Ambients

Peer-to-Peer Programs and Data

Haja Networks

2019-07-24-22a550c

ABSTRACT

The decentralized web has shown great promise for highly available, massively distributed, open, censorship-free, privacy-preserving and non-exclusive application ecosystems. The technology landscape is, however, becoming fragmented due to a lack of collaboration and safe interoperability between platforms. This fragmentation exacerbates the uncertainty, risk and cost of decentralized application development and deployment. Furthermore, competing decentralized technology projects promote application programming models that are logically centralized, removing incentives for pursuing a fully decentralized web.

To counter this problem, we introduce the Ambients protocol to create and verifiably execute distributed programs in decentralized, peer-to-peer networks. The main contribution of the Ambients protocol is a process-algebraic specification of its purely functional, safe and composable programming model. Combined with a content-addressed and location-agnostic execution model, what we present is a novel protocol for distributed computation.

A deterministic evaluation of Ambients programs is guaranteed by a novel, strongly confluent, distributed rewrite system based on ambient calculus. Deterministic verification of the rewrite system, and its integrity and authenticity, are guaranteed by recording the execution and control flow of programs as Merkle-DAG-based, partially ordered, immutable event logs. Because of these guarantees, the Ambients protocol enables building and composing truly decentralized applications, databases and services, making safe interoperability and collaboration feasible within existing networks and platforms.

Contents

I. Introduction

> *"I imagined the web as an open platform that would allow everyone, everywhere to share information, access opportunities and collaborate across geographic and cultural boundaries."* Tim Berners-Lee [59]

Background

The web started [18] as a way to share research and communicate between independent entities. Over time, as value was created and unlocked, the implementation of the web took a direction that favored centralization [56] in favor of efficiency over resilience, profit and control of markets over digital liberty. Walls have been built everywhere to keep users in, digital boundaries are hard to cross, access to opportunities is limited, a few control [48] what information we can share and how, and how we find information in the first place; and the online individual is easily tracked and compromised [68]. The current web is in trouble.

> *"Catalini and Gans defined "the cost of networking" as a problem associated with the increasing market power of the FANG group. Reduction of that cost will lead to disentanglement of the network effect benefits from market power."* *The Blockchain Effect* [55], *Catalini & Gans* [53]

Market power on the web is centralized to a handful of corporations and their platforms. This has been achieved through creating a platform and a data moat, in which the users and their data are aggregated in a single, massive database. The data is accessible to the users and developers only through a proprietary application UI, but the data is owned and controlled by the platform. Capturing the individual user, attracting their contacts and limiting competitive external developer access, so that the users can't transact outside the system, creates substantial network effects - a powerful incentive for centralization.

This consolidated market power has lead to a situation where it's hard for new businesses to compete [66] with the incumbent platforms. New businesses can't access users or their data outside the major platforms and very few can contend with the amount of resources they have. As well, new businesses can only get access to users and data by going through the incumbents, which further establishes the platforms' control over a) their users b) who can access those users and c) what the rules for access are.

This is a massive risk to entrepreneurial businesses as the rules can change at any point. This in turn reduces the number of new entrants, innovation [52] and competition. The major platforms don't have pricing or service quality pressure and they charge higher prices [31] for lower quality services. Any new business submitting to a platform can be shut down without a notice. For the end user, this ultimately means that there are less options to choose from and they're locked in. Centralization on the web has lead to data and market monopolization, with all of the ills that it creates for the market itself.

At the same time, the vast amount of data has become a liability - data gets stolen or otherwise compromised every so often. For good reason, the regulatory bodies are increasing their demands on the major platforms to take care of users' privacy. However, at other times, the regulatory bodies want and get access to the platform and their users' data. This makes the centralized platforms a lucrative environment for performing mass-surveillance and an easy target for censorship.

The situation is unsustainable and ultimately prevents innovation from happening, limiting the potential of the web. In order to protect users, enable a fair market, and encourage innovation and growth, a paradigm shift from centralized to decentralized models is needed.

The paradigm shift is achieved by *reversing the authority*.

Instead of platforms working as a gatekeeper between the user and their data and between users and other services, in reverse the users own their data and control who or which application can access it. While in the current model data is centralized around the platforms, in the decentralized model the data is "centralized" around the user.

> *"In a network of autonomous systems, an agent is only concerned with assertions about its own policy; no external agent can tell it what to do, without its consent. This is the crucial difference between autonomy and centralized management."* Burgess [6]

Instead of users asking the application if they can access and use it, the applications ask [47] the user for permission to access their data. Instead of one big database for all users, there are an infinite number of small databases, many for every user and application. Instead of having a separate account in thousands of services, self-certified identities and user profiles work across all applications. Blog posts, activity feeds, friend lists are owned by the users, who decide which service and user interface they use to access and operate on them. The user can allow their data to be used by multiple applications simultaneously, making the data re-usable and interoperable between applications and networks. Keeping local copies [33] of small databases and operating on them locally [36] is efficient and makes the user experience feel instantaneous. Applications work in disconnected environments by default.

Instead of a business requiring permission from the platforms to get access to users, they can directly communicate with the user and request access to their data. They can talk to other services through a unified common language and create new services by composing other services and data, creating emergent value and new business opportunities. An environment where businesses don't need to build and operate expensive infrastructure to acquire and keep users at high cost allows them to focus on their core offering instead of building a platform and data moat.

Because the data, users, and applications are decoupled, and because everything is cryptographically verified, the developers can build and compose applications and services swiftly and fearlessly and compete on user experience, algorithms, data insights, service quality, price, ethical values, such as respect for privacy, and more. Because the rules and access to users are not dictated by the platforms, businesses can move faster to and in the market and even small players can enter, compete, and seize opportunities.

This leveled playing field creates a new wave of innovation and decentralized applications, services, and business models. Ultimately, it is the end users who benefit from more diverse service and application offerings, better user experience, improved privacy, and lowered prices. The decentralized web will be open [37], free (as in freedom), creative, and fun.

A Glimpse of Hope

The rise of crypto-networks has created a new wave of interest in technologies that enable decentralization. Bitcoin [12] and Ethereum [20] have led the way and as a whole we've built a lot in the past few years. We've developed technologies that can be used as building blocks for decentralized systems, such as IPFS [28], libp2p [32], and most of these new technologies, protocols, and systems are open source. A variety of new, cryptocurrency-based business models and digital governance models have been created. In a short time, systems that weren't possible to build before have been conceived, implemented, and deployed.

At the core of many crypto-networks is a global ledger. The global ledger works as a "single global truth", which means that all transactions between network participants are recorded on the same global ledger. This creates a bottleneck: requiring everyone to synchronize with everyone else reduces the maximum throughput of the network as a whole. Many projects have tackled improving the overall throughput of a network, but a single stream can only flow so fast.

Most ledger-based networks have a programming interface to program the network with "smart contracts". Smart contracts are programs that run in a network by the network participants. We can create decentralized applications (dApps), make payments, run business logic, implement new protocols, and more. However, most smart contract languages and execution environments (that is the ledger they run on) are not compatible with each other [69]. This means that the developers need to write their programs in a platform-specific language, effectively needing to decide up-front in which network they wish to run their programs or services. This creates fragmentation between the plethora of networks and creates an obstacle for interoperability between them. In this regard, most of the current blockchain systems create a technological silo for the developers: you have to choose which platform to bet on before even building the application, because switching platforms means rewriting the application.

Furthermore, because the smart contract programs are tied to the underlying crypto-network and ledger, they preclude the possibility of freely developing on open, non-cryptocurrency based building blocks. That is, if a program is built on a specific blockchain platform, coins or tokens or payments are required, so the developers and users have to sign up with the network and acquire tokens for that network.

This reminds us of the problem we have with centralized platforms, and on a more broader level doesn't seem to align with the original motivations that created the whole field.

At the same time, we've become painfully aware of the realities of the centralized platforms and have started countermeasures through regulation, such as the GDPR [21] in the European Union, or even calls to break down the platform owners.

These opposing forces to centralization, especially the people and societies waking up to question the level of privacy and security the platforms offer, are a glimpse of hope. Fundamentally though, the problems still exist.

Motivations

To build and deploy infrastructure that can realize the full potential of decentralized networks, reverse the authority and decouple data from network effects [55], we think the following aspects need to be addressed:

- Asynchronous message passing
- Eventual consistency [23]
- The ability to structure large networks as smaller networks
- A decoupling of computation from the platform
- Usability in disconnected or disrupted environments

The biggest limitation to the current blockchain networks is the requirement for a single, global source of truth, the ledger, and forcing every participant to constantly synchronize and keep the global state to transact on it. While the motivations to do so are understandable (strong consistency guarantees, crypto-economic incentives, etc.), it creates an unbearable inefficiency for the network to operate.

To let infinitely diverse, offline-first applications be built on the decentralized web, the underlying protocols can not be built on a single token or network, or be dependent on them. Rather, the core protocols need to be flexible enough to build such networks on top and without requiring a payment to be able to use them. We need building blocks, not chains. The networks and protocols that require a single, global ledger are still needed and complementary to the core protocols, but the baseline needs to be as flexible, available, and efficient as possible, so that data can be decoupled from the underlying platform.

The real world is asynchronous: events occur and "things happen" disconnected from each other, at "some point in time", but rarely do we observe them *exactly at the same time*. All forms of consistency and consensus are in fact eventually consistent; messages are passed between participants, according to a predefined set of rules (the protocol), and consensus is agreed upon only when the required messages have been exchanged and the steps of the protocol were executed. While the end result, the consensus, is a form of strong consistency and requires synchronization, the underlying mechanisms are asynchronous. Messages are sent and "at some point in time" they are received and acknowledged by the receivers. From this perspective, we need to consider eventual consistency as the baseline consistency level for all data and applications. Building on asynchronous message passing and eventual consistency, we can construct stronger consistency guarantees [35] as needed.

> *"You must give up on the idea of a single database for all your data, normalized data, and joins across services. This is a different world, one that requires a different way of thinking and the use of different designs and tools"* Jonas Bonér [49]

With asynchronous messaging and eventual consistency, we gain the superpower of being able to program applications and networks that can withstand disconnected and disrupted service environments. The programs operate locally first, are always available, that is they work offline, and can synchronize with others when network connection is available. Being constantly connected is no longer needed. Overall, this leads to a better user experience than what we have today with web apps.

Every participant interacting through a single global agreement, e.g. a blockchain, creates a bottleneck, which can only be so fast. Instead of requiring disconnected and unrelated interactions to be verified by the full network (for example Alice buying a coffee from Bob and sharing photos with Charlie) the individual programs should form sub-networks. The sub-networks can be seen as "micro-networks" per application or interaction. For example, Alice sharing photos with Charlie forms a network of two participants. Or, in a chat program, a channel with 50 users forms a network of 50 participants. Dividing large networks into sub-networks per application can be seen as "built-in sharding". Each participant in a network only stores and interacts with some parts of the network, but almost never all of them.

Each crypto-network having its own execution environment and a custom programming language to program the network causes fragmentation. This fragmentation is an inefficiency that prevents network effects from forming around programs and data. The accrued value is locked in each individual network. To overcome this challenge and to unlock value, we need to decouple the program execution layer from the platform (the ledger) and realize an efficient, distributed computation model that makes it possible to run the same program in multiple, otherwise disconnected and incompatible networks.

For this purpose, we present the **Ambients protocol**. Ambients is a protocol to build and run databases and programs in a peer-to-peer network. It decouples data from the underlying platform and network and creates an interoperability layer between different networks and systems. Decentralized applications can use it to build, deploy, execute, and share code and data in a compositional, verifiably safe, and scalable way.

II. Protocol Overview

To understand the need for the Ambients protocol, let's consider the challenges that developers currently face with prevailing programming models during the paradigm shift.

The paradigm shift from platform-centric model to a decentralized one requires the decentralized applications to be equally as good or better than the ones offered by the platforms. Reversing the authority is only one of the steps. To succeed in creating better user experiences, we need to build our applications and services accordingly. The current programming models for decentralized applications and services unfortunately resemble the platforms: they use a database which is effectively centralized, often times a blockchain.

With this in mind, consider that databases are a combination of a storage (the database state) and programs to access, update and query the storage (the database services). Relational databases commonly offer SQL as an interface to update and query the database, making SQL an interoperability layer for all programs that wish to access and manipulate the database state. Smart contract platforms work the same way: blockchains are the centralized database state and smart contracts provide the database services and an interface. In a way, all centralized platforms form around a centralized database to which the data moves into. In fact, such a database model shifts the decentralized web towards the centralized platform model.

What happens when the database state is decentralized and local to its owner? The whole concept of a database is inverted: the database state no longer accumulates in one single place, but in multiple places in a network. Therefore, to have a true decentralized programming model equivalent to the traditional database-centric model, the database services (or any program) need to move to where the data is. To do so, the computation (the programs) needs to be distributed.

It turns out, solving this problem means much more than just getting decentralized databases.

Ambients protocol Summary

The Ambients protocol connects decentralized applications and networks by defining a programming model for distributed programs and a peer-to-peer distribution and computation network to run and share them.

Programs leveraging the Ambients protocol are turned into distributed executables that can be executed safely in the Ambients network. This lets the programs move where the data is. Sharing these distributed programs is essential for the interoperability and scalability of decentralized applications, for example when building the aforementioned decentralized, local-to-its-owner database where the programs form the distributed database services. The deployment and execution of the distributed programs is discussed in detail in the later chapters.

The Ambients programming model is restrictive enough to be verifiably safe. At the same time, it is expressive enough to let developers build data structures, functions, algorithms, business logic, databases, even full-fledged systems and services. Most programming languages today have an Ambients-compliant subset of features, which means that developers can build their decentralized applications and services on the Ambients protocol using a familiar programming language. The translation of a program to a distributed program using the Ambients protocol is discussed in detail in the later chapters.

The Ambients protocol is designed to be platform-independent. The Ambients network can overlay and connect multiple different Ambients-compliant runtime environments, from trusted, permissioned centralized systems (like traditional web services) to trustless, permissionless decentralized systems (like blockchain platforms). The details of Ambients-compliant runtime environments are discussed in more detail in the later chapters.

The Ambients protocol is open source, and free for everyone to build on.

Properties of the Ambients protocol

Realizing that program distribution is essential for true decentralized applications, the quality and safety of the programs become an extremely important aspect. Including third-party code is a risk to the application quality and its users, both from a functional and a security perspective, because it is impossible to know how exactly this code might behave. Thorough testing is required to ensure the quality and safety, which slows down the development and increases time-to-market. Verifying the correct behavior of any arbitrary code is arguably the "hardest problem in computer science" [67]. For any decentralized programming model to be successful, the distributed programs needs to be trustworthy and valuable.

The Ambients protocol is based on the principle that decentralized applications building on it can trust that the distributed programs are always compositional, safe, scalable, and decentralized.

The Compositionality of distributed programs guarantees the highest levels of interoperability. Compositionality is not just about modularity [41]. It means that the properties of the parts are preserved in the composition. If compositionality holds, safe, scalable, and decentralized programs can be composed together to form a new program which is also safe, scalable, and decentralized. Proving this requires a level of rigor that only mathematical constructions are known to have. In turn, that same rigor guarantees the full understanding of program behavior. This guarantee can turn an untrusted program into a trusted program and is essential for interoperability of decentralized applications and networks.

The Safety of distributed programs is an essential property for establishing the trust between applications. We define safety simply as a program behaving exactly as expected. Verifying this again requires mathematical rigor for modeling the expected behavior of programs and specifying the safety properties as logical formulas that can be checked during and after program evaluations. This is essential for establishing the trust between programs.

The Scalability of distributed programs increases the value of a decentralized network. Scalability is commonly regarded as the solution to performance issues, but it is also about ensuring that users benefit from being part of the application network. Non-scalable programs eventually become unavailable, making applications using them unavailable and denying the service from users. Scalable programs, in turn, remain available, and in the best case improve, when usage grows which increases the absolute value of the application in a decentralized network. Therefore, verifying that a distributed program scales is essential for the long-term health of a decentralized network and for the success of a decentralized programming model.

Decentralization of distributed programs is not an end goal itself, but a crucial property to enable cooperation in open and permissionless networks while allowing programs to compete. Decentralization distributes the value of programs to all participants and a protocol that is decoupled from its underlying platform allows developers and users to operate on a higher abstraction level and enjoy shared "network effects", while enabling innovation to emerge. In contrast, as previously discussed, centralized networks require to compete over users and developer mindshare, which eventually converges to the situation as it is right now. To prevent this to happen, deliberate design choices need to be made, such as requiring location-agnostic content-addressing, expecting trust to be proof-based, and guaranteeing open and permissionless network participation.

The Ambients protocol preserves these properties by specifying models for verifying the properties throughout the lifecycle of a distributed program. Detailed in the following chapters, we specify:

1. The Programming Model for translating programs to a process-algebraic representation of distributed computation
2. The Compilation Model for compiling programs to distributed executables
3. The Execution Model for deploying, executing, and sharing distributed executables in content-addressed, peer-to-peer networks

III. Distributed Programs as Ambients

The Ambients protocol defines both a programming model for distributed programs and a peer-to-peer distribution and computation network to run and share them. The programming model translates programs to a process-algebraic representation of distributed computation, which captures the meaning and expected behavior of a program when run in a distributed network, ensuring safety.

In this section we describe:

- The Programming model for Ambients programs based on immutable values and pure, total functions
- The Formal basis for the programming model which is based on a process algebra called Ambient Calculus
- The algebraic Ambients programs defined in terms of protocol primitives, using common computation abstractions as examples
- The Evaluation of Ambients programs as confluent rewrite system

Programming Model

The purpose of the programming model is to define an unambiguous translation between programs that developers write and executables that can be deployed, run, and shared in the Ambients network. This translation must retain the original meaning of the programs, which means that regardless of whether an Ambients program is run locally or remotely in the network, the evaluation must have the same end result. To achieve a deterministic end result, the programming model is based around *pure and total functions* which are evaluated to *immutable values*.

Values are like facts [58] in that they don't change. Immutable values are important in distributed systems because it's the content which differentiates a value from another, not where they are located. This property makes content-addressing possible in systems like Git [26] or IPFS [29] where values can be safely propagated and addressed around the system with negligible coordination overhead. For this reason, immutable values are the basis of Ambients programs as well.

Functions are pure if their evaluation is deterministic and does not affect any other evaluation outside themselves. Functions are total if they evaluate for all possible arguments, which also means that they must terminate. The Ambients programming model follows the total functional programming paradigm [62] and only allows programs to have pure and total functions.

Having a pure and total functional programming model has some powerful benefits [11]. Most importantly, pure functions can be evaluated independently, in isolation, with no *side-effects*. Therefore, there can be no shared state between two pure functions, making them naturally concurrent and well-suited for distributed programming. Also, if the program is a function expression, which contains only pure function calls, then the expression is *referentially transparent*. A referentially transparent function expression can simply be replaced with a computed value with no adverse consequences. Its behavior can be analyzed and verified with equational reasoning [19]. Compilers can optimize pure functions using effective techniques such as memoizing, and subexpression elimination.

However, the total functional programming model has some significant constraints. Because of the termination requirement, not every arbitrary program can be expressed using a total function. This excludes general-purpose programs with common features such as infinite loops and unbounded recursion. This is the *main limitation* of the Ambients programming model, and overcoming this limitation would require a solution to the halting problem. On the other hand, restricting the program expressivity this way lets total functional program behavior be verifiable in the first place. The totality of a functional program means that there will be *no* runtime errors [61] - if the program compiles, it is proven to work with any input, and evaluating the program will always terminate with deterministic value. Finally, as research on total functional programming [62] has shown, there are various paths forward to introduce more expressivity while ensuring the totality, like allowing structural recursion for finite data and modeling infinite evaluation with corecursion over coinductive codata. Those paths are a part of ongoing research on the Ambients protocol.

Any program that can be modeled with total functions and immutable values becomes compliant with the programming model of the Ambients protocol, and therefore can be compiled to a distributed program. However, to safely execute the distributed programs the execution itself (i.e. each step of the program) needs to be verifiable. To achieve this, we must define a formal model of distributed computation.

Process Algebra

Distributed systems are complex systems. Concurrency is key for building efficient, scalable systems, but it is also largely responsible for their complexity. Unexpected behaviors may appear even from the simplest interactions. It is next to impossible to test that distributed systems work properly. Even if we could observe the behavior of the whole distributed system reliably (we can *not*), there are endless number of different states and combinations of actions that network participants are dealing with. This is the reason for modeling distributed systems formally, as having a mathematical framework makes it possible to verify the correct behavior in *all* system states.

Correct-by-construction [7] is a design philosophy recently popularized by the CBC Casper [16] proof-of-concept. Things that are designed to be correct-by-construction use mathematical abstractions to model the thing itself and to prove its correctness. The implementation of a thing, then, needs to match the mathematical model for it to be considered correctly constructed.

There are multiple frameworks for modeling distributed computation. For example, Petri nets are a well-researched and well-suited tool for modeling the nondeterminism of concurrent behavior in static networks. The Actor model is a natural model for systems relying on asynchronous message passing with simple rules.

In the Ambients protocol, however, distributed programs are designed to be correct-by-construction by modeling their execution with an algebraic model known generally as *process algebra*.

Process algebra is a family of algebraic approaches for modeling distributed computation. Process-algebraic expressions describe a concurrent system using independent *processes* as terms and *interactions* between processes as operators. This works as a model of distributed computation because independent, parallel processes capture the nondeterministic behavior, and interactions between them become computations with well-defined reduction rules. The most significant benefit of an algebraic approach is that it makes equational reasoning possible. This is important not only for proving correctness, but also for ensuring the computability of the program, which is crucial for a total functional programming model (as previously discussed).

For example, proving the correct behavior of a distributed system is possible by modeling the system as a labeled transition system, which consists of system states and transitions between those states [63]. Process algebras define a labeled transition system where reduction rules of the algebra constrain the transitions between states. This forms a rule system for transitions between computation states, making various functional verification and analysis methods possible:

- Reachability analysis - ensuring that confidential information can only be shared between trusted parties during computation
- Specification matching - ensuring that third-party computation is equivalent with expected result
- Model checking - ensuring that logical propositions are true between any state changes

Various process algebras have been devised over the years, like CSP and π-calculus, all with characteristic properties. The ambient calculus is a process algebra with distinct properties that make it the most suitable modeling framework for the Ambients protocol (and inspired the name of the protocol, too).

Ambient Calculus

Ambient calculus, invented by Luca Cardelli and Andrew D. Gordon in their 1998 paper "Mobile Ambients" [40], introduced the concept of *mobile ambients*. The original ambient calculus has inspired many variants such as *Boxed Ambients* [13], *Push and Pull Ambients* [43], and *Virtually Timed Ambients* [2], which all extend the original algebra for various purposes. The Ambients protocol uses a variant called *Robust Ambients* (ROAM) [38] because of its safe, expressive, and intuitive co-capabilities and reduction rules.

We introduce the ambient calculus concepts along with a textual syntax for ROAM, which is used in examples throughout this paper. The same syntax is parseable by the AmbIcobjs-tool [8], which can be used to simulate the ambient programs and explore their properties and behavior.

It should be noted that as ROAM calculus is used as a *model of distributed computation*, all ROAM expressions presented with the textual syntax in this paper should be considered as human-readable representations of generated programs, not human-writable Ambients program source code. Full-scale, real-world Ambients programs, represented as ambient calculus expressions, grow too large to display in this paper and we will focus on introducing simplest building blocks that compose to bigger programs. These representations and their dynamics are useful for understanding the basis of guarantees that the Compilation Model and the Execution Model provide later in this paper.

Ambient Calculus Syntax

`P,Q ::=`	**processes**
`P|Q`	composition of parallel processes P and Q
`n[P]`	ambient n with nested process P
`M.P`	wait for action M before continuing as process P
`M ::=`	**capabilities and co-capabilities (actions)**
`in n`	can enter ambient n
`in_ n`	allow ambient n to enter
`out n`	can exit ambient n
`out_ n`	allow ambient n to exit
`open n`	can open ambient n
`open_`	can be opened

The curious reader is advised to check the full details of the syntax in the Mobile Ambients [40] and Robust Ambients [38] papers.

Ambient - The Computation Container

The *ambient* is the fundamental computation abstraction in ambient calculus. It is a computation container, with well-defined boundaries that separate an ambient from other ambients and isolate its internal computation from the outside world. Being enclosed inside an ambient, the computation has an unambiguous execution context and is not influenced by anything that happens outside the ambient. This means that the ambient calculus can model systems where programs need to have deterministic outcomes, regardless of their execution location, and can also track how and where programs are being distributed during execution.

Ambients are addressed by name. Every ambient has a name, which is used to control and authorize all actions, access, and behavior of the ambient. Two distinct ambients can share a name, which is a powerful property when modeling non-deterministic behavior of parallel processes (we'll discuss why this is so powerful in the later chapters). Once an ambient is created, there's no way to change its name while it exists, which means that names are *unforgeable*. Because of this integrity guarantee, ambient names can carry deeper meaning than just being an identifier. For example, the Ambients protocol uses names to specify type information in data structures. Using the ROAM syntax, the ambient expression describing an ambient is simply:

```
a[]
```

Here `a` is a name of the ambient and the square brackets define the *boundaries* of the ambient, everything inside them is isolated from other ambients outside `a`.

As a container, ambients form a spatial structure where they can be composed in parallel or in a hierarchy. Each ambient has an unambiguous location and an ambient can exist in parallel to other ambients or inside another ambient. Each ambient is isolated from the outside world, and therefore has no knowledge of their surrounding ambients, but they are aware of nested ambients inside them. An ambient expression describing this structure is written (in ROAM syntax) as:

```
a[ b[] ] | c[]
```

Here `a[...]`, `b[]` and `c[]` are all distinct ambients, where `a[...] | c[]` is a parallel composition of ambients `a` and `c`, and `a[ b[] ]` is a hierarchical composition of ambients `a` and `b`.

Composing ambients in such a way has powerful consequences for modeling distributed processes and inter-process relationships, because every ambient is always located either in the same or in a parallel hierarchy with any other ambient. This insight allows different evaluation strategies to be applied to differentiate when a computation is remote or local: when two ambients are composed in parallel, they are remote to each other, whereas a nested ambient is local to its enclosing ambient.

These are all examples of *immobile* ambients. The distributed system they model is also immutable. Immobile ambients are considered to be equivalent to immutable values of the Ambients programming model. The equivalence of ambients is important because referential transparency in a programming model can be verified using equivalence relations, like structural congruence and bisimilarity, which are formally provided by the ambient calculus. This guarantees that immobile ambients represent computations that are safely replaceable by immutable values during the evaluation *and* that the ambient expressions modeling a program (which is guaranteed to terminate), will reduce to an immobile ambient expression.

Ambient Capabilities

Having static and immobile ambient structures is not enough to model distributed computation. In the ambient calculus, ambients can contain not only other ambients but also *capabilities*. Capabilities can be characterized as *instructions* how to react with other ambients and as a controlled way to communicate over the ambient boundaries. The ROAM calculus extended the original ambient calculus with *co-capabilities* for additional control. Co-capabilities are dual to capabilities in that for every instruction to change or move an ambient, there must be a corresponding *authorization*, the co-capability, for a change or movement to happen.

First, the `in` capability and its `in_` co-capability define the interaction between two parallel ambients where one ambient is entering the other. This means that the following *reduction* (denoted by the arrow `→`) of an ambient expression can take place:

```
  a[in b] | b[in_ a]
→          b[ a[] ]
```

The above can be interpreted as a program of "if `a` is entering `b` and `b` is allowing `a` to enter, then `a` moves inside `b`". It's important to note that this reduction or *computation step*, as per the ambient calculus reduction rules, requires that both `in` and `in_` are consumed at the same logical time. In other words, the reduction is not complete and the computation step doesn't happen until both ambients have changed.

Second, the `out` capability and its `out_` co-capability define the interaction between the enclosing and nested ambients where the nested ambient exits the enclosing one. This means that the following reduction of an ambient expression can take place:

```
  b[a[out b]|out_ a]
→ b[               ] | a[]
```

The above can be interpreted as a program of "if `a` is exiting `b` and `b` is allowing `a` to leave, then `a` moves out of `b` as a parallel ambient". Again, for the reduction to be fully complete and the computation step to happen, it's required that both the `out` capability and the `out_` co-capability are consumed at the same time.

Finally, the `open` capability and its `open_` co-capability define the interaction between the enclosing and nested ambient where the enclosing ambient "opens the container" by removing the boundary of the target ambient, which exposes everything inside its boundaries to the surrounding ambients. In a way, the opened ambient is *dissolved*. This means that the following reduction of an ambient expression can take place:

```
  a[b[open_|c[]]|open b]
→ a[        c[]        ]
```

Here the reduction can be interpreted as "if `a` wants to open `b` and `b` allows this, then `b` and its boundaries disappear and everything inside `b` becomes nested inside `a`". Again, both the `open` capability and the `open_` co-capability must be consumed before the reduction is complete and the computation step can happen. Notably, the `open_` co-capability doesn't need a target because the enclosing ambient is the only one that can open it.

Capabilities and co-capabilities can also be defined in *paths*, which models the computation steps that need to be executed sequentially. Consider the following example of an ambient expression and its reduction:

```
  a[in c] | b[in c] | c[in_ a.in_ b.in d] | d[in_ c]
→           b[in c] | c[in_ b.in d | a[]] | d[in_ c]
→                     c[in d | b[] | a[]] | d[in_ c]
→                                           d[c[b[] | a[]]]
```

In the above example, the initial path expression `in_ a.in_ b.in d` of ambient `c` can be read as "first let `a` enter, then let `b` enter, then enter `d`". Path sequences are an essential tool for controlling non-determinism in the concurrent ambient expressions. For example, if the above example was defined using a parallel composition of capabilities instead of sequential path:

```
a[in c] | b[in c] | c[in_ a|in_ b|in d] | d[in_ c]
```

the first reduction and computation step would be a non-deterministic choice of the following:

```
#1: →         b[in c]|c[a[]  |in_ b|in d]|d[in_ c]
#2: → a[in c]       |c[in_ a| b[] |in d]|d[in_ c]
#3: → a[in c]|b[in c]                    |d[c[in_ a|in_ b]]
```

where the third option is a deadlock situation in which `a` and `b` are unable to be computed further.

Co-capabilities, which the ROAM calculus introduced, are expressive enough to model situations where concurrent processes are competing over limited resources. Like in all distributed systems, this competition has a non-deterministic outcome. As mentioned earlier, having ambients with the same name allows modeling non-deterministic behavior in parallel processes which can be observed in an ambient expression:

```
  a[in b|c[]] | a[in b|d[]] | b[in_ a]
→ a[in b|c[]] |               b[a[d[]]]
OR
→               a[in b|d[]] | b[a[c[]]]
```

In the above example, there are two `a` ambients entering `b` which has only one `in_` co-capability to be consumed, so this expression can non-deterministically reduce to either `a[in b|c[]] | b[a[d[]]]` or `a[in b|d[]] | b[a[c[]]]`. Ambients with same name are a cause of *interference*, which generally means there are unwanted, unpredictable side-effects of non-determinism when modeling distributed computations. These side-effects can range from security issues [38] to distributability issues [44]. The Ambients protocol mitigates these issues on model-level by minimizing the chance of interference by defining computation primitives based on the ROAM calculus, and during runtime with execution model construction guarantees, which enforce unique identifiers for distinct ambients.

In conclusion, the capabilities and co-capabilities transform ambients from static, immobile and immutable structures representing *values* to dynamic, mobile and mutable structures representing *computations*. As the Ambient programming model is based on functions that are guaranteed to terminate, any ambient that reduces to a *value ambient* is considered to be equivalent to a *function execution*.

Protocol Primitives

Not all mobile ambients seem to be translatable to values and functions in a way that makes sense for programs. For example, what kind of function would a mobile ambient `a[in b]` represent, or what kind of value does `hello[]` represent? We realize that to model actual values and functions and to compose them to full-blown programs, there needs be some transformation between the calculus and features present in programming models, like function arguments, evaluation scopes, data types etc. The Ambients protocol introduces a set of *protocol primitives* which provide a translation from programming constructs to an *encoding* of a program as ROAM expressions.

In the Ambients protocol, *values* are the elementary construct to which all computations reduce. In other words, the result of every computation in Ambients, is a value. The computations are represented by *protocol primitives* which consist of *computation primitives* and *distribution primitives*.

Protocol primitives are ambients which have special purpose in all Ambients programs. They are designed to assist remote and local computations with eventually converging to their final result. We define the following four primitives to encode programs as ambients:

- `func`-ambient, which creates a distributable computational context for function evaluation
- `arg`-ambient, which transfers values and functions between computational contexts
- `call`-ambient, which initiates function evaluation sequences
- `return`-ambient, which redirects remote or local code to a computational context where evaluation happens

Next, we'll define what values are in Ambients as they define the ultimate result of all protocol primitives - to encode a distributed program as a function that reduces to a value. We will then continue to define the protocol primitives.

Values

Values in the Ambients protocol are ambients that cannot be changed, which also means that *all values in the protocol are immutable*.

Informally, an ambient is a *value* if it *doesn't* have any capabilities or co-capabilities *and* all its nested ambients are values as well. Consider the following two examples of this distinction:

- `string[hello[]]` is a value because it can't be reduced any further
- `string[hello[]|open_]` is not a value, because while `string` does contain the value `hello`, the expression can be reduced further (the existence of `open_` co-capability)

Even though `string[hello[]]` is a value and can't be directly manipulated, it can be moved around if it's inside another ambient. This is an important distinction that allows values to be encapsulated into other ambients for operations, such as transformations, distribution, building persistent data structures, or signifying types, while the value itself stays immutable.

Evaluation of values just in terms of process algebra is limited. The protocol primitives do not offer any equivalence relation with structurally different but semantically similar values. This means that, for example, even if intuitively `string[hello[]]` equals the string literal `"hello"`, the encoded programs cannot combine `string[hello[]]` with `string[world[]]` in any meaningful way to form `string[helloworld[]]` to represent its string literal `"helloworld"`. Instead, equivalence is expressed by encoding programs so that they reduce to *deterministic values* which have a *deterministic runtime interpretation*. Programs may reduce either simple values like `string[helloworld[]]` or structured value expressions, like monoids such as `string[concat[left[string[hello[]]]|right[string[world[]]]]]`. The real equivalence of these value expressions is determined at runtime, as defined by the Execution Model. Because the runtime interpretation is required to be deterministic, it gives a deterministic real-world meaning to every Ambients value expression. This means that the Ambients protocol as a whole can guarantee the equivalence of program encodings `"hello" + "world"` and `"helloworld"`, which is needed for referential transparency.

The concept of a value as an algebraic expression, and its deterministic evaluation, is the core of the ambients program encoding. The Ambients Programming Model requires that programs must be pure, deterministic and total. To interpret these requirements as algebraic expressions, all programs must always reduce to a value, that is, they finish and terminate. In addition, the expressions must reduce to the same value for the same inputs, that is, they're pure and deterministic. In the next section we introduce core computation primitives which are used to encode functions that eventually reduce to values.

Computation Primitives

The Ambients Programming Model ensures that all programs will terminate, which means that their eventual end result is an *immutable value*. When encoding programs as Ambients, the final result is represented by an immobile ambient. However, being distributed and possibly highly parallel, the Ambients programs have inherent, unavoidable non-determinism, which becomes a problem when the programming model requires that programs have deterministic outputs. At the same time, programs are expected to be composable. In order to have safe, composable and deterministic encoding and evaluation of programs, the Ambients protocol defines two primitives called `func` and `arg`.

Computation Context: `func`

The `func` primitive defines a computational context for function evaluation. It establishes an *evaluation scope* and its behavior is similar to the widely established concept of *function scoping*.

Having a designated primitive for an evaluation scope allows Ambients programs to define parallel and sequential control flows that always converge to a deterministic value. As an ambient, `func` can be safely distributed as it isolates the computation inside it from other ambients, i.e. it preserves the integrity of its internal computation. In practice, this means that the runtime environment can use different evaluation strategies to decide whether a computation is evaluated locally or remotely (i.e. composed as either nested or parallel `func`s) or as a mix of both, and to track and verify the state of the computation with less complexity, in real time.

Informally, a `func` for a function `x` is defined as:

```
func[in_ x.open x.open_]
```

Here, the `func` primitive defines three, logically sequential phases:

1. Initiate the evaluation scope by allowing computation `x` to enter it with `in_ x`. This scope creates a safe addressing space for `x`, protected and isolated from other parallel computations outside `func`.
2. Evaluate the computation `x` by opening it with `open x`.
3. Reveal the computation result to the outside by allowing itself to be opened with `open_`.

The `func` above is fully reduced by the following steps (`result[]` representing an ad hoc computation result of `x`):

```
  func[in_ x.open x.open_] | x[in func.open_|result[]] |
  open func
→ func[x[open_|result[]] | open x.open_] | open func
→ func[result[] | open_]  | open func
→ result[]
```

Computation Parameter: `arg`

The `arg` primitive is used with `func` to transfer values and functions between ambients before their evaluation. This is how the protocol models function expressions with arguments. The `arg` primitive defines the *argument binding* procedure between *parameters* that are declared by functions, and *arguments* that are passed to functions in function expressions.

Informally, `arg` acts as a container for an argument `x` to transfer it to a `func` to be evaluated as parameter `y`:

```
arg[in_ x.open x.in y.open_] |
y[in_ arg.open arg.in func.open_]
```

Here, the `arg` primitive defines the binding between the argument `x` and the parameter `y` in three, logically sequential phases:

1. The `arg` waits for an argument `x`, then evaluates it, and finally moves inside the parameter `y` to be evaluated.
2. The parameter `y` waits for an `arg`, then evaluates it, and finally moves inside a `func` to be evaluated.
3. When the parameter `y` is opened inside `func`, it will evaluate to whatever value or function the argument `x` originally contained.

The composite expression above is fully reduced to a `func` ready for evaluation by the following steps (where `input[]` represent an ad hoc value of input `x`):

```
  arg[in_ x.open x.in y.open_] | x[in arg.open_|input[]] |
  y[in_ arg.open arg.in func.open_] |
  func[in_ y.open y.open_]
→ arg[open x.in y.open_ | x[open_|input[]] ] |
  y[in_ arg.open arg.in func.open_] |
  func[in_ y.open y.open_]
→ arg[in y.open_|input[]] |
  y[in_ arg.open arg.in func.open_] |
  func[in_ y.open y.open_]
→ y[open arg.in func.open_|arg[open_|input[]]] |
  func[in_ y.open y.open_]
→ y[in func.open_|input[]] |
  func[in_ y.open y.open_]
→ func[open y.open_ | y[open_|input[]]]
→ func[open_ | input[]]
```

Function Expressions With `func` and `arg`

With just `func` and `arg` primitives, we can express all pure functions. The general rule for defining function expression is to compose the function declaration with the function evaluation. This simply means composing two `func`s - the *declaration-site* which declares the parameter and the *call-site* which passes the argument - and an `arg` to bind the argument to a parameter between the two `func`s.

For example, a function expression `message("hello")` is a composition of the function definition `message(x)` which declares the parameter `x`

```
message[
  in func.open_|
  func[
    x[in_ arg.open arg.in message.open_]|
    message[in_ x.open x]|
    in_ arg.open_
  ]
]
```

and the function evaluation which passes the value `string[hello[]]` as an argument:

```
func[
  in_ message.open message.open func.open_|
  arg[
    in func.in x.open_|
    string[hello[]]
  ]
]|
open func
```

Composing these together reduces the whole program to a value:

```
message[string[hello[]]]
```

To analyze the function encodings in general, let's categorize the encodable functions by their return type and the number of parameters they have.

Functions that expect zero parameters are *constant functions*, which means that they always evaluate to the same result. Constant functions returning values are used when values need to be transformed to a function-form, e.g. as arguments to generic functions. Constant functions that return functions are the basis for locally evaluated functions. For example, JavaScript function `() => "hello"` can be encoded simply as a composition of the function definition and an evaluation without argument binding:

```
func[
  open_|
  string[hello[]]
]|
open func
```

Functions that expect more than zero parameters are generally ones that do more computation. Single-argument functions that return values are necessary for expressing transformations from input to output value. Single-argument functions that return functions enable *currying*, which is how functions with more than one argument can be expressed.

Distribution Primitives

The computation primitives encode distributed programs as ROAM expressions representing functions. In addition to function definition and evaluation, distribution of the functions is crucial for the protocol.

The Ambients protocol defines two primitives, `call` and `return`, for controlled, safe, and modular distribution of programs and data.

Request Computation: `call`

The `call` primitive allows functions to call other functions which may be local or remote. Therefore, invoking a `call` can be seen as a starting point for distributing computational workload in any program.

Informally, a function `x`, which calls function `y`, creates a `call` primitive defined as:

```
call[out x.in y.open_]
```

Here, the `call` primitive has three sequential phases:

1. Exit function `x` with `out x`.
2. Enter function `y` with `in y`.
3. Reveal the *call payload* to the function `y` by allowing `call` to be opened with `open_`.

The `call` above is fully reduced by the following steps (where `payload[]` represents an ad hoc computation payload):

```
  x[call[out x.in y.open_|payload[]] | out_ call] |
  y[in_ call.open call]
→ x[] | call[in y.open_|payload[]] | y[in_ call.open call]
→ x[] | y[call[open_|payload[]] | open call]
→ x[] | y[payload[]]
```

`return` is commonly used as a payload for `call`. The payload can also contain `arg`s, which enables partial or remote-only evaluation strategies.

Return Computation: `return`

The purpose of the `return` primitive is to include the needed instructions in a `call` to move the program control back to the *caller*, along with a result or remaining computation. Moving the control and the result back to the caller makes the evaluation of remote function possible as it's similar to the programming concept of replacing a function expression with a return value. The `return` primitive also enables declaration of functions in ROAM expression in a way that decouples them from any potential caller.

Informally, a `return` which moves the control back to a function `x` is defined as:

```
return[open_.in x]
```

The previous example, where the `payload` is replaced with a `return` primitive, is fully reduced by the following steps:

```
  x[
    call[out x.in y.open_|return[open_.in x]]|
    out_ call.in_ y
  ] |
  y[in_ call.open call.open return]
→ x[in_ y] | call[in y.open_|return[open_.in x]] |
  y[in_ call.open call.open return]
→ x[in_ y] |
  y[call[open_|return[open_.in x]]|open call.open return]
→ x[in_ y] | y[return[open_.in x]|open return]
→ x[in_ y] | y[in x]
→ x[y[]]
```

Here, the usage of `return` within `call` defines a logically sequential sequence, in addition to `call` handling:

1. After sending out the `call` to `y`, function `x` allows `y` to enter with `in_ y`.
2. After opening the `call`, function `y` opens the `return` primitive, which reveals the `in x` capability, making `y` to enter `x` for further evaluation. Due to the sequential `out_ call.in_ y` definition, any `y` will be authorized to enter `x` only once and after the `call` to `y` has moved out of `x`.

Because of this mechanism, the function `y` is unaware and fully decoupled from the caller `x` during the whole sequence, until it processes the `call` and the nested `return` and adopts the `in x` capability that redirects it back to the call-site `x`. Making the function `y` itself move, instead of creating some transient ambient representing a "return value", is a deliberate design choice which enables a variety of distributed evaluation strategies.

Evaluation Strategies

With the `call` and `return` primitives, compilers and VMs can safely use different strategies on how programs access functions and whether the programs are evaluated locally or remotely. This procedure is related to the concept of partially applied functions.

To analyze the strategies, let's consider the example JavaScript function `const plus = (a, b) => a + b` which can be called with `plus(1, 2)`. In JavaScript, the function can be transformed to an equivalent, but *partially applicable* function `const plus = (a) => (b) => a + b` which can be called with `plus(1)(2)`. The choice of returning functions instead of values is the basis for different distributed evaluation strategies.

A common pattern is local evaluation of a remote function, where a remotely defined `func` is moved to the local scope with `call` and `return` primitives and initiated with a local `arg` argument. Following the JavaScript example, this would be equivalent to a `const plus = () => (a, b) => a + b`. Partial and fully remote evaluation strategies rely on requiring the caller to include `arg`s for all or some of the function parameters within `call` before a function "returns", i.e. opens the `return` primitive.

Choosing between different evaluation strategies is a trade-off between performance and control. Local-only function evaluation relies on moving the function back to the caller once, after which it can be called repeatedly with different arguments, locally without network access. Fully remote function evaluation allows the remote function itself to control the function evaluation, similar to how the client-server and request-response based protocols work.

Ultimately, the choice whether to return a function is made by the declaring function itself and not the caller. Similarly, the available evaluation strategy options are controlled by the declaration-site. However, because all functions in the Ambients protocol are pure and total and programs are referentially transparent, the same function will return the same value when inputs are the same. This helps the execution runtime to cache functions locally and use the local evaluation more and more over time.

Computation Abstractions

With just functions and values, it is possible to compose some very useful and universal higher-level abstractions using the computation primitives. In the following sections we describe how the protocol uses ambients and the computation primitives discussed in the previous chapter to define:

- Types for having safer and more efficient computations
- Monoids which represent data structures that are combined from other similarly typed data structures
- Functors which represent data structures that describe transformations between differently typed data structures

Types

Data types are one of the most useful abstractions for data structures and computations. There's a huge body of research studying the types and type systems in programming languages. To be able to generate efficient and safe executables, practically all established programming language compilers and runtimes rely on information about the type of data. We propose a simple but effective way to define type information in the Ambients protocol.

The protocol uses value ambients in an effective way to represent types: the type is identified by the name of the ambient. This is similar to nominal type systems where equality of types is based on name or signature. For example, a string literal:

```
"hello"
```

can be represented as an ambient:

```
string[hello[]]
```

Informally, `hello` could be seen as a program which returns some immutable binary value and `string` as a program which decodes that binary value to a human-readable text.

The name of the type in itself isn't meaningful, however. To execute distributed programs like these, there needs to be both a common understanding what a type such as `string` means in the runtime environment, and a way to verify the type safety based on that data.

First, to establish the agreement about the meaning of types, the protocol runtime introduces a collection of commonly available primitive data types as a part of the runtime environment. This is discussed in detail in the Execution Model chapter.

Second, to do any type checking for the data, it needs to be in a verifiable state. This is why the name of the ambient is a meaningful type name only when it has reduced to its final state, a value ambient. Values are the end result of converged distributed computation whereas non-values are ongoing computations with non-observable and distributed state, and therefore their evaluation and consequent type checking would be non-deterministic.

Linking ambients to types has some useful properties. First, just like ambient names, the type information cannot be erased or forged in runtime which is important for type-based optimization and verification. Second, an ambient can only exist in one place at a time and cannot be copied. This makes ambient-based type system a linear type system. Linear types have been previously researched [34] as an effective way to implement fast memory management, because they make tracking the ownership of data and resources easier. In our ongoing research, we are looking into taking advantage of this to allow protocol implementations to mitigate real-world issues like implementing safe and scalable DAG compaction and garbage collection in content-addressed systems.

Types and type systems can greatly increase the safety of distributed programs with low amortized overhead and we believe the expressiveness, ergonomics, and safety of the ambient types can be improved in the future.

Monoids

Data is a computation in itself. For example, when application or database state is considered as an accumulated history of state changes in the right order, the current state at any given time is the result of a computation that reduces all of the state changes into a single data structure. Some data structures and the operations on them are algebraic structures called monoids.

Monoids are abstractions that are universally occurring everywhere in programming. A combination of a data and binary operation is a monoid if the operation's parameter types and return types are equal, the operation is associative, and there's an "identity element", like an empty value which is a starting point for the monoidal structure. For example, the following are monoids:

- Natural number addition (0 as an identity element)
- Natural number multiplication (1 as an identity element)
- String concatenation (empty string as an identity element)
- Committing a database transaction (commit as the binary function for merging two database "versions" to a new database "version", with the empty database as an identity element)

Monoids are useful because they are *the* abstraction for data structure composition. With monoids we can safely compose a new data structure from two existing data structures of the same kind. Because of this, they are useful in composing distributed programs.l

For this purpose, in the Ambients Protocol a monoid is described as a hierarchical value ambient structure, which retains the associativity of monoidal operations. Consider the string concatenation monoid as an example:

```
string_concat[
  in_ call.open call.(
    func[
      left[
        in_ arg.open arg.in string.in concat
      ]|
      right[
        in_ arg.open arg.in string.in concat
      ]|
      string[
        concat[in_ left|in_ right]|
        in_ left|in_ right
      ]|
      open_
    ]|
    open return.open_
  )
]
```

The hierarchical structure of the monoid is constructed by:

- `string_concat`, which represents the `+` function for two strings (the binary function), expecting `left` and `right` as arguments and which reduces to a `string` ambient containing a:
- `concat`, which represents the hierarchical structure, with `left` and `right` retaining the associative order. Here, the `concat`, `left`, and `right` are common operations for `string`, provided by the Execution Model

By using the computation primitives, these ambients reduce to immutable values describing the string concatenation operation of `left` and `right` values.

For example, the expression `"a" + "b"` reduces the ambients to a final value:

```
string[
  concat[
    left[string[a[]]]|
    right[string[b[]]]
  ]
]
```

And the expression `("a" + "b") + "c"` reduces the ambients to a final value:

```
string[
  concat[
    left[
      string[
        concat[
          left[string[a[]]|
          right[string[b[]]]
        ]
      ]
    ]|
    right[string[c[]]]
  ]
]
```

Note how the structure of `"a" + "b"` remains immutable, i.e. the same in both examples, which is important for building monoidal structures efficiently.

With this machinery, the whole monoid computation retains the closure and associativity properties needed for being a monoid. As the end result is defined entirely in terms of primitive types and common operations of the Execution Model, the value expression is referentially transparent and its evaluation is deterministic for all participants in the network.

Functors

Like monoids, functors are prevalent higher-order abstractions that appear almost everywhere in functional programming. Informally, a functor represents anything that can be mapped over. It is an abstract data structure which provides a way to map, i.e. transform a data structure and the data it contains, to a new similar data structure.

For instance, arrays are a good example of functors. In JavaScript, we can write `[1, 2, 3].map(x => x * 2)` which is a mapping from an array of ints to another array of ints. This transformation function can map integers to anything, but the array functor makes sure the internal structure, i.e. the order of elements, is preserved. Many familiar data structures for arbitrary data like pairs, trees, or lists are examples of functors.

The power of functors come from its properties that originate from category theory. Most importantly they obey the composition law [25]. In JavaScript, it means `["a", "bb"].map(x => x.length * 2)` is equal to `["a", "bb"].map(x => x.length).map(x => x*2)`. As compositionality is a key property and requirement for the protocol, it is important to be able to encode functors as computation primitives as well, adding to the abilities to compose data structures from other data structures.

Let's use the `identity`-functor [57] as an example, because it is a single-element container for arbitrary data, which makes it a functor without any internal structure. Therefore retaining the internal structure is a simple no-operation and we can focus just on transforming of the data. We observe there are four functions contributing to this mapping sequence in the example found in our examples-repository:

- `identity` which defines the structure
- `map_identity` which defines the mapping "implementation" function for `identity` structure, expecting execution arguments
  - `func` as a context for transformation function evaluation
  - `id` as the identity to map
- `string_length` which defines the transformation function, expecting a single argument `str` as a `string`-value for constructing `int[length[..]]`-values
- `program` itself which defines
  - `identity[string[hello[]]]` as the initial structure

Mapping an `identity[string[hello[]]]` to a new `identity[int[length[string[hello[]]]]]` by using the `string_length` transformation function is then achieved by composing the four functions together and applying the computation primitives:

1. `program` first fetches the `map_identity` to its execution context, and evaluates it, which opens holes for arguments `func` and `id`.
2. Then, `program` moves the initial `identity[string[hello[]]]` to the `id`-hole.

3. Then, `program` fetches the `string_length` function to its execution context in parallel and moves it to the `func`-hole, where `arg` primitive is used to transfer data from initial `identity` functor to `str`-hole of `string_length`, the last open hole in execution context.
4. When all the open holes in execution context are filled, the context is ready for evaluation.

After the execution context `func` is evaluated, the `program` reduces to the expected final value of a new functor:

```
identity[
  int[
    length[string[hello[]]]
  ]
]
```

This same machinery can be adapted to implement more useful functors like pairs, lists, and trees as well. Many of these are provided to the programs by the protocol runtime environment described in the Execution Model.

Computational abstractions, like functors and monoids, are good examples of the expressiveness of the programming model that Ambients protocol and its computation primitives create. However, as discussed in the programming model section, there are algorithms and data structures that cannot be implemented given the intentionally constrained programming model which limits the expressiveness. Later in this paper we discuss the future directions for Ambients protocol research, to have more expressive computation primitives and new computational abstractions, and to make more powerful runtime optimizations possible while still retaining the verifiable properties of the protocol.

IV. Compilation Model

Distributed programs need to be deployed to the network to be executed. A successful deployment requires that there's a common *executable* format that every runtime environment in the network understands.

In this chapter, we define the *compilation model*, the process of translating source code to an executable, distributed program. The end product of this process is an executable in bytecode format, ready for deployment.

Translating Ambients programs

The Ambients protocol overall is programming language-agnostic. That means almost any programming language can be used to write distributed programs, as long as there's a compiler that can process the source language and turn it into the Ambients bytecode. While most common programming languages can be used, due to the protocol primitives, functions and types, functional languages are especially well-suited to write distributed programs.

Compilation model requires all compilers to:

1. compile original source code to an intermediate abstract syntax structure (usually as in Abstract Syntax Tree)
2. translate the intermediate structure to the computation primitives, distribution primitives and computation abstractions of the Ambients protocol
3. generate the bytecode executable from the primitives

How these requirements are met is up to the compiler implementation. The compilers are free to apply a variety of optimizations and internal logic at compile-time. Generated bytecode and the correctness of the primitive translation are validated upon execution, as described in the Execution model.

Program Bytecode

It is important to ensure that programs deployed to the network keep the information hidden from the computation participants who don't need to access it. This is one of the key properties of the execution model and one of the requirements is that programs can be sliced into their parallel sub-parts, so that only a minimal part of the program is exposed to the other participants. The compilation model, and its implementation, the compiler, satisfies this requirement by producing a bytecode representation for every unique ambient and their nested ambients as the compiler output.

The program instructions, each parallel sub-part of the program (a "slice"), and their call-order, are represented as a DAG and saved to a content-addressed storage, as a Merkle-DAG, giving each program and their sub-parts a unique *hash*. Using this hash, the program can be fetched from the network and referenced by the programs. Storing the bytecode as a Merkle-DAG, we can be assured that upon fetching the program from the network, the bytecode hasn't been tampered with. By sharing the hash of the bytecode of the program, the program can be discovered in the network and included in other programs as a dependency.

The Bytecode Format

As described in the previous section, the bytecode is a sequence of compact instructions to execute the program according to the Execution model discussed in later chapters.

The bytecode is a binary format which defines the ambients and their movement as "opcodes" that are executed on "targets". The bytecode expressions are, then, a sequence of *instructions* encoded as tuples of

```
(<opcode>, <target>)
```

The definitions of the individual elements are discussed in Opcodes and Targets and the relation between the tuples, i.e. the call order of instructions, is discussed in Instruction Order. The exact bytecode format will be later specified in the detailed protocol specification. In this paper, we sketch the high-level structures and formats.

The purpose of the bytecode encoding is to:

- keep the programs as compact, efficient, and distributable as possible
- make the execution order unambiguous for easier verification, and capture the sequential and parallel instructions

In the future, there is potential to:

- Write a compiler with the protocol, i.e. verifiable compilation

- Embed a compiler in the VM (JIT-like compiler)

Opcodes

The *opcodes* capture the type of the instruction to be executed. We first define a set of opcodes for the events specific to the execution model and the opcodes for the Robust Ambient calculus terms, the capabilities and co-capabilities:

- `0`: `create`
- `1`: `deploy`
- `2`: `in`
- `3`: `in_`
- `4`: `out`
- `5`: `out_`
- `6`: `open`
- `7`: `open_`

We then define opcodes for the computation and distribution primitives of the protocol:

- `0`: `func`
- `1`: `call`
- `2`: `arg`
- `3`: `return`

Targets

We continue with the definition that the *target* in the `(<opcode>, <target>)` tuple is either the opcode for the primitive *or* the name of the target ambient. For the co-capability `open_`, the target is not used - instead, always use `0` as the target opcode. That is, `open_` compiles to `(7, 0)`.

Instruction Order

We finish by defining how the expected execution order of the instructions is captured in the bytecode. As each step in the program is represented by the tuple `(<opcode>, <target>)`, we construct an Execution DAG where the instruction tuples are the nodes of the DAG. Each instruction, a node in the Execution DAG, has an edge directed to the previous instruction of the program, which forms a causal order between them. As DAGs can branch and join, the Execution DAG captures both the sequential and parallel instructions.

For example, the program `call[out a.in b|open_]` would be represented as four execution steps:

```
("create", "call")
("out", "a")
("in", "b")
("open_", 0)
```

They would form the following Execution DAG:

```
      ("create", "call")
              |
             / \
            /   \
("out", "a")   ("open_", 0)
      |
("in", "b")
```

Using the previously defined opcodes, the instructions produced by the compiler are:

```
(0, 1)
(4, "a")
(2, "b")
(7, 0)
```

The first instruction `(0, 1)` can be read as "*create an ambient called* `call`". The second instruction `(4, "a")` maps to the capability `out a` followed by the third instruction `(2, "b")` for the capability `in b`. The `open_` co-capability is captured in the fourth instruction `(7, 0)`.

From the Execution DAG of the program, we can see how the DAG captures the sequential and parallel instructions, and divides the program into sub-parts:

```
      (0, 1)
         |
        / \
       /   \
(4, "a")   (7, 0)
   |
(2, "b")
```

V. Execution Model

Encoding programs as distributed computation primitives lets us reason about the behavior of the programs and the systems they're part of intuitively. To ensure that programs are behaving correctly in decentralized, trustless networks, the programs must be interpreted and executed on a computer, such as a virtual machine, that fulfills the execution requirements. The execution requirements and environment are described as the protocol *execution model*.

This section describes the execution model for the protocol, by defining:

- A model to encode ambients and their discrete events as a distributed logs
- A finite-state machine that represents the state of an ambient
- The constraints and specifications that all protocol implementations must adhere to
- The verification procedures for the computations and state transitions

Ambients as Logs

The protocol defines distributed programs as ambient expressions which form a set of parallel and nested ambients and a path according to which an ambient moves. A program is executed, i.e. run, by reducing its initial ambient expression to its final state. The execution of a program makes the ambient move and change its state. To capture the ambient structures and movement and to be able to verify that programs were correctly executed, we define an execution model for the protocol based on distributed logs of discrete events structured as Merkle-DAGs [65].

A log consists of discrete events, which occur and are recorded during the execution of a program. Starting from an initial state the program follows the ambient reduction rules, step-by-step, to a final state.

Every ambient in the system records its events to its own log and includes a signature to prove authenticity. During this process, every ambient records its own log, which includes a signature to prove authenticity. The aforementioned Merkle-DAG structure keeps the log partially ordered and preserves cryptographic integrity.

The state of an ambient, at any point in the execution, can be calculated from the events it has recorded in its log. It is the reductions, and in turn the events, that make ambients move and change their state. To look at it the other way around, the state of an ambient is recorded as immutable events in a distributed log. This enables us to analyze a program and its state at any given time in detail, to move back and forth between the states making, for example, time-travel debugging possible.

The recording of program execution as a log establishes the ambient structures, the expected instructions, that the correct instructions were executed and that the log contains only the expected events. This model enables any participant in the network to verify that the correct events were recorded by the correct participant at the correct time, and thus we can be sure that:

- A program was executed correctly
- A program behaved correctly

The safety properties of the protocol require that any protocol implementation and its underlying log must:

- *Partially order* its events
- Preserve the *integrity* of its events
- Enable *verification* of its events

A database that provides a log abstraction, and guarantees the aforementioned properties can be used as a storage system in a protocol implementation. In the peer-to-peer world, blockchains and some DAGs, such as Merkle-CRDTs [39], can be used as the log. For example, solutions that meet all the requirements are OrbitDB [46] and the Ethereum blockchain [20]. There are several implementations that use a similar log structure, such as "Feeds" [51] in Secure Scuttlebutt [50], "Threads" [4] in Textile [54] and "Hypercore" [27] used by the Dat [17] protocol. These may be directly usable for the execution model described here.

The execution model doesn't set any strict requirements for exchanging messages between the network participants to communicate new events. The message exchange can be implemented through various mechanisms and is discussed in more detail in the chapter Operating System and Networking Requirements.

Partial Order

The execution model requires that events in a log are at least partially ordered. Partial order means that, for some events in the log, we don't know which came first, so they are considered *concurrent*. That is, they happened at the same time and the causal order [60] between them can't be determined.

With partial ordering as the baseline consistency requirement, the execution model captures parallel execution of computation through concurrent events and stronger consistency guarantees can be achieved either through the log implementation or at the application level. For example, a total order can be derived from a partial order by giving a partially ordered log a total sorting function [45] or using a blockchain as the underlying log [24].

Most importantly, partial ordering enables eventual consistency which removes the need for participants to synchronize, reducing the consensus and coordination overhead to a minimum. This in turn greatly benefits the overall network throughput and significantly contributes to better scalability.

Integrity

Integrity of the computation and data is a crucial safety property in the execution model. The execution model requires all events to be *unforgeable* and *immutable* after they've been written to a log. As a corollary, all logs must guarantee their structural integrity, which means that the order of the events in a log must also be *unforgeable* and *immutable*.

Unforgeability can be achieved by signing each event with a public signing key of the creator of the event. The signature works as a proof that it was created by the owner of the signing key. Upon receiving an event, participants verify the signature against the event data and the given public key. If the signature verifies, the participant can be sure that the event wasn't forged.

Immutability can be achieved by structuring the events as Merkle-DAGs in a content-addressed storage system. When persisted, the content-addressed storage returns a hash to the event Merkle-DAG, which is then used to reference that event. The Merkle-DAG structure guarantees that a) any change to the event data or its references would change its hash and b) upon receiving the event its contents match the given hash. Together, these two properties establish *immutability* and *unforgeability* of the events.

Verifiability

To guarantee that distributed programs are executed and behave correctly, the execution model requires that its properties are easily and reliably verifiable. The execution model is *verifiable* if integrity and the order and authenticity of events can be verified by the network participant. This means that verification of the logs and events must be fast, simple, cheap, and reproducible for expected and unexpected behavior.

The main mechanism for verification is the event data structure and its representation as a Merkle-DAG. The event data, explained in detail in the Event Data Structure section, contains all the necessary information to verify its contents as well as its authenticity. The Merkle-DAG data structure, in turn, enables us to verify the order of the events and integrity of data and communications, that is, we can be sure that the data we received was the data we requested.

Performing the verifications at runtime is the responsibility of the virtual machine.

Logs

Logs are a unifying abstraction [64] and an indispensable tool [22] in designing and implementing distributed systems that collect multiple events into one logical data structure. Executing a program generates events that are linked together as a Merkle-DAG, forming a log. The events are hashed as Merkle-trees [3] making them cryptographically secure and giving each event an immutable content-address. Merkle-DAGs, or Merkle-trees, are a well-known and researched data structure widely used especially in peer-to-peer and crypto-currency technologies.

A log is created by defining a *manifest* object for the log. Each ambient and the events from their reductions are written to their own log.

Every event in a log has a *log identifier*. The identifier establishes which log a particular event belongs to. An identifier is distinct for every log and is assigned by the executor of a program. A distinct log identifier ties the events of a program together with the executor and protects against forged events.

Every event contains an *operation*, which specifies the ambient capability or co-capability that was "consumed" in a reduction step. For example, the events recorded by the next reduction step for a program `a[in_ b] | b[in a]` - the operations would be `in_ b` and `in a` respectively. Recording the operation in the events allows us to check the events against the expected events and to be sure that they match the expected order and protocol primitives.

Every event contains one or more links or references to the previous events in the log. Events linked together form the DAG-structure: `A ← B ← C ← D`. The links between the events define the *partial order* of events, establish the *integrity* of a log and allows traversing the log backwards in order to verify all the events in the log (that is, the full program execution). An event and its order, as part of a Merkle-DAG, can't be changed without changing the hashes of all subsequent events establishes *immutability* of the event and the log. That means that events can't be inserted to the history of a log making a log tamper-proof. Due to the movement of ambients from one ambient to another, events sometimes refer to events in other logs that form the sub-parts of a program and become part of a larger DAG describing the system and its execution.

Referencing the events all the way to the beginning of the execution establishes an isolated execution path. That is, the hash of the root event gives a unique id for the execution of the program and separates it from other calls to the same program. This construction is important as it solves the interference problem present in ambient calculus.

Every event is cryptographically signed by the executor of the program and the public key used to sign the event is included. Signing the events establishes authenticity: any participant can verify that the event was indeed produced and recorded by the log owner. The signature is created by hashing the log id, operation, creator, payload address, and refs of the event, then signing the generated hash and adding it to the signature field of the event. Referencing the previous events form the causal order, i.e. a timestamp, and including the references in the signature protects the log from replay attacks.

Parallel composition of ambients (and thus parallel execution of a program or its sub-parts) are represented as unions of Merkle-DAGs. For example, the ambient `a[b[] | c[]]` forms a DAG as a union of the two independent sub-DAGs isolated from each other. The individual DAGs can be considered as slices of the program, and subsets of the full program. This definition is important; when events are passed to other logs and participants, the receiver can only access the minimum slice of the program needed, while other parts of the program remain hidden and inaccessible to the receiver. Only the originating ambient knows all the events of the log and is able to reconstruct the full program. A receiver of an event can only reconstruct the part of the program ("slice") that the event refers to.

Events produced by the protocol primitive ambients (*func*, *arg*, *call*, *return*) are embedded in the logs of their parent ambients and they don't have their own logs. For example, events produced by executing `call` and `return` primitives are embedded in the logs of the caller and the callee. Having a separate log for each primitive would add coordination overhead between participants.

At any time, a log has a single writer which makes it clear who the executor is, who has control over an ambient and to which log they should write to, clearly separating the participants and their responsibilities. Since signing an event requires the possession of a private signing key, logs can only be written *locally*. As such, on the network level, the coordination overhead stays minimal while executing a program, and participants can make progress individually. The result is extremely low latency updates, as those happen on the local devices first. Consequently, applications have a user experience that feels faster than when an application needs to make a request to a server and wait for the response. Thus, applications can work even disconnected from the network, offline.

Events

The discrete events recorded in the logs are the result of the ambient calculus reduction steps when a program is executed. They occur at some point in time, and different events are separated in time. Events of a reduction can also happen in parallel, or *concurrently*, which results in one or more ambients recording their events at the same (logical) time. Additionally, the Robust Ambients' calculus rules define that to reduce a capability, a matching co-capability is also reduced. Hereby, there are actually two events that happen at every reduction: one event for reducing the capability (e.g. `in`) and one event for reducing the matching co-capability (e.g. `in_`). The execution model defines this as the duality of capabilities and co-capabilities.

In this section, we define a set of rules, which guarantee that the computation primitives are always recorded and verified in the same deterministic order. Even though the computation primitive ambients internally can run in parallel and have no deterministic ordering, the rules of the execution model define a specific, known-in-advance order in which the computation primitives work, and are recorded, in relation to each other. This guarantee has useful consequences:

- The events of a distributed program have causal ordering and the log forms a partially ordered set (where the causality is represented by DAGs)
- Running a distributed program will eventually converge to a deterministic end result
- The event for the end result becomes the *least upper bound* of the partially ordered set and the finished program forms a join-semilattice (for a more formal definition, the reader is advised to refer to the Merkle-CRDTs paper [39])

- Parallel computations are isolated and do not interfere with each other

Event Data Structure

An event consists of:

- An `id` that establishes which log the event belongs to
- An `op` which contains the event name, as per the bytecode format, and any required arguments for the event
- A `public key` of the creator of the event
- A `signature` to establish authenticity
- A set of `references` to the previous events in the program execution
- An optional `payload` containing the hash of computed value (included *if and only if* the program slice was fully executed)

For example, an event recorded upon reducing `in a`, which hashes to `zdpuApuYgEmSfLjSXhkhtTww78Eg9Rz5wobu2BnBqPBVSksRU`, would be represented in JSON as (*complete hashes truncated for brevity*):

```
{
  id: 'zdpuArPwAFjAJqbJYwW714H362twiSMF1TX6H5T7L...',
  op: 'in a',
  sig: '30440220264d3bab838066d856087779af511afe...',
  creator: {
    id: 'zdpuAwkLw7KAgXSEqduQQoyo9MrpkWrKDrKtBUg...',
    publicKey: '04c9680e7399c5d9589df2b62f32d568...',
  },
  refs: [
    'zdpuAmofe9Wk44ZbvMojdYPqBZ5xdrY5b8UWZmZFop4...',
    'zdpuB2UnPayCXCENwbu4bH72okXDQYfeQ8bhJk2VsPF...'
  ]
}
```

Creating Ambients

When starting a program, a `create` event is first written to the log of the program. The `create` event contains the name of the ambient which declares the existence of an ambient. For example, an ambient `a[in b.open_]` would produce an event with an op `create a`. The internal structure of an ambient, e.g. `in b.open_`, is recorded as a `deploy` event discussed in the Program Definition chapter. Subsequently, throughout the execution of the program, when a nested ambient is created, a `create` event is recorded. A log can thus contain several `create` events that were generated by the same program.

As with other events, the `create` event data structure references the previous events in the program execution, linking the nested ambients to their parent. This connection makes it possible to find the path from any event back to its parent ambient and all the way to the root ambient, the start of the program execution. The references establish the causality between the parent ambient and its nested ambients, which means the parent ambient is always created before the ambients it contains.

Defining Programs

After the existence of an ambient is declared with a `create` event, it is directly followed by a `deploy` event. The `deploy` event defines what the expected next steps of the program are. During the verification, the information in a `deploy` event is used to check against the events recorded in a log. To know if what happened was correct, we need to know what was supposed to happen. The `deploy` provides the verifier with this information, making it a core data structure that allows *verifiability* of the program behavior. It also allows the verifier to check where and when new capabilities were adopted.

If an ambient contains parallel nested ambients, a concurrent `deploy` event is created for each nested ambient. For example, the ambient `a[b[open_] | c[open_]]`, would create two concurrent `deploy` events, one for `b[open_]` and one for `c[open_]`, both referring to the `create a` event of their parent ambient `a`. Subsequently, `b[open_]` and `c[open_]` would respectively first record `create b` and `create c` followed by `deploy open_` and `deploy open_` events.

Separating the creation and deployment of the sub-parts of the program has two important consequences.

First, it allows a program to be *sliced* to its sub-parts which enables parallel, isolated execution of the parts of the program. *Slices* of a program can be distributed to the network to be executed by other network participants.

Second, *slicing* a program prevents information leakage, that is, to not reveal information about a program which shouldn't be revealed to network participants. A participant receiving an event needs to be able to traverse all the way to the root ambient to verify its source, name and expected steps, but it should not learn about the other, parallel sub-parts of the parent program. If a program was defined using only the `create` event, the receiver would learn everything about the program: all parallel ambients of the program, what other calls were made, and to whom, and more. Separating the `deploy` event from the `create` event, provides the necessary structure to keep parts of the program hidden from the other parts. This enables network participants to execute and verify a sub-part of the program, but they can't reconstruct the full program.

For example, a program that makes 100 function calls that are not dependent on each other can be distributed to 100 different network participants, each participant being oblivious of the other 99 function calls executed by the other participants.

Defining a `deploy` event and separating it from the `create` event constitutes the core structure and mechanism for verifiably distributing and executing programs in a network.

The program defined by the `deploy` event is referred to by its address, which is included in the `deploy` event.

Duality of Capabilities and Co-capabilities

The Robust Ambients calculus' reduction rules specify that a reduction finishes only when a capability and its respective co-capability are both consumed at the same time. We call this connection the *duality of capabilities and co-capabilities.*

The duality is reflected in the log events by the definition that 1) there's a direct reference between the capability and co-capability events and 2) there are no other recorded events between them.

The events for co-capabilities `in` and `out` reference the matching event of the `in` and `out` capabilities. For example, executing a program `a[call[out a] | out call]` creates the event `out a` in the log of `a` followed by the matching event `out call` which refers to the previous event: `out a ← out call`. This forms the verifiable link between the request to move in or out of an ambient, the authority that the moving ambient was allowed to do so and a record that the ambient has indeed moved.

The `open` and `open_` are ordered differently than `in` and `out`. The `open` capability references the matching `open_` co-capability, e.g. `open_ ← open a`. Opening an ambient gives new capabilities to its opener. For example, when the ambient `a[in b.open_]` enters ambient `b[in_ a.open a]`, ambient `b` adopts the `open_` capability from `a`. The capabilities adopted by the opening ambient can be deduced from the `deploy` events referred to by the `open` event and its previous events. Interestingly, this leads to an observation that the only event that can refer to an `open_` event is the respective `open` event.

Transferring Control

During the execution of a program, ambients can move out of their parent ambients and into other ambient. Moving into another ambient is marked by the `in` capability, and recording it in an event defines that the control of the ambient has been transferred from its parent to the ambient it enters.

For example, reducing the ambient `a[in b] | b[in_ a]` generates an event with an op `in b` in the log of `a`. Given the duality of capabilities and co-capabilities, a concurrent event `in_ a` is recorded in the log of `b`.

As the control of the ambient is passed to another ambient, the `in` capability is special compared to the other capabilities: when an `in` event is recorded, the control of the ambient has been passed to the destination ambient, after which the parent ambient has no control over that ambient. The destination ambient, then, adopts the remaining capabilities of the entering ambient. For example, when the ambient `a[in b.open_]` enters ambient `b[in_ a.open a]`, ambient `b` adopts the `open_` capability from `a`. In turn, the destination ambient will record the events produced by the adopted capabilities to its own log (as opposed to writing them to the parent ambient's log, or the parent ambient continuing to write the events to its own log). This minimizes the need for synchronization between the parent and destination ambients, which means that they can continue the program execution independently and in isolation of each other. The definition of transferring the control is thus a construction that allows programs to be distributed efficiently to network participants.

The control transfer sets some requirements for recording the `in` event. Because the control is transferred and the destination ambient adopts the remaining capabilities from the parent ambient, the `in` event must make sure that all capabilities and ambients of the parent are transferred. To achieve this, every `in` event references the previous events, like other events do, but in addition it references all the "*heads*" (that is, a union of events that no other event in the log points to) of the ambients and capabilities that are nested inside of the moving ambient.

Evaluating Results

When the control is transferred back to the caller, the caller's virtual machine has all the information it needs to evaluate the result of the computation.

The evaluation of the result happens by inspecting the log of events the remote participant has generated, starting from the received `in` event. Because the last recorded `in` event contains a reference to all the *heads* in the execution DAG, the caller can traverse the log all the way back to the `in` event that started the remote execution and thus determine the runtime state of the program evaluation.

If all the heads in the `in` event refer to a program that has no unconsumed capabilities or co-capabilities left to consume, the VM determines that the program evaluation has entered a constant and immutable state, a final value. This value is then evaluated by the VM to its primitive data type. Optionally, VM can use the remotely precomputed results, accessible via content-address in event `payload`. Consider the string concatenation monoid:

```
string[
  concat[
    left[string[a[]]]|
    right[string[b[]]]
  ]
]
```

The value represented by the ambients, as constructed from the log, is evaluated to `"ab"` by the string implementation of the Virtual Machine. This value is then returned by the State Machine upon querying the current state of the program.

Upon receiving the `in` event, having verified and evaluated the resulting value, the caller records the evaluation action by writing an `open_` event to the log of the returned program. The caller then writes an `open` event, to the log of the calling program, which references the previous `open_` and the content-address of the final state as a `payload`, thereby concluding the evaluation of the result.

Identities and Addresses

Deploying a program creates a root manifest file. This file contains the program "bytecode" and the public signing key of the deployer. The manifest is then signed by the deployer to prevent forging of program deployments. The manifest file is hashed and the hash of the manifest is the identifier of the program.

The identifier in turn is used to construct a *program address*. The program address consists of the protocol prefix and the identifier, separated by `/`. For example, if the manifest hashes to `zdpuAwAdomEUPx54FZVLt33ZeGZ5VrJkTgLxQiUZNBwZ3kr7e`, the address of the program can be represented as (*complete hash truncated for brevity*):

```
/amb/zdpuAwAdomEUPx54FZVLt33ZeGZ5VrJkTgLxQiUZNBwZ3...
```

The address establishes location transparency [49]: a single, logical address, to which messages can be sent to and which is always available, regardless of *where* the program is run. If hashes are identifiers to *immutable* content in a content-addressed system, then an identifier for a program in the Ambients protocol is an identifier to *mutable* content.

This effectively means that a program in an address is like a database or service that can be queried.

Sending messages and listening to this address, the network participants exchange messages about the latest events. Upon receiving events, participants apply them to their local copy of the log. To verify that an event is valid for a log, the receiver 1) fetches the manifest from the content-addressed network 2) reads the "keys" field from the manifest 3) checks that the creator's public signing key defined in the event is present in the "keys" field. If the key is found from the manifest, if the log id in the event matches the manifest hash and if the signature of the event verifies, the receiver can establish that the creator of the event is allowed to write the event to the log.

The manifest contains:

- `program`, which is the hash of the program bytecode and by which the program bytecode can be retrieved from a content-addressed storage
- `name`, to describe the program
- `keys`, which is an address to a list of keys that are allowed to write to the log of the program
- `creator`, which identifies the creator of the program and their public signing key
- `signature`, to establish authenticity of manifest

For example, a manifest that hashes to `zdpuAyJe8DpoEAbs2z3djcNs2XnQBPExisJuqfpo4mygDmLXK`, would be presented in JSON as (*complete hashes truncated for brevity*):

```
{
  program: 'zdpuAkfNT6xd5mC3Jk3ZNMGrjoqqRqSKTLjU...',
  name: 'hello-world',
  keys: '/amb/zdpuAuTSoDhKKgAfjJBRvWw4wSg5r6b3oW...',
  creator: {
    id: 'zdpuAwkLw7KAgXSEqduQQoyo9MrpkWrKDrKtBUg...',
    publicKey: '04c9680e7399c5d9589df2b62f32d568...'
  }
  signature: '30440220264d3bab838066d856087779af...',
}
```

The manifest defines the keys of the participants who are allowed to execute the program, which means they're able to write to the log of the program. The "keys" field contains an address which, when resolved, returns a list of keys. The address can be either an immutable file or an address of another ambient program that works as mutable list of keys, for example an access controller program.

Each sub-part of the program creates their own manifest and attaches the address of the manifest to the `deploy` event created by that sub-part.

Separating the manifests per sub-program keeps the full program information hidden. This means that knowing an address of a sub-part of the program doesn't reveal the address of the full program. Only the deployer of the program has the address of the full program. The deployer can give the address to others if they wish to share the program with them. By not giving the address to others, the program and its state or result stays hidden. Knowing the address is considered having "read access". However, to keep the program bytecode, its access control information and meta data confidential, the fields in the manifest file can also be encrypted.

Defining the keys in the manifest allows the deployer to define a) who can call the deployed program, e.g. only the creator, a set of nodes or anyone, and b) when requesting a computation from the network, e.g. calling a function, who can execute that function for the deployer. This allows granular access control and lets deployer define a specific, for example a "trusted", set of participants for a program or parts of it.

In addition, the manifest can be used to describe other useful information, such as encryption keys to enforce confidential computation between the participants. The granularity makes it possible to define authorization or encryption on a per-function-call level, which means that for example granting or revoking access can be done at any point in time.

Runtime Environment

Programs are executed by a runtime environment. We define the runtime environment for the protocol as a virtual machine.

The Virtual Machine

The runtime, defined as a *virtual machine* (VM), is software that takes the program's compiled bytecode as an input, verifies that the bytecode is valid, executes the instructions defined by the program, writes the events to the log, communicates with other participants in the network and verifies events received from them, interfaces with the operating system, handles and manages keys for signing and encryption, and more.

The purpose of the VM is to provide a platform-independent runtime environment to run programs in a sandboxed and isolated environment, limiting the access to the underlying operating system and hardware. It abstracts away the details of the underlying systems and hardware allowing a program to be executed the same way on any platform. For example, a VM for the protocol implemented in JavaScript can be run on Node.js or in the browsers, and a VM implemented in Rust can be run as a native program on a chosen operating system, both being able to run the same programs and communicate with each other.

Network participants running programs on the virtual machines form a network. The VM is responsible for distributing the computational workload to the network participants: it can decide to run a computation locally, or only parts of it locally and to request other parts to be computed by the network. Multiple programs can be run at the same time and a single program can perform multiple computations in parallel. It is the responsibility of the VM to coordinate and schedule the computation workloads to the appropriate resources. The VM is also responsible for communicating with the network and its participants, handling identities, signing and authenticating messages, and ultimately verifying that the protocol is followed. All VMs in the network have the responsibility to verify the remote programs upon execution and to refuse executing any invalid or incorrectly behaving programs. Incorrect behavior also includes compilers generating invalid executables and VMs failing to carry out their responsibilities, whether due to implementation errors (i.e. bugs) or malicious intent.

The VM keeps track of the program execution and encapsulates its state using state machines.

The VM works as an interface between the programs and the operating system. It provides APIs for the programs to access operating system level services. This includes for example, an access to storage to persist data, networking functionality to communicate with the network, and cryptographic primitives for hashing or encryption. The VM also implements primitive data types, such as *integers* or *strings*, and provides a core library for the programs to use. The required data types and interfaces are defined in Primitive Data Types and System Interface.

The VMs are free to do optimizations internally to make the execution of programs more efficient. For example, optimizations could include pre-fetching programs or logs from the network, optimizing network topologies or using program type information for more efficient evaluation of computation results.

Discovery

In order to distribute programs and have them run by other network participants, the execution model defines that a *discovery* mechanism is used to become aware of programs and participants willing to execute them.

The discovery mechanism itself is not strictly defined by the execution model and implementations are free to use various mechanisms or protocols to perform the discovery. In general and at minimum, the discovery mechanism should communicate the address of the program or the hash of the program bytecode, in order to establish that the correct program is verifiably executed by the remote participants.

For example, a Publish-Subscribe mechanism, a marketplace that connects those wishing to distribute a program and those willing to execute it, a private network or system with existing discovery service, or even out-of-band mechanisms can all be used to connect the participants and exchange the program information.

State Machine

The virtual machine uses a finite-state machine for each program. The state machines track the state of the program: where the execution of the program is at any given time, the possible next steps and the result of the program. In other words, the state machine represents the computed value of a program.

The state machine takes a log of events as its input and outputs the current state. To do this, the state machine replays the events by starting from the first event and applying each event to the current state, updating its state on each event. Upon receiving a new event from the network, the event is passed after verification to the state machine's update function, which triggers the calculation of the new state. If all events required to change the state have been received, the state machine proceeds to the next state through a a state transition.

The state machine is internal to the VM and is not exposed to the user.

Primitive Data Types

The virtual machine implements a set of common, primitive data types such as strings, numerals, arrays, and more. The implementation of the primitive data types, and the functions to operate on them (e.g. *addition* or *multiplication* operations on integers) allows the VM to encode and decode between the event data and typed runtime data to efficiently evaluate and handle such types.

Depending on the source language a program is written in, the primitive data types can be built-in to the source programming language or they can be exposed to the user through a library.

The detailed list of primitive data types to be implemented by the virtual machines will be defined in the future. For now, we envision at least the following types to be included, and upon which to expand in the future:

- Booleans
- Integers
- Floating-point numbers
- Bytes
- Characters
- Strings
- Tuples
- Lists

System Interface

The system interface provides a unified way for programs to use operating system services across all platforms. The programs will want to persist data on disk, to be able to communicate with other participants in the network, to use cryptographic keys and functions, and more. The system interface exposes this functionality to the programs and the virtual machine manages the calls to the actual services.

It is important to separate the system level calls from the application and protocol level, as it draws a clear line between what the protocol can guarantee and what it cannot: *all requests and responses to the operating system are* ***not*** *verifiable by the protocol*. That is, the user must trust their execution environment, i.e. the virtual machine and the operating system, to function correctly.

The system level services, accessed through the system interface, are I/O operations. From the system perspective, they cause side effects and as the system calls can go all the way down to the hardware level, the protocol can't verify that the I/O actually happened. However, all system interface services, except the Untrusted Code Execution Interface, are required to be deterministic in a way that they return either failure or always the same result for same input.

As a rule, programs using the system interface are executable by any node in the network, so all virtual machines must implement the system interface. The Untrusted Code Execution Interface is the exception to this rule, and programs using it are not expected to be run by all nodes, only by a subset of the network.

Having a unified system interface for all programs allows the virtual machines, i.e. the protocol implementations, to use different components as part of their implementation. This allows different storage backends to be used and the user can choose the storage according to their needs. For example, in a trustless environment an application could choose to use a Merkle-tree-based content-addressed storage system, such as IPFS [29], whereas in a trusted system, the users could opt to use a traditional database system.

The exact APIs for the system interface will be specified in the detailed protocol specification. We envision the interfaces to provide access to at least the following services:

Content-addressed Storage provides a file system and a data storage layer and can be used to store arbitrary data in various formats and persist files in directory hierarchies. The programs can fetch data, download files or list contents of a directory through the storage interface.

Peer-to-Peer Networking provides functionality to manage connections to other participants on the system and network level. The programs can request the system to open a connection to a certain participant, to discover new participants through DHT and other discovery mechanisms, or to join a group communication channel through a Pubsub system.

A **Cryptography Interface** provides secure cryptography functionality to the programs. Signing and encryption keys can be generated, signatures can be verified and data decrypted through the interface. Keys can be stored and managed through the interface. For example, a wallet for a crypto-currency, such as an Ethereum, could be used as the underlying implementation.

The **Untrusted Code Execution Interface** provides a way to access untrusted or highly hardware-dependent services of the system. For example, random number generators and time APIs can be accessed through the interface. Programs that require specific binaries to be available in the operating system, or require communication with external systems, such as to make calls to smart contracts or to location-addressed systems, can use the untrusted code execution interface to access those services. However, since the interface allows access to arbitrary functionality, not all participants in a network will support all the same functionality. The support for specific services through the Untrusted Code Execution Interfaces depends on each individual participant.

The relationship and the order of the layers from a program down to the operating system level can be described with the following diagram:

```
+-----------------------------------------------+
|                   Program                     |
+-----------------------------------------------+
|                     VM                        |
+-----------------------------------------------+
|               System Interface                |
+-------+-------+------+------------------------+
|Storage|Network|Crypto|Untrusted Code Execution|
+-------+-------+------+------------------------+
```

VI. Applications

Fundamentally, the Ambients protocol makes it possible to share application or system state as distributable data structures. By modeling data and systems as functions, developers can easily move between different levels of abstractions. The programming and execution model that the Ambients protocol institutes has several important consequences.

First, there are no more servers to interact with to query and modify the application's data. The state of the database doesn't need to be replicated in a separate application-level data model but instead, the database is directly programmable from the application. This reduces the coordination overhead and the complexity of all translation layers of classes and objects trying to keep the constantly updating database and application data model in sync. Being directly programmable also means that developers are free to use familiar language and tools, which lowers the threshold for grasping the programming model, and ultimately leads to quicker iteration times.

Second, developers can create local-first applications that work offline, creating a superior user experience compared to the client-server model. For example, the user can post to their blog while offline due to the "blog program" and "posts database" being local programs on the user's device. When they publish a post, the program doesn't need to make a request to a server and instead the post is written directly to the local database. The act of publishing feels magically instantaneous. This is especially important for the decentralized web to win the hearts of mobile users.

Third, all data can be seen as data structures on different levels of abstractions. At the smallest level, the developer can model and share the application state with primitive data structures, such as Sets, Lists or even an individual number (counter). Updating that data model propagates changes to all participants using the model, safely under the Ambients execution model. New data models and structures can be composed from existing models and structures to support more complex applications. By composing new data structures and sharing functions to update, filter or transform the data, we start seeing something resembling a database. When we combine multiple databases, data structures, such as user profiles or activity feeds and posts, and add business logic, we start seeing an application. Composing several data structures, databases and applications together, we have a system, a decentralized service.

Fourth, because of the lock-step fashion of function calls in the Ambients protocol, we can look at the function calls as exchanging messages in request-response style. This allows us to create transactions and protocols. Participant A sends a message X to the other participants in the program and they, upon receiving the message, send an acknowledgment of X back to participant A. This constitutes a simple two-step protocol, but the same pattern can be extended to any complex set of rules and behaviors. For example, a blockchain or crypto-network protocol can be constructed by following this perspective.

Finally, the ultimate benefit for the applications comes from separating the data and programs from the underlying platform. Making the program available in a network makes the computed data reusable in multiple applications, even at the same time. For example, a "GitHub" and a "Twitter" application can both update a user's profile or activity feed, or the user can share and collaborate on a playlist created in their music player with a friend using another music player. Data and programs being reusable, like code modules, means that developers and their users are free to switch platforms, service providers or data sources at any time, keeping their data as they go. Being programming language- and platform-agnostic, the Ambients programming model lets developers continue to use familiar tools and languages to build their applications, and to decide on the level of consistency, security and decentralization they want to offer to their users. With all this power, developers can stop fearing their platform choices and start focusing again on quality, user experience, security, and everything that is actually valuable to their users.

Data Structures

> *"Data structures are algorithms with memory." Elad Verbin*

The Ambients protocol allows any data structure that can be modeled as a function to be turned into a distributed program. This makes it a very powerful tool for building components and shared data structures for distributed systems and services.

Conflict-free Replicated Datatypes, CRDTs [14], have turned out to be hugely helpful data structures in implementing distributed system. In essence, they represent a shared state of a particular data structure that can automatically converge to a deterministic result across the participants. That is, all participants, having received all updates to the data structure, observe the same value.

Operation-based CRDTs can be modeled as updates to the data structure, causally ordered in time. The execution model of the Ambients protocol defines the same operation-based, causality-defining structure and lends itself perfectly to implementing CRDTs. For example, adding an element to a Growth-Only Set CRDT by calling a function `gset_append("hello", prevSet)` would generate the events in the execution log that, upon reading the value, would result in a set of `["hello"]`.

Databases

Databases are specialized data structures. Most databases have a concept of an operations log, also called the "transaction log", which records all updates to the database and from which the state of the database is computed from. The log in the execution model of Ambients is an operations log and thus directly translates to databases. Add indexing and the ability to query, and we have a database. Access to the database and data can be controlled on granular levels. The Ambients protocol and its core execution model is an operations log and as such, any kind of database or data model can be built on it.

For example, database tables are data structures that can be constructed from database "operations" and upon querying the database, multiple tables are joined to return the result. In event sourcing, log databases are commonly used as the event store and the Ambients protocol provides a natural way to do distributed event sourcing and state updates. In the decentralized world, OrbitDB [46] at its core has a distributed, peer-to-peer operations log (ipfs-log) [30] and its execution model is very similar to that of the Ambients protocol. Using the log abstraction, OrbitDB is able to offer multiple database models: key-value databases, document stores, counters, event logs, CRDTs and more. 3Box [1], built on OrbitDB, provides a very specific and specialized type of database: user profiles.

Protocols

Consider a protocol where a specific sequence of specific messages are exchanged in order to reach a final agreement. At each step, preconditions are verified before progressing to the next step. Protocols can be anything from a simple two-party transaction (or payment) protocol to a complex set of rules and states such as distributed consensus protocols. For example, payment channels and state channels, business logic ("if this then that"), access controllers ("policy") or consensus protocols, can be seen as an exchange of an ordered sequence of messages and the Ambients protocol is well-suited for modeling such logic.

Decentralized Applications

If we break down the structure of many proprietary web applications and services today, we see that structurally they contain many smaller components and data structures. For example, GitHub contains organizations, users, repositories, issues, comments, etc.; Twitter contains users, tweets, a feed, private messages, etc.; a chat application has users, bots and channels; a blog has posts, comments, pages, etc.; an e-commerce site has shops, items, reviews, payments, delivery tracking, etc.

By breaking down the application data and state to small databases and data structures, we can compose applications from existing parts. Since the parts are not tied to the particular applications, e.g. a user's database of tweets, other applications can build new UIs to view and modify it, or use the database as part of an aggregation service that composes data from multiple users together. Multiple programs can use the same database, at the same time, to offer different views to the data.

Programs using the Ambients protocol become offline-capable out of the box because of the locality of data. This allows snappy UIs, and applications can still work when there's no internet connection (e.g. in LAN) or the connection is sporadic (e.g. mobile devices or IoT).

Generally, applications that enable collaboration on the same document or data are well suited to be built on Ambients. Collaborative document editing, chat software, discussion forums, comment feeds and task lists are all exemplary programs that benefit from decentralization. Turn-based and real-time multiplayer games can verifiably track the state of a game session, player's actions and inventory, and on a higher-level, the state of the whole game world.

While Ambients protocol enables data models where the users own their data, this is not always desired or needed. For example, who "owns" a public chat room? The decision of who owns the data is still something the developer needs to decide, but building programs on Ambients gives the developer the *possibility* to let users own their data, bring in their own identities, and generally respect user privacy and consent (or lack thereof).

Digital Services

Databases, identities, APIs and other services are the cornerstones of many digital services. Composing small and large pieces of data, aggregating data and running an algorithm to provide new insight to that data or providing authentication or payments services, are all valuable services in the decentralized web.

Due to the compositionality of ambients, services can be composed from various parts, both decentralized and centralized, public and private, to create larger systems and networks that serve their clients' needs. For example, a web API is a service to access data, Facebook as a whole is a service that combines many smaller parts together (contacts, posts, news feed, sharing, messages, etc.), a bank service that validates that a payment was made, a private or proprietary registry that tracks and validates membership of users, or a marketplace that connects, asks, and bids.

VII. Future Work

The Ambients protocol combines efficient and practical distributed systems with safe and formal models of distributed computation, forming a novel approach for building truly decentralized systems.

While the protocol outlined in this paper forms the core of the Ambients protocol, there are various additions and improvements that will be included in the protocol in the future. This section outlines planned future additions and improvements to the protocol, ongoing research and practical development work.

As an open source protocol, future research and development will be performed and coordinated by the Ambients protocol community on GitHub at https://github.com/ambientsprotocol [9].

Research

Current and future research is focused on improving the protocol capabilities to make the execution model more robust and the programming model more expressive, making emergent system designs possible.

To make the programming model more expressive, there is ongoing research to include additional computation abstractions. These include conditionals such as *if-else* statements, the *ternary* operator and pattern matching. Together, these abstractions give the end user better expressiveness to handle control flow in their programs.

In order to have more powerful program and data composition primitives, we plan to add computation abstractions for applicative functors [10], monads [42] and profunctors [5], and more. In addition, it would be interesting to use *codata types with copattern matching* [15] to represent infinitely running programs.

An ambient calculus variant for the Ambients protocol is being researched, to improve the overall efficiency of the protocol. This variant defines the Ambients protocol primitives as ambient calculus primitives, making the protocol encodings simpler and more efficient for the execution environment.

In the future, more research is needed to enable safe optimizations in compilers (inlining, constant folding, etc.) and to use various caching strategies and algorithms in the execution model to improve local versus remote latency. For example, it would be interesting to create a model for optimal resource utilization to maximize the overall network throughput.

Development

We envision Ambients protocol implementations being used in various environments. This means that development efforts need to focus on making sure there is wide support for underlying operating systems and platforms. Early implementation efforts are focused on covering most common operating systems and hardware architectures as well as supporting web browsers (JavaScript or WASM). In the future, we envision Ambients programs being able to be run on resource-limited devices, such as mobile and IoT devices.

Future development efforts will focus on improving the architecture and features of the virtual machines. For example, we envision VMs being able to do efficient garbage collection, just-in-time compilation of Ambients programs and even to model verifiable compilation.

Compiler development efforts will focus initially on providing support for some commonly used programming languages, such as JavaScript. In the future, we envision adding compiler support for a host of languages that can be used to create Ambients programs. Also, we look forward to improve the compilers by letting them connect to the VMs and participate in the verification process - to make the developer experience friendlier and safer, and to shorten the feedback loop for verifying the correctness of whole program.

As the Ambients protocol gets implemented for various environments, better tools are needed to make life easier for the implementors and researchers. AmbiCobjs [8] has been a crucial tool in designing the protocol so far and we envision a modern version to be implemented that allows simulating, debugging and auditing Ambients programs and the networks they form in real-time. For example, given the nature of the step-wise recording of the execution of Ambients programs, the protocol is well-suited for time-travel debugging and simulation of the programs. Given the mobility of ambients themselves, visualizing the networks and nodes that participate and perform computation would be highly beneficial to better understand and debug the networks and programs.

VIII. Conclusion

We have introduced the Ambients protocol for creating and verifiably executing distributed programs in decentralized, peer-to-peer networks. The main contributions of the protocol are defined in three models: the Programming model, Compilation model and Execution model.

The Ambients Programming model defines the translation of a source program to a distributed program. The Programming model guarantees that if a source program can be defined as pure and total functions and immutable values, it can be transformed to a distributed Ambients program, defined as a composition of protocol primitives. The Ambients Compilation model defines the structure of distributed program executables. The Ambients Execution model defines the control flow of Ambients program deployment and execution in a content-addressed peer-to-peer network, recorded as a Merkle-DAG based event log. The Execution model guarantees the verifiability, authenticity and integrity of the execution of all distributed Ambients programs. The guarantees of all models are enforced and enabled by modeling the deterministic evaluation of programs as a strongly confluent rewrite system based on a process algebra called ambient calculus.

Building on the computational properties and the hybrid distributed evaluation strategies of Ambients, we identify an emerging breed of decentralized applications, such as data structures and databases with shared distributed state, protocols for distributed consensus and program logic, and even digital services with compositional APIs, bridging the centralized and decentralized world.

We look forward to implementing the Ambients protocol as an open-source community and together pursue the vision of a fully, truly, decentralized web.

IX. References

1. 3Box, https://3box.io/
2. Johnsen, E.B., Steffen, M., Stumpf, J.B.: A Calculus of Virtually Timed Ambients. In: James, P. and Roggenbach, M. (eds.) Recent Trends in Algebraic Development Techniques. pp. 88–103. Springer International Publishing, Cham (2017)
3. Merkle, R.C.: A Certified Digital Signature. In: Brassard, G. (ed.) Advances in Cryptology — CRYPTO' 89 Proceedings. pp. 218–238. Springer New York, New York, NY (1990)
4. Farmer, C.: A deeper look at the tech behind Textile's Threads, https://medium.com/textileio/wip-textile-threads-whitepaper-just-kidding-6ce3a6624338
5. A Neighborhood of Infinity: Profunctors in Haskell, http://blog.sigfpe.com/2011/07/profunctors-in-haskell.html
6. Borril, P., Burgess, M., Craw, T., Dvorkin, M.: A Promise Theory Perspective on Data Networks. arXiv:1405.2627 [cs]. (2014)
7. Birch, M.: A Visualization for the Future of Blockchain Consensus, https://medium.com/rchain-cooperative/a-visualization-for-the-future-of-blockchain-consensus-b6710b2f50d6
8. Zimmer, P.: Ambient Programming in Icobjs, http://www-sop.inria.fr/mimosa/ambicobjs/
9. Ambients, https://github.com/ambientsprotocol
10. Applicative functor - HaskellWiki, https://wiki.haskell.org/Applicative_functor
11. Alexander, A.: Benefits of Functional Programming, https://alvinalexander.com/scala/fp-book/benefits-of-functional-programming
12. Bitcoin, https://bitcoin.org/en/
13. Bugliesi, M., Castagna, G., Crafa, S.: Boxed Ambients. In: Theoretical Aspects of Computer Software. pp. 38–63. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
14. Preguiça, N., Baquero, C., Shapiro, M.: Conflict-free Replicated Data Types (CRDTs). arXiv:1805.06358 [cs]. (2018). doi:10.1007/978-3-319-63962-8_185-1
15. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13. p. 27. ACM Press, Rome, Italy (2013)
16. Correct-by-construction Casper Wiki, https://github.com/ethereum/cbc-casper/wiki
17. Dat Foundation, https://dat.foundation/
18. Zeltser, L.: Early History of the World-Wide Web: Origins and Beyond, https://zeltser.com/web-history/#Origins_WWW
19. Gonzalez, G.: Equational reasoning, http://www.haskellforall.com/2013/12/equational-reasoning.html
20. Ethereum, https://ethereum.org
21. EUGDPR – Information Portal, https://eugdpr.org/
22. Fowler, M.: Event Sourcing, https://martinfowler.com/eaaDev/EventSourcing.html
23. Vogel, W.: Eventually Consistent - Revisited, https://www.allthingsdistributed.com/2008/12/eventually_consistent.html
24. Vukolic, M.: Eventually Returning to Strong Consistency. IEEE Data Eng. Bull. 39, 39–44 (2016)
25. Functor - HaskellWiki, https://wiki.haskell.org/Functor#Functor_Laws
26. Git, https://git-scm.com/
27. Buus, M.: Hypercore is a secure, distributed append-only log., https://github.com/mafintosh/hypercore
28. Benet, J.: IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561 [cs]. (2014)
29. Protocol Labs: IPFS is the Distributed Web, https://ipfs.io/
30. ipfs-log - Append-only log CRDT on IPFS, https://github.com/orbitdb/ipfs-log
31. Claburn, T.: "Lambda and serverless is one of the worst forms of proprietary lock-in we've ever seen in the history of humanity," https://www.theregister.co.uk/2017/11/06/coreos_kubernetes_v_world/
32. Protocol Labs: libp2p - A modular network stack, https://libp2p.io/
33. Helland, P.: Life Beyond Distributed Transactions - ACM Queue, https://queue.acm.org/detail.cfm?id=3025012
34. Bernardy, J.-P., Spiwack, A.: Linear types make performance more predictable, https://www.tweag.io/posts/2017-03-13-linear-types.html
35. Bailis, P.: Linearizability versus Serializability, http://www.bailis.org/blog/linearizability-versus-serializability/
36. Kleppmann, M., Wiggins, A., van Hardenberg, P., McGranaghan, M.: Local-first software: You own your data, in spite of the cloud, https://www.inkandswitch.com/local-first.html
37. Kahle, B.: Locking the Web Open: A Call for a Decentralized Web, http://brewster.kahle.org/2015/08/11/locking-the-web-open-a-call-for-a-distributed-web-2/
38. Guan, X., Yang, Y., You, J.: Making Ambients More Robust. (2002)
39. Sanjuan, H., Pöyhtäri, S., Teixeira, P.: Merkle-CRDTs, https://hector.link/presentations/merkle-crdts/merkle-crdts.pdf, (2019)
40. Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure. pp. 140–155. Springer-Verlag, London, UK, UK (1998)
41. Genovese, F.R.: Modularity vs Compositionality: A History of Misunderstandings, https://blog.statebox.org/modularity-vs-compositionality-a-history-of-misunderstandings-be0150033568
42. Monad - HaskellWiki, https://wiki.haskell.org/Monad

43. Phillips, I., Vigliotti, M.G.: On Reduction Semantics for the Push and Pull Ambient Calculus. In: Baeza-Yates, R., Montanari, U., and Santoro, N. (eds.) Foundations of Information Technology in the Era of Network and Mobile Computing. pp. 550–562. Springer US, Boston, MA (2002)
44. Peters, K., Nestmann, U.: On the Distributability of Mobile Ambients. arXiv:1808.01599 [cs]. (2018)
45. Kleppmann, M., Gomes, V.B.F., Mulligan, D.P., Beresford, A.R.: OpSets: Sequential Specifications for Replicated Datatypes (Extended Version). arXiv:1805.04263 [cs]. (2018)
46. OrbitDB, https://github.com/orbitdb
47. Verborgh, R.: Paradigm shifts for the decentralized Web, https://ruben.verborgh.org/blog/2017/12/20/paradigm-shifts-for-the-decentralized-web/
48. Human Rights Watch: "Race to the Bottom": Corporate Complicity in Chinese Internet Censorship, https://www.hrw.org/reports/2006/china0806/5.htm
49. Bonér, J.: Reactive Microsystems: The Evolution of Microservices at Scale. O'Reilly Media, Inc (2017)
50. Scuttlebutt - a decent(ralised) secure gossip platform, https://www.scuttlebutt.nz/
51. Scuttlebutt Protocol Guide, https://ssbc.github.io/scuttlebutt-protocol-guide/#feeds
52. Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs]. (2018)
53. Catalini, C., Gans, J.S.: Some Simple Economics of the Blockchain. Social Science Research Network, Rochester, NY (2017)
54. Textile, https://textile.io/
55. Barrera, C.: The Blockchain Effect, https://medium.com/mit-cryptoeconomics-lab/the-blockchain-effect-86bd01006ec2
56. Tabora, V.: The Evolution of the Internet, From Decentralized to Centralized, https://hackernoon.com/the-evolution-of-the-internet-from-decentralized-to-centralized-3e2fa65898f5
57. Seemann, M.: The Identity functor, https://blog.ploeh.dk/2018/09/03/the-identity-functor/
58. Hickey, R.: The Value of Values, https://gotocon.com/dl/goto-cph-2012/slides/value-of-values.pdf
59. Berners-Lee, T.: Three challenges for the web, according to its inventor, https://webfoundation.org/2017/03/web-turns-28-letter/
60. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM. 21, 558–565 (1978). doi:10.1145/359545.359563
61. Kwang, Y.S.: Total functional programming, https://kseo.github.io/posts/2015-06-18-total-functional-programming.html
62. Turner, D.A.: Total Functional Programming. J. UCS. 10, 751–768 (2004). doi:10.3217/jucs-010-07-0751
63. Hillston, J.: Tuning Systems: From Composition to Performance. The Computer Journal. 48, 385–400 (2005). doi:10.1093/comjnl/bxh097
64. Kreps, J.: What every software engineer should know about real-time data's unifying abstraction, https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
65. What is a Merkle DAG?, https://discuss.ipfs.io/t/what-is-a-merkle-dag/386
66. Dixon, C.: Why Decentralization Matters – Featured Stories, https://medium.com/s/story/why-decentralization-matters-5e3f79f7638e
67. Pressler, R.: Why Writing Correct Software Is Hard, http://blog.paralleluniverse.co/2016/07/23/correctness-and-complexity/
68. World's Biggest Data Breaches & Hacks, https://informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/
69. Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., Knottenbelt, W.J.: XCLAIM: Trustless, Interoperable Cryptocurrency-Backed Assets. (2018)