions, but also that the transfer of assets from one wallet to another strictly preserves the total amount of currency in the system. Such constraints require a significant level of assurance – much higher than that of traditional database frameworks.

Economic resources are data under strong validity constraints.

Following and improving on existing blockchain systems, ADAPT is a resource-base, which means that it is designed to support complex structures comprised of both data and resources such as cryptocurrency, votes, and other consensus tokens that require strongly enforced validity constraints.

The difference between ADAPT and other decentralized database systems, such as BigChainDB, Orbit-DB, and Bluzelle is that: (1) ADAPT eschews single underlying economics as detrimental to the long-term health of the space, and (2) ADAPT databases exist as a replacement for, not an addition to, the blockchain[5]. The entire ADAPT database can run under the network consensus, because ultimately that's what is required for mutable application data.

2.2 Flexible Network Architecture

Proof-of-work consensus algorithms incentivize all decentralized applications and components to share a single underlying blockchain network. The more value Ether has, the more secure the Ethereum blockchain becomes, and the more applications want to utilize it.

When dApps share the same network, they affect each other in unpredictable and detrimental ways.

Unfortunately, sharing a single network amongst an ever-growing number of applications and use cases creates a situation in which problems or transaction spikes in one application cause a drop in the reliability of all others. We call this inter-application contagion. A recent example of this dynamic was the spike in Ethereum transaction costs caused by the launch and consequent exponential popularity of CryptoKitties, a decentralized economic game.

[5]These are not contradictory. ADAPT framework does not impose an economic model, but

Proof-of-stake removes the need for all apps to run on the same network.

Using economically motivated consensus protocols leads to a different outcome. Because security of such networks depends not on compute cycles but on the ownership distribution of tokens, it becomes possible and even beneficial to run many separate consensus databases, eliminating contagion and ameliorating the general scalability problem in the ecosystem. This promotes innovation by encouraging a large number of small and easy-to-create decentralized databases. As long as the interoperability protocols between such databases maintain compatibility at data and transaction levels, applications can implement specific improvements they need in their own networks, rather than requiring complex synchronized upgrades (a.k.a. forks) to a single underlying ecosystem.

A further improvement comes from the ability to run a connected network of ADAPT databases performing a variety of functions. Consensus-based networks are not the only way to build decentralized applications. A different architecture altogether is one where each network member runs their own small database as part of the overall system.

As an example, imagine a decentralized consensus-based search engine for goods and services that provide information on a multitude of small centralized databases run by individual members. This would be an architecture for a decentralized marketplace similar to Amazon. Individual members run their own self-hosted databases that keep track of individual orders and inventory. The overall search engine and payment system is a global consensus database that ensures fairness and openness of this structure.

We desire to create a system for the broadest decentralization possible. In this context, ADAPT does not impose cryptoeconomic choices on its users. Different dApps may require different types of economic incentives, security, and the consensus process. They may even eschew economics altogether. Because ADAPT is a toolkit without a single underlying economic model, such freedom becomes possible.

ADAPT is designed to enable a broad range of network architectures without imposing any specific solutions.

2.3 Recourse and Governance

Governance is essential in the decentralized context. This is because in order to be resilient and scalable, networks must be able to adjust parameters, create bug-fixes, and adopt new features as needed. When we talk about decentralized governance, we usually refer to a process wherein decisions are made without creating a power imbalance that can be exploited by the governing entity.

Formal governance is essential for decentralized systems.

With such a system in place, it then becomes possible to implement recourse within the decentralized network. Recourse here refers to one's ability to resolve complex transaction conflicts, revert erroneous transactions, implement last-resort dispute resolution, reverse or block transactions belonging to parasitic entities, and so on.

Different applications have different recourse and governance needs.

The need for recourse and governance, although broadly disputed,[6] is a major reason for why applications should run on their own networks. Each dApp may choose and implement its own governance structure that extends to all aspects of the decentralized database hosting its data. We encourage experimentation with a

ADAPT will offer a range of pre-built governance modules.

individual ADAPT databases can and should have it.

[6] See for example Vlad Zamfir's article on this topic.

number of different governance models in this context. ADAPT will contain plug-and-play governance components, and we hope that the developer community will contribute many possible solutions for others to employ.

2.4 Gradual Decentralization

Decentralization is at odds with upgradability and the agile development process, placing significant and disproportional burdens on upfront testing of blockchain software. Upgrading economically enabled decentralized code is a single point of attack and misbehavior, and so has to be severely restricted. Such restrictions, however, prevent organic improvements and significantly complicate fixing bugs. This leads to developers trying to put as many features as possible up front, which conflicts with the often aggressive deadlines placed on product MVP delivery by regulatory and funding needs. This causes uncomplicated projects to remain in development stages for months and even years for fear of missing a critical bug or an important feature.

Decentralization is at odds with upgradability and agile development.

Effectively, blockchain developers have only a limited set of options: run upgradable software that does nothing (testnet), or run useful software that can't be easily upgraded (mainnet). Applications of any value are forced into a model that is only useful for systems with significant economic value – that of highly restricted upgradability.

Slowly moving an application from a centralized to an increasingly decentralized configuration enables ease of critical early stage upgradability.

By offering a modular architecture, ADAPT strives to make decentralization a gradual process. It will allow applications that hold small economic value to run in a centralized environment; those that hold medium economic value to run in a semi-decentralized environment with trusted validators; and those that hold large economic value to run in a fully decentralized trustless environment. The same application can make its way through all these stages in sequence by switching the consensus and storage layer of the system, while keeping the code that defines its interactions and data layout. Essentially, gradual decentralization amounts to a series of steps, each performing a controlled reconfiguration and redeployment of an existing system.

The extent of decentralization should mirror the organic growth of economic value.

Furthermore, regulatory restrictions in some jurisdictions place complex constraints on compliant issuance and the distribution of cryptoassets. A broad guideline adopted by many regulators to identify and restrict securities is reliance on the efforts of others to receive profits. Note that mining would not constitute such reliance, as the miner is required to actually run a network node to generate profits. Modern dApps have neither the need nor the ability to mine their currency through work. Allowing dApps to launch on their own networks and decentralize gradually permits initial supporters to contribute work to dApps in the form of mining, or more generally, perform transaction validation and data storage services. This may allow compliant funding for dApps, which is not possible with current blockchain architectures.

Gradual deployment may be the answer to some regulatory questions.

3 A Quick Overview of the ADAPT Ecosystem

We start with the overview of the toolkit components and the structure of a single ADAPT database. We go on to show how various types of ADAPT databases fit together into a multitude of decentralized architectures. We finally describe a vision of the ADAPT ecosystem as a whole, a constellation of loosely coupled interoperable database instances, both consensus and private, with a variety of infrastructure services providing important functions: connectivity, secondary markets, identity, and so on.

3.1 The Anatomy of a Database

Figure 1 lays out the structure of the ADAPT toolkit as a framework for developing decentralized databases with the typical layers present in any database instance. A full-featured dApp may contain one or more such database instances. For example, a dApp may need to instantiate a single consensus database to serve as its economic layer, or a multitude of private databases that interoperate at the data and message level. In each case, the specific database instances will use different parts of the toolkit.

ADAPT database programming model is agnostic to its consensus algorithm or lack thereof.

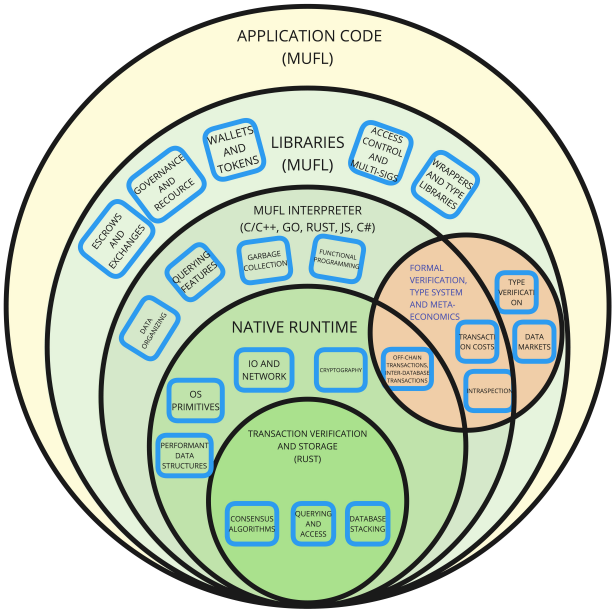

Figure 1: Layers of code within an ADAPT database. Blue rectangles indicate components that are part of the toolkit.

The generic database instance structure shown in Figure 1 has five layers:

1. **Application-specific code** in ADAPT's programming language, MUFL,

⟵ *Layers of tools and component for programming decentralized databases.*

oversees the participant interactions and manages the application's data.

2. **Application libraries** in MUFL provide reusable primitives, such as tokens, wallets, exchange contracts, voting, and many others.

3. **The MUFL interpreter** translates MUFL into calls to the MUFL API transmitted down to the native runtime.

4. **The native runtime** implemented in a native programming language, such as C++ or Rust, exposes the MUFL API for all data structures and library functions provided by the native runtime. This makes available such primitives as performant tables, network primitives, string operations, standard datatypes, and so on. The details of MUFL API are further detailed in Section 5.1.

5. **The data storage and transaction verification layer** accesses data in the database according to the database's storage protocol. This layer hosts the consensus protocol, if the database instance is decentralized; or a simple transactional storage layer, if the instance is private; or anything in between.

Choose the components that are right for your network.

ADAPT emphasizes putting flexibility in the hands of dApp developers. When a developer wants to launch a consensus network, they implement the database in MUFL and launch it on top of a consensus algorithm provided by the toolkit. The consensus algorithm is fully abstracted from the application.[7] ADAPT allows the developer to write and test their application on a centralized private database and distribute it slowly and responsibly by putting it through a number of upgrades, replacing the transactional layer with a more decentralized consensus algorithm at every step. The application code will undergo only minimal changes during these upgrades.

The native runtime of ADAPT implements a number of important primitives and exposes them to the MUFL programmer via the MUFL API. This layer implements functions such as cryptographic primitives (for transaction signing), native datatypes, native high-performance data structures, and procedures used to access other databases. Ideally, MUFL databases can serve as clients to other MUFL databases, although, in the case of multiple consensus databases, this could be a difficult functionality to implement.

3.2 The Spectrum of dApp Architectures

ADAPT enables a broad spectrum of decentralized architectures. The ones we chose to present here are those with which we are already familiar. We imagine that the ADAPT framework may be used to put together a variety of others. Discovering them is up to the future innovators experimenting with the toolkit.

[7] This is not 100% accurate, as the consensus algorithm may need to access data within the database. Properly architecting this interaction is an important goal of ADAPT.

3.2.1 Pure Blockchain dApp

In many current applications, the blockchain is the sole moderator of interactions between participants. Applications such as Cryptokitties, SiaCoin, Augur, and Steem are "pure" decentralized applications in which thin clients connect directly to a consensus database on the backend, presenting data to users and enabling them to transact.

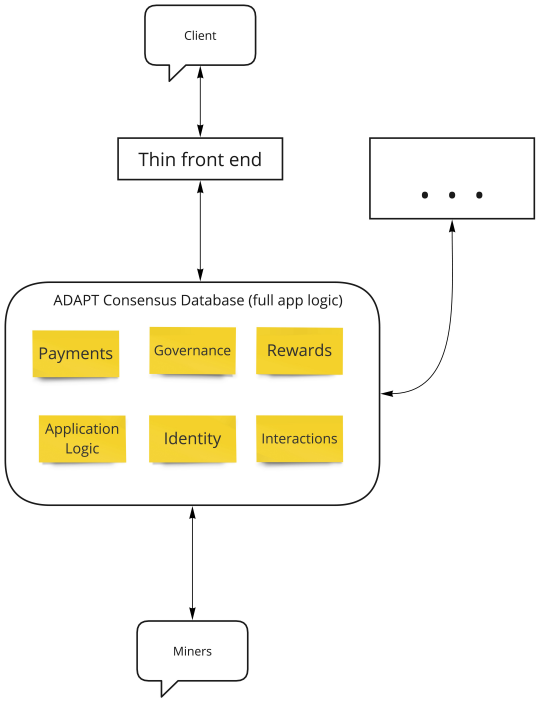

Figure 2: This is an application in which thin clients connect to a "fat" blockchain that drives all the application logic and participant interactions.

Using the ADAPT toolkit to build such a network, one would program the application logic in MUFL, develop the base economic model or chose it from the available components, and launch the database with the transaction and storage layer based on one of the available consensus algorithms. The entire contents of the database in this case operate under the network consensus, which could be expensive, but is an essential feature of fully decentralized applications.

3.2.2 Semi-Decentralized Application

A slightly different model is presented in Figure 3. In this case, a decentralized database serves as a backend for a centralized service or business model. Examples are a messaging application with its own cryptocurrency, a video streaming service that wants to reward content creators, or even a business that wants to put its stock on the blockchain.

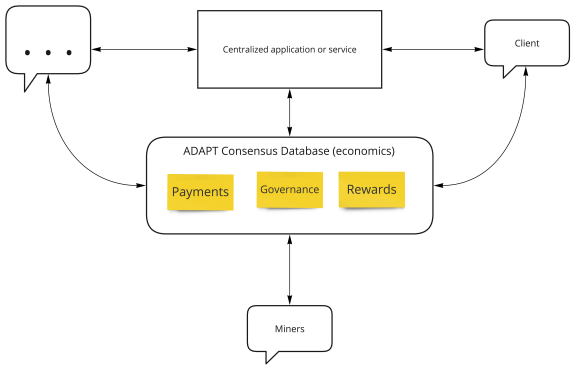

Figure 3: Some parts of this application run privately or are otherwise centralized. Decentralization is only used to host governance and economics.

To build such a network with ADAPT will require much less effort because the database only contains economic components but no application logic. We envision being able to create such a system in a few days.

Note that even though ADAPT is designed for applications to run on their own decentralized networks, nothing prevents the creation of decentralized databases that host multiple assets or economic systems. A case could be made for some networks that permit users to create their own assets. Hosting multiple simple economic systems in the same network would improve tradability and liquidity of assets, if economic interoperability between independent databases proves difficult or expensive.

3.2.3 Hybrid Application

This architecture (see Figure 4) is based on one core consensus database providing economics and directory services, with each user running an independent private node that hosts their own data for others to access directly. It improves scalability in cases where some data doesn't require network consensus. This architecture serves such use cases as decentralized marketplaces, content and data sharing networks, and distributed social networks. Section 6 provides an in-depth example.

The ability to implement both private and distributed databases in the same framework improves the ease of developing such systems. Private databases can

transparently request data from other private databases, as well as the consensus database. Using the same language to implement all database types allows effective code reuse.

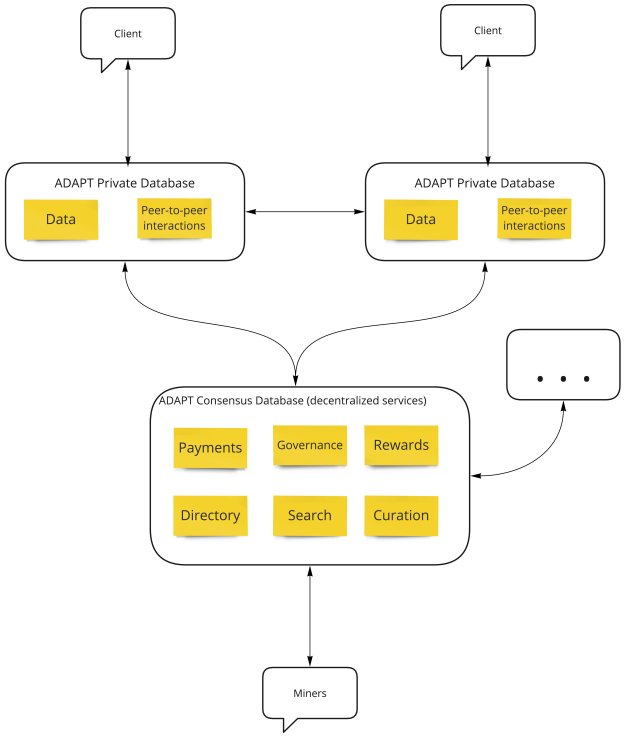

Figure 4: This is a hybrid application where some data is stored under consensus, while other data is hosted privately.

3.2.4 Non-Consensus Networks

A particularly interesting decentralized architecture is one that does not use a consensus database, but is rather a network of interoperating private databases. This is illustrated in Figure 5. This architecture is well-suited for use cases such as decentralized messaging and social media. Of particular interest are peer-to-peer credit and trust networks that enable exchange of economic value between participants without employing a consensus algorithm.

ADAPT databases in this model maintain a list of peers in a globally connected network and operate under a consistent protocol for data exchange. We believe that while network consensus is an important innovation, it is by far not the only way

to achieve resilience of a fully decentralized system. Distributed non-consensus networks may be a better solution for some important situations and may have better performance characteristics than consensus databases. This decentralization model is most robust against connectivity black swan events, because establishing both liveness and consistency in consensus databases is hard under large-scale internet malfunctions, such as prolonged network fragmentation.

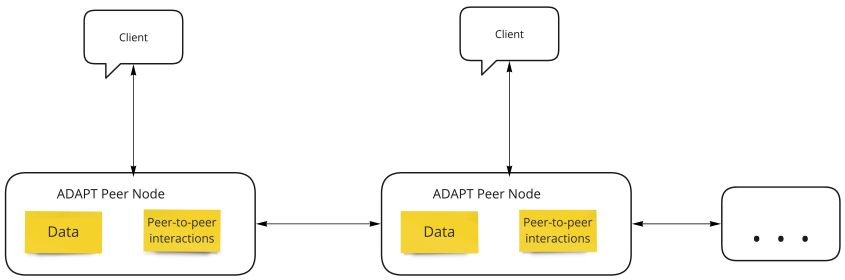

Figure 5: This is a distributed network of non-consensus databases.

3.3 The ADAPT Ecosystem as a Whole

The ADAPT ecosystem is comprised of all databases built using the framework. Assuming that economic interoperability of independent networks is sufficiently commoditized in the future, we imagine all decentralized applications in the ADAPT ecosystem forming a loose network of economic and information exchange, as presented in Figure 6.

Databases serving end-user applications form the application layer and are able to access data and economic infrastructure implemented by the applications in the infrastructure layer. The interoperability services, such as oracles and transfer mechanisms, provide connections between the two layers.

***ADAPT** application databases form a loosely connected ecosystem of data and economic resources.*

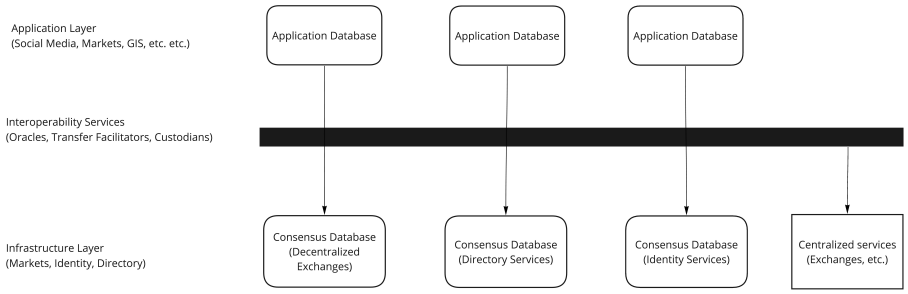

Figure 6: The ADAPT ecosystem is a meta-network of interoperating consensus databases.

4 Programming Language

4.1 Why a New Programming Language?

Section 1.2 discusses the basic requirements for the programming model appropriate to the decentralized context. To summarize, it posits that blockchains are databases, but with a specialized set of needs: (1) to host code in addition to data (e.g. smart contracts); (2) to control access and permissions at a fine grain level (e.g. who owns which wallet); and (3) to impose strong non-trivial constraints on data being stored (e.g. token supply invariance).

Decentralized programming requires a new type of database programming.

Since blockchain programming is essentially database programming, the language model must emphasize atomic transactionality, effective persistence, bulk querying, and convenient data modelling. To date, these categories of features were the focus of database programming: relational, graph, document-based, and others.

However, the fine grain permissions and restrictions required in decentralized systems make existing database programming techniques unsuitable. In the blockchain context the emphasis is not on bulk data, but on each data item individually. The database layer acquires significant complexity, because the entire middle layer of an application must operate under strong yet granular network constraints.

We do not take lightly the decision to design a new programming language (as fun as it is to do this). But at the same time, we believe that the tools we offer are sufficiently suited to the problem space, that the effort the reader will spend understanding them will be rewarded handsomely.

There is only so far one can go in solving a new problem with familiar tools.

4.2 Overview

At the heart of ADAPT's programming language, MUFL,[8] lies the notion of mutable functions. It establishes the triplet relationship *"function-parameter-result"* between data items. This foundational way of modeling data drives many aspects of ADAPT: the form of the programming language, data storage conventions, query mechanics, data encoding protocols, and the extensibility APIs. While simple, the mutable function approach is sufficiently generic to describe any data structures and relationships.

Mutable functions represent the foundational "function-parameter-result" relationship...

MUFL is a functional language,[9] and establishes code in the form of lambda-constructs as data within ADAPT databases. By storing code with data, we allow the same mechanism to manage both, which simplifies the task of programming governance and upgrade policies. This also allows treating access to code the same way as access to data, giving rise to possibilities such as pay-to-run code.

...which is sufficiently generic to express any data structure.

Database contains both data and code.

MUFL is easily extensible through binding of natively implemented data structures. It is through such bindings that a system implemented in MUFL can achieve the performance characteristics necessary for a large multi-user application without compromising security.

Fully extensible via natively implemented features.

Mutable functions are a construct that drives all database features of MUFL. All data is presented as functions the values of which can be explicitly mutated (see

[8] Pronounced "muffle".

[9] I.e. loosely based on the ideas of lambda calculus.

the tutorial in Section 4.3). Mutating a function is like saying "assign value y to $f(x)$," which causes the value of $f(x)$ to subsequently return y until mutated again or deleted. This primitive is fundamental to accessing and modifying any data in ADAPT databases: arrays, tables, records, lists, strings, etc.

All data is represented as functions that mutate.

The way to control permissions and data constraints is based on restricting access to variable names inside a module. The compiler ensures proper access to hidden data. Mutations of restricted data are thus confined to the code inside the module that manages it. The publicly exposed code can perform necessary checks, such as ensuring the validity of a cryptographic signature, prior to invoking private code or mutating private data.

Data access control achieved through compile-time restrictions on name visibility.

Similar restrictions can be applied to reading data from within smart contract code.[10] Modules in ADAPT databases are subject to validity constraints imposed by the compiler. In this context, visibility of data for reading can be easily restricted. This type of restriction is novel in blockchain and enables interesting new features, such as requiring payment for access to certain valuable data or code.

Permission data for both reading and writing!

What follows is a basic MUFL tutorial with some discussion of the MUFL API and its data representation. Rather than discussing the architecture of the system in abstract terms, we let the language speak for itself and hope readers bear with us.

4.3 Basic Tutorial[11]

At this stage of the language presentation, we avoid introducing the type system and several other higher level constructs. We will return to these topics later.

First, basic constants and name binding:

One-time name binding.

```
a IS 1.          Bind value '1' to name 'a'.
                 Dot terminates statements, com-
                 ments are C++ style.
b IS a.          Bind name 'b' to the value of 'a'.
c IS "Alex".     String literals in quotes.
e IS NIL.        NIL is a special value, also used as
                 boolean false.
f IS TRUE.       TRUE is another special value.
```

Basic operations on mutable functions (more on this later):

[10] With respect to reading data, we distinguish between two different contexts: on the one hand, anyone can download a public database in its entirety, and can therefore analyze and retrieve any data stored within it; on the other hand, data can be accessed from within a process or module running under network consensus. Obviously, no one can prevent people from seeing data inside a public database. However, when accessing data from the database code, reading can be restricted just as easily as writing.

[11] It may make sense for the reader to examine the example in Section 4.7 in parallel with the concepts introduced in this section

```
d IS NEW.                       Bind name 'd' to a new abstract
                                point.
                                Points serve both as values and as
                                mutable functions.
d 1 BECOMES "hello".            BECOMES mutates the value of
                                mutable function 'd' at point 1.
print (d 1) "\n".               The '(a b)' syntax is 'a(b)' in math-
                                ematical notation: evaluate a at pa-
                                rameter value b. This will print
                                "hello".
d 2 BECOMES NEW.                Now '(d 2)' is also a new point.
d 2 1 BECOMES "goodbye".        Currying syntax '(a b c)' is a(b)(c)
                                in mathematical notation.
print (d 2 1) "\n".             Prints "goodbye".
d 2 1 DELETE.                   Delete the point in the function (d
                                2) which we have previously set to
                                "goodbye"
```

Mutable functions allow changing and retrieving data at their points.

Modules and methods:

```
MODULE test {                   Defines a module called 'test'.
    a, b, c ARE NEW.            A shortcut for 'a IS NEW. b IS
                                NEW. c IS NEW.'
    d IS FUNC TAKES x, y, z
        RETURN x + y + z.       FUNC TAKES params RETURN
                                expression.
    e IS FUNC TAKES x, y DOES { FUNC TAKES params DOES
                                {transaction}
        RETURN x + y.
    }
}

print (test::d 1 2 3) "\n".     Prints "6". Note that
                                (test::d 1 2) will return a
                                function of a single argument –
                                that's how currying works.
print (::test::e 5 20) "\n".    Prints "25". Namespaces work ex-
                                actly like C++.
```

C++ like namespace syntax is used to access names in other modules. Modules help organize code. FUNCs are conventional functional constructs.

Hiding names is a primary mechanism for protecting data.

Arrays, sets and pattern matching:

```
arr IS ["alex", 2, 3].                  An array.
print (arr 1) "\n".                     Prints "alex". Arrays are mutable
                                        functions int to value.

dict IS ("hello"->"world",
    "joker"->"batman").                 A dictionary.
print (dict "hello") "\n".              Prints "world". Dictionaries are
                                        mutable functions.

company IS ("AlexB", "Jake",
"Oleg", "AlexF").                       Sets are mutable functions mapping
                                        a value to a boolean.
print (company "AlexB") "\n".           Prints "TRUE".
mixed IS ("A", "B", "C"->1).            Sets and dictionaries can mix.
BIND [a,b] TO [1,2].                    Array pattern matching. Equivalent
                                        to "a IS 1. b IS 2."
BIND ("hello"->a) TO ("hello"->1).      Dictionary pattern matching.
                                        Equivalent to "a IS 1".
```

Arrays, sets, and maps are just mutable functions. Sets and maps can be mixed together. Pattern binding quickly unpacks simple data.

4.4 Mutable Functions

MUFL database features are built around the notion of mutable functions. The above run-through of the basic language used mutable functions in several places, both explicitly and implicitly. In this section, we will explore this concept a little more.

The following brief tutorial presents how one uses mutable functions to create a data structure representing a relational table. The `BECOMES` keyword mutates a point of the virtual function. `a b BECOMES c` mutates function `a` at point `b`. (The code below still avoids constraints and type information.)

```
records IS NEW.                         Create a mutable
                                        function to represent
                                        the table.
rec1, rec2 ARE NEW.                     Create a couple more,
                                        this is a syntactic
                                        shortcut.
rec1 "name" BECOMES "John".
rec1 "email_address" BECOMES "john@address.com".
rec2 "name" BECOMES "Bob".
rec2 "email_address" BECOMES "bob@somewhere.io".
records "john@address.com" BECOMES rec1.
records "bob@somewhere.io" BECOMES rec2.
print (records "john@address.com" "name").  This will print
                                        "John".
```

Mutable function created with NEW. Data is stored by mutating the function with BECOMES. Evaluating the function later will return the data.

Obviously, in order to make tables and records broadly usable, this data structure would have to be wrapped into a module that exposes it in a convenient way,

Any natively implemented data structure can expose itself as a mutable function.

rather than accessing every mutable function directly. The example further in this paper demonstrates such API. In Section 5.1, we discuss the way performant data structures of any complexity can be presented to the MUFL interpreter as a mutable functions.

In general, mutable functions can serve to represent a wide variety of data structures. In the basic tutorial, we saw the following syntax:

```
a IS [1,2,3,"hello"].
```

Which is really the same as

```
a IS NEW.
a 0 BECOMES 1.
a 1 BECOMES 2.
a 2 BECOMES 3.
a 3 BECOMES "hello"
```

which shows a way to represent an array as a mutable function.

4.5 Scanning

If we see mutable functions as indexed collections of their points, we can envision a way to iterate over those collections. This ability gives rise to MUFL scanning, a way to query large data sets.

Scanning is the way to query data sets built as mutable functions.

```
a is [["a","b","c"],["d","e","f"]].
SCAN a BIND i THEN BIND j HAS value DO {
    print "i = " i ", j = " j ",
        value = " value "\n".              Will print all elements of the
                                           matrix.
}
SCAN a BIND i THEN WITH 2 HAS value DO {
    print "i = " i ", j = 2,
         value = " value "\n".             Will print all elements (x,2),
                                           i.e. "b" and "e".
}
SCAN a BIND i THEN BIND j HAS value
    WHERE (i == j) DO {
    print "i = " i ", j = " j ",
       value = " value "\n".               Will print all elements (x,x),
                                           i.e. "a" and "e".
}
```

Applying this to our relational structure in Section 4.4 on page 21, we can now find all records where the field "name" is set to "John":

```
SCAN records BIND _ HAS rec WHERE (rec "name" == "John" ) DO {
    print "This John has email " (rec "email_address") "\n"
}
```

Or we could've done this more simply:

```
SCAN records BIND email HAS rec WHERE (rec "name" == "John" ) DO {
    print "This John has email " email "\n"
}
```

Finally, note that we can bind patterns here:

```
a is [["a","b"],["e","f"]].
SCAN a BIND i HAS [first, second] WHERE second == "f" DO {
    print first ", " second "\n".
}
```

A useful feature in scanning is integer sequences. Integer sequences are synthetic arrays of numbers constructed iteratively.

```
seq1 IS (1..10).        Sequence from 1 to 10 inclusive, same as
                        [1,2,3,4,5,6,7,8,9,10].
seq2 IS (1..10 ++2).    Same as [1,3,5,7,9]
seq3 IS (10..1 --1).    Reverse sequence [10,9,8,7,6,5,4,3,2,1]
```

Integer sequences are synthetic arrays designed for efficient looping on integer numbers.

With scanning and integer sequences you can write loops

```
SCAN (1..10) HAS i DO {
...
}
```

Substituting a SCAN sub-clause for a binding in a SCAN statement, you narrow the scope of the parent-SCAN.

```
chars IS ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", ... , "z" ].
SCAN chars WITH SCAN (3..7) DO HAS char {
    print char.
}
```

Will print `"defgh"`.

Integer sequences are an interesting case study in implementing MUFL API described in Section 5.1. Integer sequences are not true arrays, because they only allow scanning, but not lookups. So an expression `((1..10++2) 3)` may produce

an error instead of evaluating to value 7, while the SCAN loops will work just fine. This is because the MUFL API calls for lookup and for iteration are different, and the developer of the plug-in has a choice of which one to implement. While it is possible to implement lookups for sequences, we chose in our current implementation to forego this feature as unnecessary, leaving the implementation of the integer sequence datatype lightweight, but limited.

4.6 Type System and Introspection[12]

To understand the MUFL type system, we note that the state known at compile time of a program itself constitutes a database we call metabase. Access to the metabase that respects name hiding and some other basic constraints gives MUFL its introspection and metaprogramming features.

Access to compiler's state, its metabase, provides introspection and metaprogramming features.

Consider the following definition of a new mutable function:

```
a IS NEW mutable_function string integer.   Defines a mutable function that takes a string and returns and integer.
```

We assume here that `mutable`, `string`, and `integer` are items from the metabase. In order to define new types, we need a syntax that allows us to access the metabase:

```
META {
     int_dictionary IS mutable_function string integer.
}
a IS NEW int_dictionary.
```

High-level types are metaobjects. They live inside the metabase.

Here is what this means: `mutable_function` is a function defined in the metabase. Its result is a new value, again in the metabase, which represents the type of mutable functions that take a string and store an integer value. Obviously, `string` and `integer` are existing primitive types defined as objects in the metabase.

Let's look at another example.

```
META {
     short_string IS fixed_string 25.
     long_string IS fixed_string 256.
     person_record IS record ("name"->short_string,
         "email_address"->long_string).
     person_table IS mutable_function long_string person_record.
}
person IS NEW person_record.
people IS NEW person_table.
```

What we have just done is define the type system representation for our relational table from Section 4.4.

[12]This section is work in progress, please treat it as such.

Obviously, the context tells the compiler whether the name being referenced is a run-time name or a compile-time name. What appears after `NEW` is assumed to be a name in the metabase that exports some kind of a constructor function.

There is much more you can do with this. For example, given enough hooks into the compiler's database, you can create a type modifier that prevents data from leaking outside of a given module, or ensures proper checks for consistency and correctness at compile time.

4.7 Token And Wallet Contract

Diving into MUFL further, we now present a full-fledged (albeit simplified) definition of a secure data structure that manages a supply of scarce tokens in a set of wallets protected by cryptographic signatures. The implementation is fully modular. It contains three modules: (1) `signed_gate`, which defines a generic way to protect a function call with a cryptographic signature; (2) `cells`, which defines a generic mechanism to handle scarce funds; and (3) `wallets`, which puts the two together to create a secure wallet that can only be accessed through cryptographic signatures.

MUFL allows for modular high-level implementation of basic primitives, which in other systems are implemented at the platform level.

The first two modules are internal library modules. They are available to code running inside the MUFL database, but not to the outside world. The third module exposes an API for sending transactions into the database. Code management of a MUFL database is outside the scope of this basic tutorial, but we assume that there are many options, such as (1) anyone can run code, or (2) nobody can run code, but can only access the database through APIs exposed to the outside world. We note that the simple implementation presented here is secure in both cases, assuming new code may not modify existing modules.

This example implements Gates (secure access control), Cells (funds management under invariant supply), and a Wallet (a combination of the two).

First comes the module that defines cryptographic gates. A gate is a structure that holds a reference to some functionality and only permits invoking this functionality after checking that the cryptographic signature matches that of the designated signatory. Methods are provided to create a gate given a lambda and its signatory, as well as to invoke the gate with a signed transaction.

Gates are a general construct that allows protecting any data or function with a cryptographic signature.

```
MODULE signed_gate {
```

It starts with a public type definition for `t_gate_id` which is a `unique_id`. Of `unique_id` it is assumed that it is a data type that represents a hexadecimal string of sufficient length to hold a public key. We will see further down that we use `unique_id` and `cryptography::t_public_key` interchangeably. The assumption is that the basic type library for MUFL contains both, and they do the right thing.

Cryptographic primitives are provided by the toolkit.

```
    META {
        t_gate_id IS unique_id.
    }
```

The module contains `HIDDEN` code. This is like C++ `private:`, it defines names not accessible from outside of `MODULE signed_gate`.

Data hiding is achieved through name hiding.

```
    HIDDEN {
```

And defines some internal data types:

```
        META {
            t_gate IS records::type (
                "lambda" -> function [any] NIL,
                "signatory" -> cryptography::t_public_key,
            ).

            t_coll_gates IS mutable t_gate_id t_gate.
        }
```

HIDDEN collections are only accessible by code inside the current module.

The type `t_coll_gates` is the type of a set of records (collection) we will store in the database. Next, we actually define the name that refers to the data in the permanent storage of the database that will hold this type:

```
        tbl_gates IS NEW PERSISTENT t_coll_gates.
```

Next is a function that creates a new gate:

```
        create_gate_internal IS FUNC TAKES
            lambda TYPE function [any] NIL,
            signatory TYPE cryptography::public_key
        DOES {
            gate IS NEW t_gate (
                "lambda"      -> lambda,
                "signatory"   -> signatory,
            ).

            gate_id IS NEW t_gate_id.
            tbl_gates gate_id BECOMES gate.

            RETURN gate_id.
        }
```

Many things are happening here. `lambda` is defined as a functional type that takes any argument and doesn't return anything. `gate` is created as a new record of type `t_gate`. `gate_id` constructs a new unique numeric identifier. `tbl_gates` is updated to hold this new record.

Note that numeric identifiers (such as `gate_id`) are different from `point` identifiers in that they are transparent and can come from the outside world in the form of numeric values, whereas points are opaque and can only be retrieved from the database via valid means. `tbl_gates` is a point (all mutable functions are first and foremost points), and so one is unable to access it unless one already holds it. Since this module already holds the reference to `tbl_gates` via its name, the code inside

the module is permitted to access it, but not the code from any other module in the system.

Next is the function that, given the gate identifier, will invoke the gate after checking the signature against its known signatory.

```
        invoke_gate_internal IS FUNC TAKES
            gate_id TYPE t_gate_id,
            arguments TYPE any,
            signature TYPE t_signature
        DOES {
            gate IS tbl_gates gate_id.
            signature_valid IS
                ::cryptography::verify_signature
                    (gate signatory) transaction signature.

            IF NOT signature_valid THEN ABORT.
            gate "lambda" arguments.
        }
    }
```

Security by name hiding requires that some data cannot be accessed in any way other than by legally accessing its name.

Note that nobody outside of this module can get hold of the actual record for the gate. Access is limited to the functions provided. `invoke_gate_internal` is one such function. Given the identifier for the gate it retrieves it from the database, checks that the signature matches the required public key and then proceeds to invoke the lambda with the parameter[13].

This concludes the `HIDDEN` section. Next is a piece of generic code that exposes these functions to other modules and introduces better type checking. It emits a new sub-module that properly matches the lambda's argument type to the lambda's type. This aspect of the language is not finalized, however, we wanted to provide it as the demonstration of MUFL metaprogramming ability.

```
    META {
        setup IS FUNC TAKES
            t TYPE meta::type,
            name TYPE string
        DOES {
            EMIT MODULE name {
                create IS FUNC TAKES
                    lambda TYPE function [t] NIL,
                    signatory TYPE cryptography::t_public_key
                RETURN
                    private::create_gate_internal lambda signatory.

                operate IS FUNC TAKES
```

[13]Security via name hiding is not my idea. It was first explained to me by Lucius Gregory Meredith of the RChain project.

```
                    args TYPE t,
                    gate TYPE t_gate_id,
                    signature TYPE cryptography::t_signature
                RETURN
                    private::invoke_gate_internal
                        gate args signature.
            }
        }
    }
```

Metaprogramming features are not finalized.

This proposes a feature `EMIT MODULE` that permits the user to construct code parameterized by types or values. Another module calling `setup` will receive a sub-module whose types are properly checked.

Next is the module that defines generic cells holding tokens. We assume for simplicity's sake that there is only one type of tokens in this network. A cell is a data structure that contains a number representing the amount of funds held inside the cell. Cells are not cryptographically protected unless wrapped into a gate, which is something we will see done below. This design is useful because it enables passing a cell from one party to another via its restricted reference. This could be useful, for example, to supply funds to someone else, where the total amount of funds needed is not known, but is bounded. The sender would wrap funds into a cell and pass that cell to the recipient. After the recipient deducts the correct amount of funds, the sender may retrieve the remainder.

Cells are a general mechanism for holding funds, ensuring invariant token supply, but no security.

```
MODULE cells {
    HIDDEN {
        META {
            t_amount IS decimal.
            t_cell_id IS cryptography::t_public_key,
            t_cell IS records::type (
                "holdings"      -> t_amount,
                "dkey"   -> cryptography::t_public_key,
                "wkey" -> point
            ).

            t_coll_cells IS mutable t_cell_id t_cell.
            t_wkey IS point.
            t_coll_cells_by_wkey IS mutable t_wkey t_cell.
        }

        tbl_cells IS NEW PERSISTENT t_coll_cells.
        tbl_cells_by_wkey IS NEW PERSISTENT t_coll_cells_by_wkey.
```

The idea here is that every cell has two different access points: `dkey` and `wkey`. `dkey` is used for deposit, while `wkey` is used for withdrawal. That's why `wkey` is defined as a `point`: we don't want people to find out its number from examining

the raw database data and then accessing it without permission. `dkey` on the other end is just an address and can be supplied by anyone.

Next is an internal function used to create a cell. We see that it takes the initial amount of funds. This feature is only used to generate the initial token supply, however. In all other situations, zero is passed for the new cell. Also note the function performs a check and refuses to proceed if the cell with this identifier already exists.

Cells have two handles – one for withdrawal that you hide behind a gate, and one for deposit.

```
create_cell_internal IS FUNC TAKES
    cell_id TYPE t_cell_id
    amount TYPE t_amount,
RETURNS t_wkey
DOES {
    IF tbl_cells cell_id ~= NIL THEN ABORT.

    wkey IS NEW t_wkey.

    cell IS NEW t_cell (
        "holdings"      -> amount,
        "dkey"          -> cell_id,
        "wkey"          -> wkey
    ).

    tbl_cells cell_id BECOMES cell.
    tbl_cells_by_wkey wkey BECOMES cell.
    RETURN wkey.
}
```

Transfer function takes the `wkey` for its `from` parameter. There is a number of checks here required for the validity of the operation.

```
transfer_internal IS FUNC TAKES
    from    TYPE t_wkey,
    to      TYPE t_cell_id,
    amount  TYPE t_amount
DOES {
    IF amount < 0 THEN ABORT.

    from_cell IS tbl_cells_by_wkey from.
    IF from_cell == NIL ABORT.

    to_cell IS tbl_cells to.
    IF to_cell == NIL THEN ABORT.

    IF from_cell "holdings" - amount < 0 THEN ABORT.
    IF to_cell "holdings" + amount < to_cell "holdings"
```

```
                THEN ABORT.

            from_cell "holdings" BECOMES from_cell "holdings"
                - amount.
            to_cell "holdings" BECOMES to_cell "holdings" + amount.
        }
```

Finally, we create the initial token supply and conclude the `HIDDEN` section. In a real implementation, there would be multiple token types, and this would be a function. Here for simplicity we create the supply explicitly once. This is obviously a bug, since we are losing the `wkey` of the initial supply, so nobody can ever transfer any tokens. We leave fixing it as an exercise to the reader.

```
        create_cell_internal
            "0x523A9BfA794BC2D5Bd36768c9447Da4a29624c50" 1000000.
    }
```

In the public section of the module, we expose some of this functionality to its clients. The first function is a simple helper that retrieves the `cell_id` by its `wkey` (the opposite would obviously not be allowed for security reasons). This will conclude the `cells` module.

```
    get_dkey IS FUNC TAKES
        wkey TYPE t_wkey
    RETURNS t_cell_id
    RETURN tbl_cells_by_wkey t_wkey "dkey".

    create IS FUNC TAKES
        cell_id TYPE t_cell_id
    RETURN create_cell_internal cell_id 0.

    transfer IS FUNC TAKES
        from    TYPE t_wkey,
        to      TYPE t_cell_id,
        amount  TYPE t_amount
    DOES {
        transfer_internal from to amount.
    }
}
```

Adding a gate to the cell's withdrawal handle makes it into a wallet.

Finally, we combine the two library modules we created so far into a wallet framework that is accessible from outside the environment controlled by the MUFL compiler. `wallets` is the module that exposes some of its features to external parties, potentially making these accessible via RPC or other similar means.

```
MODULE wallets {
    HIDDEN {
        META {
            t_coll_wallet_gates IS mutable t_cell_id
                signed_gate::t_gate_id.
        }

        tbl_wallet_gates IS NEW t_coll_wallet_gates.
    }

    META {
        t_wspec IS records::type (
            "from" -> t_cell_id,
            "to" -> t_cell_id,
            "amount"  -> t_amount,
        )

        signed_gate::setup t_wspec "gate".
    }
```

Note the last line above: it calls the `signed_gate`'s `setup` function that will construct the sub-module `gate` inside `wallets`. Further down we will see calls to `gate::create` and `gate::operate`. This is where they refer to.

The following public function creates a wallet as a combination of a cell and a gate. The gate hides within it the `wkey` for the cell and imposes a requirement to supply a valid signature for withdrawal transactions. The wallet itself is then exposed to the world, through its numerical identifier (which is identical to its public key).

```
    create_wallet IS FUNC TAKES
        signatory TYPE cryptography::t_public_key
    DOES {
        wkey IS cells::create 0 signatory.
        transfer_func IS FUNC TAKES args TYPE t_wspec DOES {
            IF t_wspec "from" ~= cells::get_dkey wkey THEN ABORT.
            cells::transfer wkey (args "to") (args "amount").
        }

        gate IS gate::create transfer_func signatory.
        tbl_wallet_gates signatory BECOMES gate.
    }
```

Each database exposes an external RPC API for clients to use.

Finally, we construct the access points that an external party may invoke. We leave out the topic of DOS attack protection from this tutorial, but obviously MUFL applications must implement additional controls of these transactions to prevent intentional system overload. First, we define the types that will be supplied to the API access points.

```
    META {
        t_API_create_wallet IS records::type {
            "signatory" -> cryptography::public_key
        }

        t_API_transfer IS tuple [t_wspec, cryptography::signature].
    }
```

Next, we define the access points themselves. Type `API_entry_point` is a special type that interacts with the network interface of the node running MUFL and establishes a listener.

```
    create_wallet IS NEW API_entry_point t_API_create_wallet.
    transfer IS NEW API_entry_point t_API_transfer.
```

Next, we define what the API points actually do when invoked. Here, `"invoke"` is a field of the record representing the API access point. `"sanitize"` is assumed to be a standard method that every valid type has: it will `ABORT` if an error is detected.

```
    create_wallet "invoke" BECOMES FUNC TAKES arg DOES
    {
        t_API_create_wallet "sanitize" arg.
        BIND ("signatory"->signatory) TO arg.

        create_wallet signatory.
    }

    transfer "invoke" BECOMES FUNC TAKES arg DOES
    {
        t_API_transfer "sanitize" arg.
        BIND [wspec, signature] TO arg.

        BIND ("from"->from) TO wspec.

        gate IS tbl_wallet_gates from.

        gate::operate gate wspec signature.
    }
}
```

This concludes the example, which represented a simplistic (but functionally complete) version of a database that implements economic interactions between participants. The database implemented here, if launched on top of a functioning consensus network, and if additionally ensured against DOS attacks and vetted for security, would provide a fully functioning payment system on a scarce token supply.

5 Architecture

This section will examine the ADAPT toolkit from the architectural standpoint. In summary, the mutable function approach implies interesting architectural choices based on the fact that mutable functions are a consistent and uniform construct, allowing a broad variety of data structures and components to be implemented through a minimal set of fundamental primitives we call the MUFL API. After having presented the API as the centerpiece, we will see how the MUFL compiler can use it to tie together all components of the system.

5.1 Data Access API

A completely generic representation of mutable functions is one which uses a triplestore table, where the function is the first column, the argument is the second column, and the result is the third column. Let us return now to the relational data structure defined in Section 4.4 on page 21. It yields the following table of data:

Data represented as mutable functions can be stored generically as a triplestore.

records	"john@address.com"	rec1
records	"bob@somewhere.io"	rec2
rec1	"name"	"John"
rec1	"email_address"	"john@address.com"
rec2	"name"	"Bob"
rec2	"email_address"	"bob@somewhere.io"

Here `records`, `rec1`, and `rec2` are opaque unique identifiers we call `points`, of which it is only known that they are distinct from each other. For efficiency, the triplestore table should be searchable by the first column and iterable over the second column.

Plug-ins can expose themselves via MUFL API to become visible to MUFL code as mutable functions.

It is easy to see that the entire database could be represented as a single triplestore table, organized into higher-order structures by the MUFL code that governs it. Every lookup of a mutable function will perform a search in the table, and every mutation will perform an insert. This is completely generic, but not very efficient, even if searches and inserts are logarithmic (b-tree) or constant-time (hash-map). However, what's important here is not the representation but the primitive methods it must support. These are four such methods: `lookup(function, argument)`, `mutate(function,argument,result)`, `delete(function,argument)`, and `scan(function, iterator)`; and additional helper methods for transaction management: `fork` and `merge`, which will be explained later on.

A native data structure, such as a high-performance relational table, may implement these methods and thus create a MUFL binding that will be used exactly the same way as a generic mutable function represented as a triplestore. In fact, all data and all functional constructs in MUFL operate using this same API, which creates uniformity, with further optimizations possible given sufficient compile-time information.

5.2 Transaction API

Transactions are deltas on datasets.

The triplestore representation of MUFL data suggests an interesting feature: dataset stacking. Given two triplestore-based datasets A and B, stacking is the procedure whereby a new synthetic dataset $C = A|B$ is defined as all records in B plus those records in A, where the function and argument columns do not match any in B. An additional feature is that B may have records that represent deleted data.

Using the dataset above as an example, we can use stacking to construct a new dataset that updates Bob's email address. It will look something like this:

Dataset A

records	"john@address.com"	rec1
records	"bob@somewhere.io"	rec2
rec1	"name"	"John"
rec1	"email_address"	"john@address.com"
rec2	"name"	"Bob"
rec2	"email_address"	"bob@somewhere.io"

Dataset B

DELETED		
records	"b@mm.com"	rec2
rec2	"email_address"	"b@mm.com"

Dataset C

records	"john@address.com"	rec1
records	"b@mm.com"	rec2
rec1	"name"	"John"
rec1	"email_address"	"john@address.com"
rec2	"name"	"Bob"
rec2	"email_address"	"b@mm.com"

There are many uses for this feature, but, most importantly, dataset stacking of triplestore tables can be used to represent database transactions. In order to enable uniform access to transaction management part of the toolkit, we identify the methods required to implement transactionality of any data representation. It is rather simple: the `fork` method takes a data structure and generates an empty transaction for it, while the `merge` method takes a completed transaction and merges it with the underlying data (commits it), while ensuring that no conflicts are present.

It is up to the specific native implementation to decide on the specifics of how stacking is implemented. For example, for some data structures, the `fork` method might simply copy the entire data structure, overriding the parent upon commit. This is often the simplest approach, but it presents little granularity in case of partial updates, falsely signaling transaction conflicts where there are none. Alternatively, the data structure may implement a mechanism where each transaction represents a true difference between the old state and the new state (as in the triplestore example). In this approach, conflict detection may be more granular but less efficient.

5.3 Toolkit Components

Having examined the MUFL API in sufficient detail, we now turn to the overview of the toolkit architecture as a whole.

ADAPT is a modular toolkit that provides tools and components for decentralized databases. Figure 7 shows the basic types of components and how they work together.

At the top is the dApp implementation, which provides the application-specific features of the decentralized application. It uses libraries of MUFL components, types, and patterns provided by ADAPT. The MUFL interpreter and virtual machine address native builtin components through the MUFL API. Native builtins are data structures, such as record, tables, arrays, documents (e.g JSON), strings, arrays, and many others. In this category are also system primitives, such as cryp-

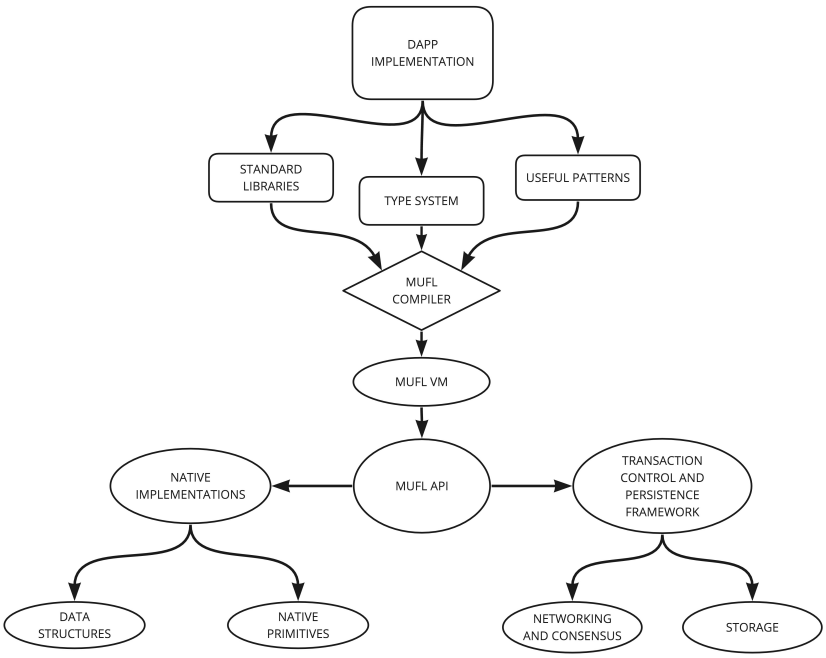

Figure 7: ADAPT toolkit architecture

tography libraries, libraries of functions that process strings and textual information, and so on.

Finally, native components interoperate with the transaction layer that provides networking and storage implementations. There are, again, multiple modules that provide functionality in this category. For example, if the database operates under consensus, the consensus protocol is part of the transaction framework. The network components accept and manage external connections and provide basic access security, if necessary.

ADAPT has a layered architecture, comprised of, top to bottom: MUFL application code, MUFL standard libraries, the compiler, MUFL API, natively implemented plugins, transaction and storage layer.

6 Decentralized Application Example: Damazone, a Two-Sided Marketplace

We now outline an example application built with ADAPT. The goal of this section is to demonstrate what a decentralized application can do with ADAPT. We outline a full spectrum of features – some commonplace in todays blockchain programming, some significantly simplified by the approach ADAPT takes, some totally impossible with other platforms.

6.1 Overview

Let's think of a dApp – a decentralized marketplace.

The application in focus is a decentralized two-sided marketplace of goods and services called Damazone. Damazone allows people to digitally present their wares, transact with each other, and accumulate reputation. Damazone has its own transaction currency and enables self-hosting of data for individual stores – pictures, descriptions, order tracking, and so on.

6.2 The Economic Layer and Crowdfunding

At the core of the system is a consensus-database we call "DamaCore" that hosts the tokens for the market, as well as identity and reputation data for market participants. Separately, users who want to offer products for sale run their own self-hosted databases representing their online stores.

Funded by users and contributors.

DamaCore is running under an economically-incentivized consensus algorithm based on a staking token called DamaShare. DamaShares are issued in limited supply at inception and are sold to those users willing to put work into staking the tokens, validating the transactions and performing required database maintenance tasks, such as running garbage collection algorithms and others. DamaShares carry no profit potential other than through contributing such efforts to the Damazone network. There may even be reasons to make DamaShares non-transferrable, at least for a period of time.

DamaCoin is the transaction currency on Damazone. In order to register their store with Damazone, users must stake DamaCoin. DamaCoin is also used by purchasers to pay for goods and by all participants for recording transactions in DamaCore. Store owners can also spend DamaCoin to promote their products in Damazone's search directory.

Has its own transaction currency.

Damazone desires to promote its economic efficiency. To ensure that DamaCoin does not become a speculative asset, Damazone imposes a negative interest rate on DamaCoin. To ensure that interest rate is applied to all DamaCoin holdings in the Damazone economy, stakers that run the DamaCore database nodes are required to record occasional transactions in users' wallets that reduce their holdings by a certain amount. This is the additional work that stakers of DamaCore will be charged with performing.

Uses a negative interest rate.

6.3 Private Data Hosting

Individual store owners spin up their electronic stores as self-hosted private ADAPT nodes that interact with DamaCore. Owners have full control of their node. They may impose access restrictions, gates, data encryption and so on. A reference implementation of such peripheral node is provided by the Damazone development team, but it can be updated freely by each individual owner. Peripheral nodes, of course, can not be trusted to hold economic value, so all economics resides in DamaCore.

Sellers are in full control of their privately-hosted stores.

Damazone supports off-chain processing through a system similar to Ethereum payment channels. Channels are represented as mini-databases shared between channel participants that describe the current channel state. Users' private databases store multiple such channels and enable the off-chain payment protocol.

Off-chain transactions are used for efficiency.

Clearly, there are many details of Damazone's model that are we are leaving out of scope of this brief outline. We intend to show how ADAPT's role as a toolkit enables the flexibility necessary to develop such decentralized models. We now turn to the approach needed to implement Damazone, and the importance of the freedom to do so.

6.4 The Role of Validators

Making sure that transaction validators are aligned with your applications is paramount. Shared networks are not able to cater to the needs of individual dApps, so the dApps' future is somewhat unknown. A particularly bad outcome, for example, would arise if transaction validators decide to exclude transactions of some application from the shared blockchain, perhaps to reduce the load on the network. In DamaZone, this would not be a likely outcome, as it uses a dedicated consensus database not shared with other applications.

Damazone seeds its validator community at the time of the crowdsale. Early supporters of the project would be able to purchase the highly lucrative right to validate DamaCore transactions, as this is the only way to mint DamaCoin. However, they are not investors, as they are required to do concrete (and onerous) work to perform the tasks beneficial to the community: validating transactions, imposing correct interest rates, and other maintenance requirements.

Having its own cohort of validators is a huge advantage.

In the future, large holders of DamaCoin may also join the ranks of transaction validators, making the network more democratic and more decentralized. However, the structure at the initial stages is designed to bring together a highly aligned group of users (rather than investors), who contribute specific effort to bootstrapping of the network.

Validators do a bunch of useful application-specific stuff, not just transaction verification.

6.5 Data Ownership and Self-Hosting

When full decentralization is achieved this will become a fully open and democratic network.

The self-hosting of data by individual store owners is not a novel idea.[14] However, providing owners with a database that speaks the same language as the central

[14] See, for example, OpenBazaar

ledger makes the task of programming such a system much simpler. It additionally enables store owners to experiment with new features.

While a self-hosted node implemented in C++ or Go doesn't preclude such experimentation, a node implemented in MUFL gives users a consistent method for building new functionality and connecting with other existing tools. We can imagine, for example, that given an open-source MUFL library that implements a personal accounting ledger, a store owner may want to include it into their node. Or the owner may additionally connect their node with other consensus databases, in order to exchange DamaCoin for another asset. This is in contrast with data hosting networks that aim to augment blockchain infrastructure,[15] which do not give individuals such broad flexibility.

An additional benefit of the architecture that includes self-hosting is privacy. Being in control of their own ADAPT nodes, users can have the flexibility to choose to whom and under what circumstances they reveal their private information. Furthermore, self-hosted ADAPT databases may store their hash-commitment in DamaCore, and reveal their information as a Merkle Tree, thus ensuring against the possibility of bait-and-switch, if desired.

Users can choose to whom to reveal their information.

While these possibilities are not unique to ADAPT, we build the framework to be particularly convenient for these types of solutions. Our most important goal is consistency of the programming model and of data representation, such that different databases may exchange information freely and reliably, with client software only required to implement one type of binding (not multiple): one for the blockchain, one for the data store.

6.6 Off-Chain Transactions

A particularly interesting aspect of decentralized architectures are peer-to-peer transactions mediated by blockchain: state channels, Lightning Network, Plasma, and others. The general idea behind these models is that participants privately agree on a sequence of transactions, which get committed to the global ledger either very infrequently, or when a conflict arises.

Peer-to-peer off-chain transactions improve efficiency.

In ADAPT, these models can be represented as yet another kind of a database, something in the middle between private self-hosted and global consensus-based. We call them Multiparty Shared Databases (MSDs). This section describes how they work.

Let's imagine that two participants, Alice and Bob, want to transact privately. To do so, they must first commit funds to a smart contract in the main consensus database. When the desired funds are committed, the smart contract creates an MSD in some initial state that reflects the individual holdings of the participants. MSD can then be downloaded and can process transactions that reflect transfers between the participants. There are only two kinds of transactions in such an MSD: (1) A transfers funds to B, and (2) B transfers funds to A.

Peer-to-peer channels are represented as temporary privately held MUFL databases that can be occasionally committed to the main database.

The MSD has two validators: A and B. It has no consensus protocol per se, except that every transaction builds on all prior ones and requires the signature of the participant who records it. Transactions are broadcast between participants.

[15] Orbit-DB, Bluzelle and others already mentioned

Having created the MSD, A and B can transact privately within it, up to the limit established by the funds originally committed. When the private transaction channel is closed, the MSD (or the most relevant part thereof) is sent to the smart contract holding the funds in the main database for final accounting. This is described in Figure 8.

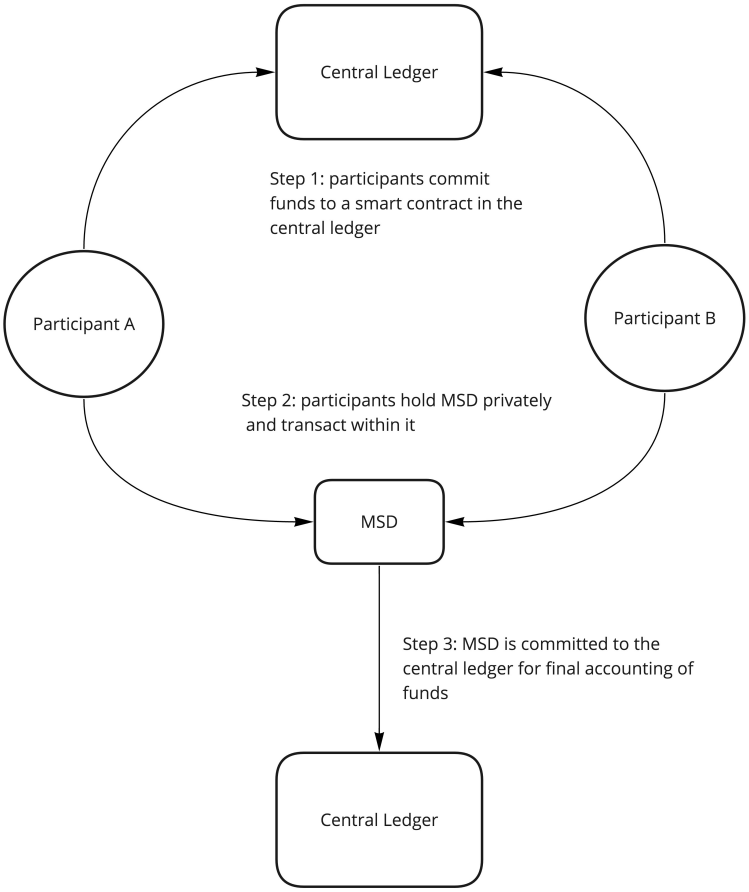

Figure 8: Lifetime of Multiparty Shared Databases representing private payment channels

Again, there are details of this process we are leaving out, such as what happens when participants disagree on the outcome, because their complexity is irrelevant to the overall architecture. But let's return to the example of Damazone and include the private transactions via MSDs into the architecture.

Like any ADAPT databases, MSDs are packets of data organized by MUFL

Uniformity of the MUFL programming model makes building peer channels particularly easy.

code through the MUFL API. In order to transact cheaply, Damazone participants may have private channels open with any number of other users. The self-hosted peripheral databases that participants maintain can store the MSDs representing these private transaction channels. The code running by the peripheral database nodes can manage all the interactions directly in MUFL. This is illustrated in Figure 9.

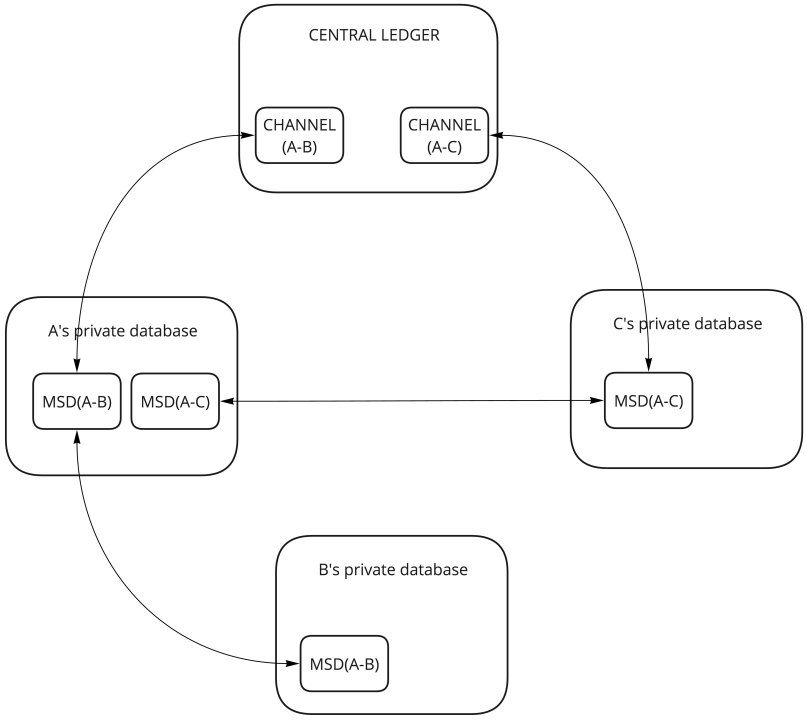

Figure 9: Multiparty Shared Databases are data items inside every user's individual self-hosted node.

The priority here is to simplify the process of programming private payment and transaction channels and to provide a framework that makes developing such models more straightforward. MUFL provides such consistency through mutable function APIs and the resulting features of dataset stacking and access uniformity. ADAPT-based architectures become versatile not because they offer a new solution to an old problem, but because the programming paradigm presents a particularly uniform way to implement old solutions in a consistent and interoperable way.

Sometimes synthesis is the best innovation. A way to do something ten times more conveniently leads to a non-linear increase in the richness of the resulting ecosystem.

7 Roadmap, Collaborations, and Funding

7.1 Implementation Sequence

The ADAPT toolkit at its core is not a complex system. The current prototype implementation of MUFL, which delivers about 60% of the language functionality is just over 8,000 lines of C++ code. The toolkit modularity is derived from the architecture in which the MUFL API exposes all the necessary components to database implementations written in MUFL. Consequently, ADAPT implementation must start with the MUFL compiler and the full specification of the MUFL API, since these are the elements that hold central importance to ADAPT as a whole.

First implement MUFL compiler and API.

We estimate that a proof-of-concept implementation that delivers a full set of MUFL features on top of a simple, centralized storage system will take three to six months. The outcome of this stage is a system that can demonstrate the process of implementing proof-of-concept multi-user database applications and economies. After this stage is completed, ADAPT should become available to dApp developers as a prototyping tool. Simultaneously, we will start building the decentralized consensus-based storage components, which will allow developers to deliver their applications as fully decentralized networks.

A dApp POC programming tool in three to six months.

There are some important components of the ADAPT architecture that have not been fully fleshed out yet. Such is the type system, the metaprogramming layer, and verification tools. Additionally, there is a significant amount of work required to implement MUFL libraries for critical components of decentralized applications: tokens, wallets, escrow contracts, exchange contracts, and so on. This work will proceed in parallel with the development of the core native components, once the MUFL implementation sufficiently matures.

Consensus algorithms can be implemented in parallel with future dApps.

The work on the type system, metaprogramming, MUFL libraries, and verification components will be ongoing.

7.2 Partnerships

We are looking to partner with groups developing decentralized applications and who understand the limitations of current architectures. Are you looking to build a blockchain network in which miners perform additional tasks? Are you looking to build a transaction recourse process into your network? Do you have unique ideas for base economics of a decentralized network? Are you struggling with fund custody on behalf of your users as a means to simplify user experience? In all these cases, ADAPT may be a solution for you. Please talk to us. We welcome both input and contributions, no matter how small.

Talk to us if you are interested in contributing.

7.3 Funding

The initial stages of the ADAPT implementation will be funded by donations. In exchange for a donation, you will receive a non-fungible asset on Ethereum to ascertain your founding contribution to future generations of ADAPT users. These non-fungibles are collectables, like pins or badges, and have no monetary value whatsoever. They will record your contribution amount and date. Keep them in your possession or gift them to friends as you see fit – they are your ticket to a "thank you" for when/if ADAPT becomes successful. Similar non-fungible assets

will be given to all contributors and participants in the early stages of development.

Our initial fundraising is capped at $400,000, which will cover the costs of the initial six months of work, with some additional expenses related to community building around the ecosystem. There is no timeline and no formal crowdsale.

Donate for a non-fungible "thank you."

8 Conclusion

Today's blockchain platforms are designed as single networks, which presents a serious weakness, which comes not from being architecturally wrong, but from being architecturally *specific*. The specificity of their designs – computational, architectural, and economic – makes such platforms/networks unsuitable for subsequent innovation by their users. Coupled with upgrade difficulties inevitable in the decentralized context, blockchain networks will suffer from the adoption conundrum: new platforms will always provide better features than the incumbents with significant adoption. This is because pre-adoption networks are free to learn from the experience of incumbents faster than the incumbents themselves.

Architectural specificity of blockchain platforms is a serious weakness.

What's worse, each new blockchain platform comes to market pre-programmed as a self-fulfilling and self-aggrandizing money machine for its creators and early supporters. This sets the stage for bitter economic wars, the victims of which are user experience, architectural freedom, and innovation itself. These wars rage in the space at the intersection of technology, economics, and politics – wars for control of the digital economic space, the landscape of which is all but clear at such early stages.

The root cause of this bitter conflict is intellectual insecurity – the misguided belief that the number of good ideas is fundamentally limited, that the digital economic landscape of tomorrow has a finite territory. We do not hold such a belief. We imagine that we can all become infinitely enriched by expanding our collective knowledge through non-competitive innovation. We do not oppose personal wealth as a goal, but it should come from innovation, not at its expense.

Why is inventing a new consensus protocol a reason for someone to be a billionaire?

Decentralization is the infinite space of innovation. Let's explore it together.

The strength of the ADAPT model is its deliberate lack of overarching underlying economics in favor of delivering tools that can be used to implement any economic model as part of the decentralized applications that use it. The framework is designed to enable innovation, not inhibit it. By synthesizing the needs of full-stack decentralized applications into an open and extensible framework, we strive to make decentralization a commodity.

ADAPT is designed to help this process without leeching value out of users.

People frequently ask me how the developers will be incentivized to build and maintain ADAPT without a token economy that creates strong monetary incentives. Yet, we very much disagree with the recent thinking that outsized rewards are the only way to build something big. There are plenty of examples of large, industry-strength open-source systems that sustain mostly on interest and the sense of contributing to something that benefits others. Additionally, there are ways to receive economic value without hurting users and supporters.

When ADAPT collects funds to pay programmers, it will be through donations only. We truly believe that project developers and users who recognize the value of what we are trying to build will want to contribute to the effort without expecting

anything in return, other than a useful product. More than money, however, we want people to contribute expertise, insight, and time.

A decentralization space offers an unlimited potential to positively affect our lives. Let's work together on making this vision a reality.

9 Social Commitment

In the days of the "ICO madness," we are founding ADAPT as a platform for innovation in decentralization and technology-moderated social systems. ADAPT will not be built on a single cryptocurrency, but will encourage every participant to create their own.

Yup...

ADAPT stands for sane decentralization. It will support all use cases, both centralized, consortia, trusted validators, and fully decentralized public topologies. It will be built in a way that allows a slow and measured transition from centralized to decentralized models and enables open and unbounded innovation by all users who wish to add components, modules, consensus protocols, infrastructure solutions, application libraries, etc. It will encourage usage in any context, be it for economic benefit, personal enrichment, or for public applications built with no economic purpose.

Thanks for reading!

Decentralization is more than just crypto.

The community that uses an open-source platform and contributes to it is the ultimate judge and jury of its value and success.

Decentralization is a destroyer of egos.

WELCOME.