Almost every computing system nowadays is distributed, ranging from multicore CPUs prevalent in everyday desktops/laptops/Internet of Things (IoTs) to the very Internet itself. Thus, understanding the foundations of distributed computing is important for the design of efficient computational techniques across all scientific fields. Moreover, many problems that are trivial to solve sequentially are impossible or infeasible to solve in a distributed fashion, thus presenting us with problems of deep intellectual yet practical interest.

My research is primarily concerned with the theory and practice of distributed computing spanning several domain-specific computing models. I am interested in capturing real-world manifestations of concurrency phenomena in complex distributed (hardware or low-level/application-layer software) environments; deriving algorithms, complexity bounds and implementing verifiable prototype implementations. To that end, I have worked extensively on algorithms, formal semantics, lower bounds and programming models for multicore CPUs as well as cloud infrastructures, implementations of concurrent data structures, efficient and privacy-preserving protocols for distributed cryptocurrencies as well as applications of distributed algorithmic techniques in the context of networking. Typically, my work has involved bringing together a confluence of ideas from theoretical computer science and formal program modelling to better understanding systems architecture and networking design so as to bridge the gap from theory to practice in building principled distributed systems.

Distributed computing: A personal perspective. Designing provably correct distributed programs requires overcoming some nontrivial challenges, the most important of which is achieving efficient synchronization among processes. When there are several processes that attempt to concurrently access the same data, they will need to coordinate their actions to ensure correct program behaviour, thus motivating the search for efficient synchronization techniques. Synchronization though is hard due to failures and the asynchrony pervasive in distributed systems. In fact, one of the seminal results in distributed computing is the impossibility of deterministically achieving consensus: processes initially propose a value and must eventually agree on one of the proposed values, among failure-prone processes in an asynchronous environment. Alternatively phrased, this result implies that it is impossible to achieve Consistency (or safety), Availability (or progress) and Partition-tolerance in an asynchronous environment, the so-called CAP theorem. Intuitively, this result implies that, due to asynchrony, programmers of distributed applications may be unable to simultaneously provide strong variants of all three features-consistency, availability and the ability to cope with failures.

Consequently, I find myself asking the following unsurprising questions about a distributed computation:

(i) What is the model of computation? Answering this question requires identifying the communication model: via shared memory or message passing or variants thereof; timing assumption: synchrony vs. asynchrony which specifies the relative speeds with which processes take steps in the computation; the failure pattern which identifies the ways in which some subset of processes may become faulty; computing and memory capacity of the processes, etc.

(ii) Given a computational model, is it (im)possible to build a distributed application for a specific choice of consistency and availability? Can we capture the trade-off between consistency and scalability?

(iii) What are the complexity metrics that characterize the cost of the computation and can we derive bounds that identify the implementation’s theoretical cost?

(iv) What are the available hardware and programming abstractions that help application programmers realize a working implementation with largely sequential semantics in mind and finally

(v) How can we verify that the resulting implementation conforms to its intended sequential semantics?

Generally speaking, when reasoning about distributed computations in any context; multicore machines or graph algorithms for routing in today’s Internet network architectures or information sharing in social peer-to-peer networks, all the above questions apply. But there are also other practical considerations that may define the parameters of the
distributed model: Is there a bound on the number of participating computational entities? Are we seeking deterministic or randomized algorithms? Are we seeking to enforce any security, privacy or anonymity properties over the distributed computation? Lastly, how easy or hard is the distributed programming model? This latter question is especially important from a practical standpoint since it is the simplicity of the programming model that determines whether ordinary programmers may choose to adopt it. Reasoning about the correctness of a distributed computation is a science onto itself and the programming model must ideally enable programmers to build distributed applications with largely their sequential semantics in mind and without worrying about synchronization problems that may arise.

Document structure. In general, I often find that my choice of distributed computing models to focus my research is motivated and shaped by emerging new hardware trends that require a new abstract computation model to accurately describe concurrency phenomena or via introduction of techniques from distributed computing to domains where the sequential implementation continues to be state-of-the-art. Typically, it is the choice of the distributed model that dictates the (possibly) new mathematical and hardware/software machinery to learn or refine for implementing provably correct algorithms within the model.

The following sections give a flavor of the kind of distributed computation models I have studied and hopefully, a hint of how distributed computing techniques have become all-pervasive; directing emerging hardware trends to deploying applications on the cloud platform and how my own research has contributed to expanding the problem and solution space. In Part I, I focus on my work concerning synchronization algorithms for today’s multicore CPU architectures. This is typically modelled as a shared memory over which processes communicate using the CPU’s instruction set. In Part II, I focus on distributed models in which processes communicate by sending and receiving messages, as in cloud computing and smart contract ecosystems. In Part III, I focus on distributed network algorithms: distributed computing challenges at the heart of the modern network computing stack. I conclude this statement by outlining future research directions.

HAL: The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words, foolproof and incapable of error.

... 
HAL: I’ve just picked up a fault in the AE35 unit. It’s going to go 100% failure in 72 hours.
HAL: It can only be attributable to human error.

Stanley Kubrick-2001: A Space Odyssey

1 Shared memory model

Prior and ongoing work

1.1 Towards safe in-memory transactions

Traditional solutions for synchronization in shared memory like locking that provide mutual exclusion, i.e., restricting data access to at most one process at a time, come with limitations. Coarse-grained locking typically serializes access to a large amount of data and does not fully exploit hardware concurrency. Program-specific fine-grained locking, on the other hand, is a dark art to most programmers and trusted to the wisdom of a few computing experts. Thus, it is appealing to seek a middle ground between these two extremes: a synchronization mechanism that relieves the programmer of the overhead of reasoning about the conflicts that may arise from concurrent operations without severely limiting the program’s performance. The Transactional memory (TM) abstraction [54][84] is such a mechanism: it combines an easy-to-use programming interface with an efficient utilization of the concurrent-computing abilities provided by multicore architectures.

Transactional memory allows the user to declare sequences of instructions as speculative transactions that can either commit or abort. If a transaction commits, it appears to be executed sequentially, so that the committed transactions constitute a correct sequential execution. If a transaction aborts, none of its instructions can affect other
transactions. The TM implementation endeavors to execute these instructions in a manner that efficiently utilizes the concurrent computing facilities provided by multicore architectures.

1.1.1 Safety for Transactional Memory [13, 12, 14]

We formalize the semantics of a safe TM: every transaction, including aborted and incomplete ones, must observe a view that is consistent with some sequential execution. This is important, since if the intermediate view is not consistent with any sequential execution, the application may experience a fatal irrevocable error or enter an infinite loop. Additionally, the response of a transaction’s read should not depend on an ongoing transaction that has not started committing yet. This restriction, referred to as deferred-update semantics, appears desirable, since the ongoing transaction may still abort, thus rendering the read inconsistent. We define the notion of deferred-update semantics formally and apply it to several TM consistency criteria proposed in literature, including the popular consistency criterion of opacity [49] and some of its relaxations [57, 50]. We then verify if the resulting TM consistency criterion is a safety property [75, 8, 69] in the formal sense, i.e., the set of histories (interleavings of invocations and responses of transactional operations) is prefix-closed and limit-closed.

1.1.2 Complexity of transactional memory

One may observe that a TM implementation that aborts or never commits any transaction is trivially safe, but not very useful. Thus, the TM implementation must satisfy some nontrivial liveness property specifying the conditions under which the transactional operations must return some response and a progress property specifying the conditions under which the transaction is allowed to abort.

Two properties considered important for TM performance are read invisibility [16] and disjoint-access parallelism [58]. Read invisibility may boost the concurrency of a TM implementation by ensuring that no reading action can cause any other transaction to abort. The idea of disjoint-access parallelism is to allow transactions that do not access the same data item to proceed independently of each other without memory contention.

We investigate the inherent complexities in terms of time and memory resources associated with implementing safe TMs that provide strong liveness and progress properties, possibly combined with attractive requirements like read invisibility and disjoint-access parallelism. Which classes of TM implementations are (im)possible to solve?

Blocking TMs [62, 65]. We begin by studying TM implementations that are blocking, in the sense that, a transaction may be delayed or aborted due to concurrent transactions. (i) We prove that, even inherently sequential TMs, that allow a transaction to be aborted due to a concurrent transaction, incur significant complexity costs when combined with read invisibility and disjoint-access parallelism. (ii) We prove that, progressive TMs, that allow a transaction to be aborted only if it encounters a read-write or write-write conflict with a concurrent transaction [49], may need to exclusively control a linear number of data items at some point in the execution. (iii) We then turn our focus to strongly progressive TMs [49] that, in addition to progressiveness, ensures that not all concurrent transactions conflicting over a single data item abort. We prove that in any strongly progressive TM implementation that accesses the shared memory with read, write and conditional primitives, such as compare-and-swap, the total number of remote memory references [9, 15] (RMRs) that take place in an execution in which n concurrent processes perform transactions on a single data item might reach $\Omega(n \log n)$ in the worst-case. (iv) We show that, with respect to the amount of expensive synchronization patterns like compare-and-swap instructions and memory barriers [11, 71], progressive implementations are asymptotically optimal. We use this result to establish a linear (in the transaction’s data set size) separation between the worst-case transaction expensive synchronization complexity of progressive TMs and permissive TMs that allow a transaction to abort only if committing it would violate opacity.

Non-blocking TMs [63]. Next, we focus on TMs that avoid using locks and rely on non-blocking synchronization: a prematurely halted transaction cannot not prevent other transactions from committing. Possibly the weakest non-blocking progress condition is obstruction-freedom [52, 58] stipulating that every transaction running in the absence of step contention, i.e., not encountering steps of concurrent transactions, must commit. In fact, several early TM implementations [58, 70, 84, 85, 36] satisfied obstruction-freedom. However, circa. 2005, several papers presented the case for a shift from TMs that provide obstruction-free TM-progress to lock-based progressive TMs [28, 27, 33]. They argued that lock-based TMs tend to outperform obstruction-free ones by allowing for simpler algorithms with lower complexity overheads. We prove the following lower bounds for obstruction-free TMs. (i) Combining invisible reads with even weak forms of disjoint-access parallelism [17] in obstruction-free TMs is impossible, (ii) A read operation in a n-process obstruction-free TM implementation incurs $\Omega(n)$ memory stalls [32, 10], (iii) A read-only
transaction may need to perform a linear (in \( n \)) number of expensive synchronization patterns. We then present a progressive TM implementation that beats all of these lower bounds, thus suggesting that the course correction from non-blocking (obstruction-free) TMs to blocking (progressive) TMs was indeed justified.

**Partially non-blocking TMs** [64]. Lastly, we explore the costs of providing non-blocking progress to only a *subset* of transactions. Specifically, we require *read-only* transactions to commit *wait-free*, i.e., every transaction commits within a finite number of its steps, but *updating* transactions are guaranteed to commit only if they run in the absence of concurrency. We show that combining this kind of *partial* wait-freedom with read invisibility or disjoint-access parallelism comes with inherent costs. Specifically, we establish the following lower bounds for TMs that provide this kind of partial wait-freedom. (i) This kind of partial wait-freedom equipped with invisible reads results in maintaining unbounded sets of *versions* for every data item. (ii) It is impossible to implement a *strict* form of disjoint-access parallelism [47]. (iii) Combining with the weak form of disjoint-access parallelism means that a read-only transaction (with an arbitrarily large read set) must sometimes perform at least one expensive synchronization pattern per read operation in some executions.

**1.1.3 Complexity of Hybrid transactional memory**

The TM abstraction, in its original manifestation, augmented the processor’s *cache-coherence protocol* and extended the CPU’s instruction set with instructions to indicate which memory accesses must be transactional [54]. Most popular TM designs, subsequent to the original proposal in [54] have implemented all the functionality in software [26, 34, 53, 70, 59]. More recently, CPUs have included hardware extensions to support small transactions [77, 11, 74]. Hardware transactions may be spuriously aborted due to several reasons: cache capacity overflow, interrupts *etc*. This has led to proposals for *best-effort* HyTMs in which the fast, but potentially unreliable hardware transactions are complemented with slower, but more reliable software transactions. However, the fundamental limitations of building a HyTM with nontrivial concurrency between hardware and software transactions are not well understood. Typically, hardware transactions usually employ *code instrumentation* techniques to detect concurrency scenarios and abort in the case of contention. But are there inherent costs of implementing a HyTM, and what are the trade-offs between these costs and provided concurrency, i.e., the ability of the HyTM to execute hardware and software transactions in parallel?

**Cost of instrumentation in Hybrid transactional memory** [6, 5, 7]. (i) We propose a general model for HyTM implementations, which captures the notion of *cached accesses* as performed by hardware transactions, and precisely defines instrumentation costs in a quantifiable way. (ii) We derive lower and upper bounds in this model, which capture for the first time, an inherent trade-off on the degree of concurrency allowed between hardware and software transactions and the *instrumentation overhead* introduced on the hardware.

**Cost of concurrency in Hybrid transactional memory** [20, 19]. State-of-the-art *Software Transactional Memory (TM)* implementations achieve good performance by carefully avoiding the overhead of *incremental validation*, i.e., re-reading previously read data items to preclude inconsistent executions. Hardware TMs promise even better performance. However hardware transactions offer no progress guarantees since they may abort for *spurious* reasons and conflict aborts. Thus, they must be combined with software TMs, thus leading to the advent of *hybrid* TMs (HyTM). To allow hardware transactions in a HyTM to detect conflicts with software transactions, hardware transactions must be *instrumented* to perform additional metadata accesses. This instrumentation introduces overhead.

We first show that, unlike in software TMs, software transactions in HyTMs cannot avoid incremental validation. Specifically, we establish that *progressive* HyTMs in which a transaction may be aborted only due to a *data* conflict with a concurrent transaction, must necessarily incur a validation cost that is *linear* in the size of the transaction’s read set. This is in stark contrast to progressive software TMs which can achieve \( O(1) \) complexity operations. Secondly, we present two provably *opaque* HyTM algorithms in which both hardware and software transactions perform an optimal number of metadata accesses. The first algorithm is *progressive* and the second algorithm is progressive only for *read-only* software transactions. We show how some of the metadata accesses in these algorithms can be performed *non-speculatively* without violating opacity. We evaluate implementations of these algorithms on Intel Haswell, which does not support non-speculative accesses inside a hardware transaction, and IBM Power8, which does.

**1.2 Concurrency-optimal data structures**

To be pessimistic or optimistic? [44, 45]. Lock-based implementations are conventionally *pessimistic* in nature: the operations invoked by processes are not “abortable” and return only after they are successfully completed.
TM abstraction is a realization of optimistic concurrency control: speculatively execute transactions, abort and roll back on dynamically detected conflicts. But are optimistic implementations fundamentally better equipped to exploit concurrency than pessimistic ones? Our work makes the following contributions: (i) We provide a framework to analytically capture the inherent concurrency provided by two broad classes of synchronization techniques: pessimistic implementations that implement some form of mutual exclusion and optimistic implementations based on speculative executions. (ii) We explore the concurrency properties of search data structures which can be represented in the form of directed acyclic graphs exporting insert, delete and search operations. We prove, for the first time, that pessimistic (e.g., based on conservative locking) and optimistic serializable (e.g., based on serializable transactional memory) implementations of search data-structures are incomparable in terms of concurrency. Specifically, there exist simple interleaveings of sequential code that cannot be accepted by any pessimistic (and resp., serializable optimistic) implementation, but accepted by a serializable optimistic one (and resp., pessimistic). Thus, neither of these two implementation classes is concurrency-optimal. Our results suggest that “semantics-aware” optimistic implementations may be better suited to exploiting concurrency than their pessimistic counterparts.

**Concurrency-optimal list and binary search tree [46, 4]**. We propose the first provably concurrency-optimal data structure, the Versioned list. We show that the previous most efficient lists are not concurrency-optimal in that they reject schedules of memory accesses that would not violate consistency. To this end, we consider the classic set implementation as an example and show that, unlike the Harris-Michael [55] and the Lazy list-based sets [51], the Versioned list is concurrency-optimal in that it accepts all correct schedules. In addition, the Versioned list is probably the fastest list algorithm to date. It builds upon a new pre-locking validation technique that exploits versioning and try-locks to achieve high performance of update operations and to reduce the overhead of read-only operations. We implement our algorithm in Java 8, using the new StampedLock, and in C11, exploiting the stdatomic intrinsics, and show that it outperforms the Harris-Michael algorithm, the Lazy list algorithm and the Selfish optimization of Fomitchev and Ruppert’s algorithm [35] on Power8, SPARC and x86-64 architectures.

We also present the first concurrency-optimal implementation of a binary search tree (BST). The implementation is based on a standard sequential implementation of an internal tree, and it ensures that every schedule, i.e., interleaving of steps of the sequential code, unless linearizability is violated. To ensure this property, we use a novel read-write locking scheme that protects tree edges in addition to nodes. Our implementation outperforms the state-of-the art BSTs on most basic workloads, which suggests that optimizing the set of accepted schedules of the sequential code can be an adequate design principle for efficient concurrent data structures.

**Ongoing and future work**

All the projects described above concerns the asynchronous crash-stop shared memory model. Below, I outline some ongoing shared memory work concerning adversary models in the crash-recoverable and malicious domains.

**1.3 Beyond crash faults in shared memory**

**Concurrent data structures for non-volatile memory (NVM)** It is expected that current volatile memory based on DRAM will be augmented by storage-class memories (SCM) that are non-volatile and byte-addressable. The primary advantage of this hardware development is that it removes the need for two distinct file formats: the in-memory object format and the persistent file format for the block-oriented traditional persistent storage à la NAND flash that is prevalent in today’s solid-state devices. Yet, whether the data structure is designed directly on NVM or via a combination of DRAM plus NVM, i.e., DRAM with a NVM backup on account of the DRAM crash failure, there remain several open questions concerning the design of efficient persistent concurrent data structures. Firstly, the data structure state must be constantly updated in the non-volatile memory so that in the event of a crash failure, the computation may re-start from the most recent consistent state of the data structure. This write-back to the NVM must be atomic so that the recovered data structure state is consistent. Secondly, this raises the following question: what must be representation of the data structure in the NVM? For e.g., in a sorted linked-list-based set, it may be sufficient to store the set of values contained in the set, as opposed to an unsorted one since the pointer references can not be deterministically re-created during the re-start procedure invoked after the crash-recovery.

**Computing in oblivious memory.** We consider the problem of computing in an asynchronous shared memory that is oblivious [42]: no information is divulged to an adversary about the access patterns of the processes to the shared memory. Besides the fact that this a nontrivial problem to formulate and solve, it had wide ranging applications
especially in the context of today’s cloud computing services. Consider a server hosting a shared memory for client processes to access by invoking operations of a distributed algorithm. However, clients would wish that the server or the other clients be oblivious to their individual access patterns. Specifically, a data structure implementation is said to be oblivious if any two equal-length sequences of operations invoked are *computationally indistinguishable* to anyone but the client, a solution that typically involves employing *homomorphic encryption* schemes [39]. One of my goals is formalizing the notion of obliviousness in the distributed setting as well as deriving algorithms and complexity bounds for implementing oblivious non-blocking cloud applications.

2 Message-passing model

Ongoing and future work

2.1 Multi-actor semantics for programming scalable cloud services

**Distributed programming model for cloud services** [78]. A major challenge in writing applications that execute across hosts, such as distributed online service applications, is to reconcile (a) concurrency (i.e., allowing components to execute independently on disjoint tasks), and (b) cooperation (i.e., allowing components to work together on common tasks). A good compromise between the two is vital to scalability, a core concern in distributed applications.

The actor model of computation is an accepted programming model for such distributed applications in that actors can execute in individual threads (concurrency) and interact via asynchronous message passing (collaboration). However, this makes it hard for programmers to reason about *combinations* of messages as opposed to individual messages, which is essential in many scenarios.

We present a variant of the actor model in which messages can be composed into atomic units, which are executed in a strictly serializable fashion, whilst still retaining a high degree of concurrency. In short, our model is based on an orchestration of actors along a directed acyclic graph which supports decentralized synchronization among actors based on their actual interaction. We present an actor-based programming model, Atomic Events and Ownership Network (AEON), based on a dynamic DAG-inducing referencing discipline and implemented as an extension to C++. Concretely AEON provides the following properties: (i) **Programmability**: programmers need only reason about sequential semantics when reasoning about concurrency resulting from multi-actor events; (ii) **Scalability**: its runtime protocol guarantees *serializable* and *starvation-free* execution of multi-actor events, while maximizing parallel execution; (iii) **Elasticity**: supports *elasticity* enabling the programmer to transparently migrate individual actors without violating atomicity or entailing significant performance overheads. We implemented a highly available and fault-tolerant prototype of AEON in C++. We present formal operational semantics which proves serializability and the absence of deadlocks. Extensive experiments show several complex cloud applications built atop AEON significantly outperform others built using existing state-of-the-art distributed cloud programming protocols. According to the experiments, AEON is about 3x faster than similar programming models (EventWave [24] and Orleans [22]).

**Programmable Elasticity for Actor-based Cloud Applications** [79]. *Serverless computing* allows developers to program elastic cloud applications consisting of *stateless* functions that can be automatically scaled in and out, thus freeing them from managing servers. However, many cloud applications are *stateful* — while executing, the state of one function needs to be shared with another. Providing elasticity for stateful functions is much more challenging, as an elasticity decision (e.g., migration) for a stateful entity will affect others in ways which are hard to grasp without any application knowledge. We have implemented a programming framework for elastic stateful cloud applications that includes (1) an elasticity programming language as a second “level” of programming (complementing the main application programming language) for writing elasticity rules, and (2) a novel semantics-aware elasticity management runtime that tracks program execution and acts upon application features as suggested by elasticity rules. Extensive evaluation shows that our framework significantly improves the efficiency of a large class of multi-actor service applications, e.g., achieving same performance as a vanilla setup with 25% fewer resources, or speeding up computations by 24% with fixed resources.

2.2 Resilient multi-party protocols

The consensus problem in which processes initially propose a value and must *eventually agree* on one of the proposed values [34] is the most fundamental problem in large message-passing distributed systems. A general version of the
consensus problem would be Secure Multi-party Computation (MPC) \[87\] in which mutually distrusting parties (or processes) perform computations on private inputs without revealing any new information other than the result of the computation. While I have primarily worked only on classic consensus, I am actively exploring general problems within the MPC space under a wide class of adversary and timing models. More recently, I am exploring implementations of distributed algorithms using Trusted Computing Modules which has become extremely relevant since Intel released Software Guard Extensions (SGX) \[25\] that enables secure remote computation among other use-cases. Yet, what sort of trusted computing modules must be provided by hardware manufacturers that can benefit a maximal set of distributed applications that are resilient against malicious adversaries? This is a problem space I have made some preliminary progress and intend to continue working on.

Next, I outline one specific variant of consensus of recent interest I have worked on; however I am always interested in exploring domain-specific variants of consensus in wide-ranging adversarial models.

**Byzantine generalized consensus** \[76\]. The Paxos \[67\] protocol for reaching agreement in a distributed system has made its way to the core of the implementation of the services that are used by millions of people over the Internet, in particular since Paxos-based state machine replication is the key component of Google’s Chubby lock service \[21\], or the open source ZooKeeper project \[59\], used by Yahoo! among others.

One of the most recent members of the Paxos family of protocols is Generalized Paxos. This variant of Paxos has the characteristic that it departs from the original specification of consensus, allowing for a weaker safety condition where different processes can have a different views on a sequence being agreed upon. However, much like the original Paxos counterpart, Generalized Paxos does not have a simple implementation. Furthermore, with the recent practical adoption of Byzantine \[82\] fault tolerant protocols in the context of blockchain protocols, it is timely and important to understand how Generalized Paxos can be implemented in the Byzantine model. In this work, we make two main contributions. First, we attempt to provide a simpler description of Generalized Paxos, based on a simpler specification and the pseudocode for a solution that can be readily implemented. Second, we extend the protocol to the Byzantine fault model and provide the respective correctness proof.

### 2.3 Concurrency within smart contract ecosystems

Smart contracts promise the reinvention of the monetary circuit by applying tamper-resistant computer system concepts to the financial sector, thus introducing efficiency and communal monitoring of real time transactions. Such contracts not only provide a fundamental building block of digital currencies, they are also the cornerstone of future generations of market economy and online exchanges with applications beyond the financial sector. Typically, users employ digital wallets to inject digital transactions into a distributed network and continuously extend a distributed data structure that maintains the list of transactions issued by all users over time. The vitality of this ecosystem is conditional on the security, privacy and scalability of all its components: the distributed network, digital wallets, programming and runtime environments that allow the specification of arbitrary smart contracts, digital currency exchanges and other domain-specific protocols required for integration with the overlying application.

While the span of potential future applications are promising, open research problems remain to be addressed in order to transition to real-world applications, the most crucial being the end-to-end security of transactions. Below, I outline one particular smart contract model I have worked on, but I am interested in addressing security, privacy and scalability challenges that are inherent to emerging smart contract system models.

**Asynchronous and highly concurrent Payment-Channel Networks** \[72\] Cryptocurrencies like Bitcoin \[73\] and Ripple \[83\] have grown as a possible avenue for secure decentralized online payments and credit exchange between arbitrary pairs of processes in a distributed system. It is expected that any cryptocurrency synchronization protocol needed to execute secure financial transactions provide resilience against Byzantine adversaries, i.e., computing entities exhibiting malicious behavior. Unlike traditional protocols for Byzantine agreement typically require knowledge of the set of participating processes \[23\], protocols for cryptocurrencies are expected to work in a peer-to-peer setting in which several processes may join or leave the network arbitrarily. In such cryptocurrencies, processes continuously extend a distributed data structure called a blockchain that maintains the list of transactions issued by processes over time. Protocols like Bitcoin achieve agreement over the blockchain data structure by forcing processes to solve a proof-of-work \[31\]: a cryptographic puzzle that essentially allows the Bitcoin to tradeoff computation for communication complexity.

However, permissionless blockchains based on global consensus such as Bitcoin are inherently limited in transaction throughput and latency. Current efforts to address this key issue focus on off-chain payment channels that can be combined in a Payment-Channel Network (PCN) to enable an unlimited number of payments without requiring to
access the blockchain other than to register the initial and final capacity of each channel. While this approach paves the way for low latency and high throughput of payments, its deployment in practice raises several privacy issues as well as technical challenges related to the inherently concurrent nature of payments, such as race conditions and deadlocks, that have been understudied so far.

In this work, we lay the foundations for privacy and concurrency in PCNs, presenting a formal definition in the Universal Composability framework as well as practical and provably secure enforcement solutions. In particular, we present *Fulgor* and *Rayo*. Fulgor is the first payment protocol for PCNs that provides provable privacy guarantees for PCN and is fully compatible with current Bitcoin. Nevertheless, Fulgor is a blocking protocol and therefore prone to deadlocks of concurrent payments as in currently available PCNs. Instead, Rayo is the first protocol for PCNs that enforces non-blocking progress (i.e., at least one of the concurrent payments terminates). We show through a new impossibility result that the latter property necessarily comes at the cost of breaking anonymity. At the core of Fulgor and Rayo is Multi-Hop HTLC, a new smart contract, compatible with Bitcoin, that provides conditional payments while reducing running time and communication overhead with respect to previous approaches.

### 3 Distributed network algorithmics

The ever increasing availability of IoT devices and applications necessitates communication protocols that scale commensurately. The management and control of these interconnecting networks is heavily strained as billions of transitive IoT devices are deployed and new applications are dynamically introduced into enterprise networks. I have initiated projects on distributed network algorithmics: addressing algorithmic protocols are the heart of the modern networking stack from the lens of distributed computing theory, systems and verification. These projects concern: (i) resilient and robust protocols for enforcing consistent data plane updates via distributed network controllers; (ii) algorithms for smart, secure and scalable edge-computing infrastructure; (iii) distributed estimation of network topology and provenance of network state; and (iv) architectures for high-fidelity and programmable network experimentation infrastructure.

### Ongoing and future work

#### 3.1 Distributed network policy update protocols

In many Software-Defined Networking (SDN) deployments, the control plane ends up being actually centralized, yielding a single point of failure and attack. This paper models the interaction between the data plane and a distributed control plane consisting of a set of failure-prone and potentially malicious (compromised) control devices, and implements a secure and robust controller platform that allows network administrators to integrate new network functionality as with a centralized approach. Concretely, the network administrator may program the data plane from the perspective of a centralized controller without worrying about distribution, asynchrony, failures, attacks, or coordination problems that any of these could cause. We introduce a formal SDN computation model for applying network policy updates and show that it is impossible to implement asynchronous non-blocking and strongly consistent SDN controller platforms in that model. (i) We present a protocol with provably linearizable semantics for applying network policy updates that is resilient against faulty/malicious control devices as long as a correct majority exists, and (ii) a modification to the protocol that improves performance by relaxing the guarantees of linearizability to exploit commutativity among updates. Extensive experiments conducted with a functional prototype over a large networked infrastructure supporting OpenVSwitch-compatible SmartNIC hardware show that our strongly consistent protocol induces bearable overhead. In fact, it achieves higher throughput in most cases investigated than the Ravana [61] platform which addresses only benign (crash) failures. In ongoing work, I am extending our algorithms to support a dynamic controller set: controllers may crash or get compromised and should be restored or replaced, and the system may expand to new geographical regions bringing in new controllers [50, 29]. While existing solutions have been presented in a general distributed systems sense I plan to explore them in an SDN specific context to determine if the SDN model can provide optimizations to existing solutions. As part of this, we plan to develop a comprehensive set of asynchronous protocols to allow control group modifications leveraging techniques from distributed key generation [60] and threshold signatures [38].

1 à la George Varghese’s lecture on ”Life in the Fast Lane: Viewed from the Confluence Lens” [86].
3.2 Protocols for secure and distributed edge-computing infrastructure

By introducing computing and storage capabilities at the Internet’s edge (e.g., cloudlets, micro-clouds, fog nodes 18, 80, 81), edge computing promises the delivery of highly responsive and privacy-preserving applications. Dispersed cloudlets located in appropriate proximity to edge devices serve as a control and data nexus that allows the “smart edge” ecosystem to grow commensurate with the demands of its applications and users. Enabling edge computing requires many fundamental capabilities; many of which are not natively supported in our current Internet protocols. Our work adopts a holistic approach to making the edge of the Internet more secure and scalable by proposing a novel protocol suite well-suited to the demands of edge computing and data. We take a clean-slate approach to the co-design of an edge-centric forwarding plane and control plane, decoupled from one another, that can evolve independently of end-user applications and the specific mechanisms used at the physical (PHY)- and medium-access control (MAC) layers. Specifically, I am working on the following: (i) Simplify datagram forwarding and resource discovery using novel approaches to interval routing [37, 40]; (ii) An integrated approach to secure and reliable transport and caching of content that eliminates the need for routers to maintain transport-level state or end-to-end connections.

3.3 Distributed network estimation

Accurate and timely estimates of network characteristics is crucial to scaling network performance, e.g., via installation of dynamic network policy updates to adapt routing algorithms, traffic congestion protocols, enforce Quality-of-Service requirements, detect malicious traffic, etc. The fundamental prerequisite to estimating network performance characteristics is deriving an accurate view of the network topology. At its heart, this is a distributed computing problem: reaching agreement on the structure of the network topology from edge-based localized topology observations. Specifically, consider a network with \( n \) nodes, specified by the set \( N = \{1, 2, \ldots, n\} \). For each node \( i \), suppose there are a few nodes specified in neighbor set \( N_i \) which have direct link to node \( i \), i.e. broadcasting from nodes in set \( N_i \) is received at node \( i \). Each node has its own view about the network topology, and each node shares its view with its neighbors in order to improve local views and eventually agree on the global topology. For several Internet-like graphs, I am currently exploring computability and complexity results for distributed network topology estimation. Looking ahead, I am exploring distributed techniques for inferential network monitoring, for e.g., computing individual link delays from derived network topology and path-level delays.

3.4 Research and development infrastructure for federated network testbeds and high-fidelity experiment orchestration [43]

I am involved in the Merge software infrastructure [43] to interconnect individual computing nodes into an evolving ecosystem, capable of supporting rigorous system-level cybersecurity experimentation for the evolving needs of the cybersecurity research community. The architecture provides the following features: (i) a Constraint-based expression model to define experiments with semantics that directly capture validity; (ii) find and allocate resources that fall within validity region of a constraint-based experiment via graph embedding algorithms for mapping experiment network topologies against physical testbed infrastructure; (iii) orchestrates the materialization of an experiment across multiple testbed sites; (iv) incorporate new types of devices dynamically and give resource providers proactive control over resources and give researchers instant global visibility of the resource space. The Merge infrastructure currently supports the orchestration of the Dispersed Computing Testbed [2], a testbed facility that provides advanced network modeling and emulation capabilities, allowing researchers to deploy systems in high operational-fidelity environments.

4 New directions: Beyond the horizon

In his wonderfully sarcastic critique of the scientific community in *His Master’s Voice* 2, the great Polish writer Stanisław Lem refers to a specialist as a barbarian whose ignorance is not well-rounded. While I have attempted at becoming a specialist on all things distributed, I am culturally not totally ignorant of innovations that are needed in both the problem and solution space of distributed systems and their application across the STEM fields. Of course, it is far easier to explain work which I have already done than define the problem and solution space for questions I would like to address in the future. Pretty much every research topic listed in this statement has unresolved questions concerning almost all aspects of distributed computing: tight bounds, verification and faster prototypes. Looking

---

ahead, I am very much interested and committed to understanding concurrency phenomena inspired by *quantum computing, neuromorphic computing* and *optical networking*. Moreover, given the richness and diversity of emerging computational trends, I envision periodically re-orienting my research priorities in working towards distributed models and algorithms for enforcing safety, security and privacy inspired by domain-specific application trends.

**References**


