Verifying distributed system implementations (Summary)

Distributed systems underlie most uses of a modern computer, including cloud computing, e-commerce, social networking, and content delivery services; it is critical that these systems operate correctly. Implementing distributed systems is difficult because they must handle both concurrency and failure: machines may arbitrarily crash, and networks may reorder, drop, or duplicate packets, creating numerous opportunities for unexpected corner cases. Further, popular services must support hundreds of thousands of simultaneous requests which requires highly optimized and scalable software.

Despite the importance of distributed software, we lack a coherent framework and methodology for their correct construction. The state of the art is little more than code review, test, and pray, even though the set of possible behaviors is often too complex to permit good test coverage. The result is predictable: critical systems fail, leading to data loss, major service outages, and embarrassment. On a single day this summer, the NYSE halted trading, United grounded flights, and the Wall Street Journal website failed, all due to errors in distributed systems.

Our goal is to change this reality: to develop the tools and techniques needed to make verifying highly-tuned distributed systems *implementations* practical. Previous research has focused on verifying simplified models of algorithms, leaving open the question of whether implementations match the model. We aim to close this formality gap and convert the practice of building distributed systems to a verification-based approach that eases the burden for programmers to implement correct, high-performance, and maintainable distributed systems.

Intellectual merit To achieve our goals, we will develop a verification framework, implemented in the popular Coq proof assistant, that enables formally reasoning about distributed system *implementations* in realistic fault models. We will initially focus on developing techniques to compositionally verify safety properties where the programmer can prove correctness for applications and reusable fault-tolerance mechanisms independently, and then compose the proofs to make strong end-to-end guarantees. We will extend our framework to verify rich specifications for a broad class of critical systems, including reconfiguration and system updates to dynamically add nodes or replace software in a running system, vertical composition to modularly optimize system layers via observational refinement, and horizontal composition to compose entire systems via rely/guarantee reasoning. We will further support verifying *liveness* properties to ensure systems always service client requests even in the face of failures, and explore evolving consistency guarantees where systems switch between strong and weak consistency to maintain quality of service.

We will develop a new verification methodology to achieve these goals. Re-proving systems as they evolve is the most difficult aspect of large verification efforts, but the topic is relatively unexplored, leading to ad-hoc, labor-intensive approaches. Our key insight is *planning for change* by modularizing both framework and system architectures to better accommodate updating proofs in response to changes in specifications or code. Our methodology will span the entire proving process, including techniques to verify aspects of existing implementations, domain-specific languages to ensure modularity by construction, specialized tools and program logics to dispatch common proof obligations, and "structural properties" to guide tactic development toward more general and reusable automation.

We will validate our framework and methodology by working with practitioners to verify practical distributed systems. We tentatively plan to prove correctness of a crash-safe distributed key-value store, of the Raft consensus protocol with extensions such as log compaction, and of BGP routing.

Broader impacts Billions of people use distributed systems every day in critical applications ranging from health care to banking. When these systems fail, they threaten users' convenience, finances, security,

and safety. If successful, our work will benefit every corner of society by enabling the creation of high-performance, verified distributed systems. We will make all of our software publicly available, and integrate our research into education via use in classes and by training students in formal verification and distributed systems research.

Verifying distributed system implementations (Description)

1 Introduction

Billions of people depend on distributed systems every day for health care, banking, communications, transportation, e-commerce, entertainment, and more. These applications are implemented as distributed systems for scalability and availability: even when load spikes, machines crash, or networks misbehave, the system as a whole should continue servicing user requests. However, distributed systems still fail in practice. Such failures lead to data loss and major service outages that threaten users' convenience, finances, security, and safety. For example, in April 2011 a malfunction of failure recovery in Amazon Elastic Compute Cloud (EC2) caused a major outage that brought down web sites, such as Foursquare, Reddit, Quora, and PBS [Ama11, NYT11, Hig11]. As another example, commercially-available distributed datastores contain numerous safety bugs that can lead to data loss and inconsistency [Kin13]. On a single day this past summer, the NYSE halted trading, United grounded flights, and the Wall Street Journal website failed, all due to failures in distributed systems.

Distributed systems are difficult to implement correctly because they must handle both concurrency and failure: machines may arbitrarily crash and networks may reorder, drop, or duplicate packets, creating numerous opportunities for unexpected corner cases. Further, the most popular services need to support hundreds of thousands of simultaneous requests, creating a need for highly optimized and scalable software and often very subtle tradeoffs between responsiveness and consistency guarantees. Large companies like Google and Amazon address this challenge by employing highly trained experts with advanced degrees and years of experience to build their systems, and in addition they devote vast computing resources to brute-force testing. Errors still occur in production because the set of possible failures is too complex to permit effective testing or manual reasoning.

A promising avenue is to *prove* that the distributed system works correctly in all cases. This verification is a major research challenge, so researchers typically only prove the correctness high-level algorithms for simplified models of distributed systems [Lyn96]. Reasoning about such simplified models eases the verification burden by omitting gritty details, but many subtle errors arise from exactly such corner cases. Further, traditional pen-and-paper proofs are themselves large, complex artifacts prone to subtle reasoning errors. The difficulty of constructing proofs also tends to "freeze" their design: even small algorithmic or implementation changes often require significant effort to re-verify. This makes it difficult to keep verifications up to date with practical distributed systems that are constantly evolving to improve performance and meet new demands.

Goals Our goal is to enable the construction of much more reliable distributed systems by building a practical verification-based approach that eases the burden for programmers to implement correct, high-performance, and maintainable distributed systems. We address this challenge by co-designing a new **verification framework** for formally reasoning about distributed system *implementations* along with a new **verification methodology** and tools to guide effective *proof engineering* and to provide maintainability. We plan to implement our framework in Coq, a popular proof assistant which provides facilities for specifying, implementing, and verifying programs in an expressive language based on dependent types. Proofs in Coq are fully formal and machine-checked, ensuring that verifications in our framework provide strong guarantees. Further, we will automatically extract system implementations from Coq, ensuring that correctness guarantees hold for the code that is actually run rather than a simplified model.

Why Now? Over the past decade, there has been tremendous progress in mechanical proof assistants. To use these tools, a programmer implements their system in the proof assistant's programming language, specifies its correctness in the proof assistant's logic, and then interactively proves that the implementation always satisfies its specification. This approach has been used to develop practical compilers [Ler09b, TL10], operating systems [KEH+09, WLZ+14], web browsers [JTL12], and common servers like SSH [RRJ+14]. Multiple studies [LAS14, YCER11] have experimentally demonstrated that the formally-verified CompCert C compiler is substantially more reliable than traditionally-developed compilers such as GCC and LLVM. While these studies have uncovered thousands of bugs in traditional compilers, no compiler validation study to date has found a test case that causes CompCert to produce wrong code. Providing a similar level of reliability for distributed systems would provide significant societal and economic benefits.

Why Us? Our research team is uniquely suited to apply verification techniques to the domain of distributed systems, with expertise in proof-assistant-based formal verification (Tatlock), software engineering (Ernst), and distributed systems research (Anderson). Our past research includes several verification "firsts" within the proof assistant community: the first formally verified web browser [JTL12], the first fully automated verifications of security properties for SSH and web servers [RRJ+14], and the first automated techniques for formally verifying complex loop optimizations [TL10] in CompCert. Our students are all co-advised, allowing us to bridge the traditional silos of academic research.

1.1 Intellectual Merit

Formally verifying systems is difficult for two primary reasons, and our research agenda focuses on addressing each of these barriers. First, it is difficult to formalize the domain, initially develop the system, and prove it correct; for this, we develop a **verification framework**. Second, it is difficult to evolve verified systems; for this, we develop a **verification methodology**.

Verification Framework To support proving distributed systems, we will formalize both the semantics of nodes in a distributed system as message handlers, and also various fault models which include failures such as machine crashes and network misbehaviors. We will design our fault models to be composable: a fault model that allows packet drops could be combined with a model that allows packet duplication to produce a model where either type of failure may occur. This composability will enable users to reason precisely about their environment by choosing the failures their system must tolerate and combining the corresponding fault models.

Once we have formalized system semantics and fault models, we will focus on developing *verified system transformers* that enable compositional reasoning where the programmer can verify applications and reusable fault-tolerance mechanisms independently and then compose the proofs to make strong end-to-end guarantees. We will also develop techniques for modular fault handling, so that multiple verified fault-tolerance mechanisms for different types of faults can be composed into a single verified mechanism that handles all the fault types of its constituent mechanisms.

We will initially focus on verifying safety properties. One of our early tentative goals will be to formally verify *state machine safety* for a full implementation of Raft [OO14], a high-performance consensus protocol used in dozens of distributed systems [Raf].

We will extend our framework to increase expressiveness, including features such as reconfiguration to dynamically add nodes to a running system, vertical composition to modularly optimize system layers via observational refinement, and horizontal composition to compose entire systems via rely/guarantee reasoning. We will further extend the framework to support verifying *liveness* properties to ensure systems are available,

that is, they always service client requests even in the face of failures. Liveness properties such as availability are the primary objective of distributed systems, but are rarely proven even for high-level algorithms.

We will explore evolving consistency where systems switch between strong and weak consistency guarantees to maintain quality of service, tradeoffs between performance and correctness where probabilistic guarantees are established via sampling, and dynamic system updates where a new version of a system is deployed while running. We will also explore verification in more idealized models where proofs may be easier to construct, e.g. synchronous round-based models, and determine how such proofs may be soundly transferred to realistic, asynchronous fault models. These features will support verifying rich specifications for a broad class of critical systems.

Our case studies will gradually scale up from small fault-tolerance mechanisms such as a retransmission library and a primary-backup replication system, to larger systems like a crash-safe distributed key-value store, an optimized version of Raft that supports features such as log compaction, and a verified implementation of the Border Gateway Protocol.

Our case studies will help keep our goals in focus and provide useful examples of how our methodology (described below) can be applied to real systems. We plan to develop our framework iteratively, so that we always have a currently-useful version of our framework and the systems we have verified. As we iterate, we expect to explore additional research challenges including developing a verified serialization library for encoding bit-level messages on the wire.

Proof Engineering Methodologies Machine-checkable proofs are large, complex software artifacts whose construction requires heroic effort by highly trained experts. As with other software development, no matter how carefully one plans the verification effort, subtle issues arise that require revising specifications, implementations, and proof strategies. As a result, the most expensive aspect of large formal verification efforts in proof assistants is re-proving existing parts of a system. Surprisingly, this challenge is relatively unexplored, leading past efforts to adopt ad-hoc, labor-intensive approaches that have limited their success.

To ensure our framework can be successfully applied, our methodology will develop techniques to *plan for change* at all levels of the verification effort. We will modularize both framework and system architectures to better accommodate updating proofs in response to changes in specifications or code. Our methodology will span the entire proving process, including techniques to verify aspects of existing implementations, domain specific languages to ensure modularity by construction, specialized tools and program logics to dispatch common proof obligations, and "structural properties" to guide tactic development toward more general and reusable automation. The principles we develop will reduce the effort required to re-prove systems during the iterative verification process and as systems evolve over time.

1.2 Broader Impacts of the Proposed Work

Our proposed research will benefit several different communities:

- **For Distributed System Users: More Reliable Systems** Billions of people use distributed systems every day in critical applications ranging from health care to banking. When these systems fail, they threaten users' convenience, finances, security, and safety. If successful, our work will benefit every corner of society by enabling the creation of high-performance, verified distributed systems.

- **For Distributed System Developers: Better Tools** Distributed systems developers today are in a "can't win" situation: they are tasked with building software that can scale, be highly responsive, and never fail, yet they are provided none of the tools needed to do so. Our framework and methodology are targeted at

them. We will make all of our software publicly available under a permissive license, such as the MIT or BSD license.

Providing correctness guarantees will also enable distributed systems developers to experiment with confidence. Developers often know exactly what sorts of faults their system needs to tolerate, but lack a way to specify that fact. They also often know which optimizations will be most profitable for their application, but currently, making any changes to fault-tolerance mechanisms presents serious risks since testing is not able to ensure correctness.

- **For Students: Bridging Silos** We will train students to understand the challenges and opportunities of distributed systems and to be facile with formal proofs, verification, and dependent type theory. We have already started incorporating Coq into multiple UW graduate classes, to enthusiastic students response. Colleagues at Penn, Princeton, and MIT are already using our ideas and tools in their classes.

  Integrating our research into education is key to training the the next generation of engineers in a better approach. We will also train PhD, Masters, undergraduate, and high school students doing research in formal verification and distributed systems.

2 Verification Framework

Our goal is to create a framework that can verify optimized distributed system implementations, provide flexible fault models, and enable programmers to build modular, layered systems. For performance reasons, real-world implementations often diverge in important ways from their high-level descriptions [CGR07]. Previous automated reasoning tools like TLA+ [Lam02] and Alloy [Jac12] can analyze abstract *models* of distributed algorithms, but few practical distributed system *implementations* have been formally verified. One challenge is that distributed systems run in a diverse range of environments. For example, some networks may reorder packets, while other networks may also duplicate them. It is important that engineers be able to verify their systems in the fault model most appropriate to their production environment. Our framework will enable the programmer to separately prove correctness of application-level behavior and correctness of fault-tolerance mechanisms, and to allow these proofs to be easily composed.

Below we describe an early prototype we built to explore the initial feasibility of our research program. To radically improve the current practice of constructing distributed systems, our initial prototype must be extended account for the complexities of the real world and we must develop new techniques to ensure service level agreements and liveness properties are satisfied, enable dynamically modifying guarantees in response to load spike, and support online update.

Prior Work Our initial explorations into the design space for a distributed system verification framework produced Verdi [WWP+15b], a prototype Coq toolchain for writing executable distributed systems and verifying them. By directly reasoning about running code, Verdi avoids a *formality gap* between the model and the implementation. While Verdi does not provide many of the features required to change the state of the art today, it serves as a strong foundation from which we will launch our research program. Below we briefly describe two of the key ideas from Verdi which have provided initial encouraging success: *network semantics* for formalizing fault models and *verified system transformers* for decomposing certain classes of application-level verification from fault-tolerance verification.

Network Semantics The correctness of a distributed system relies on assumptions about its environment. For example, one distributed system may assume a reliable network, while others may be designed to tolerate

$$\frac{H_{\text{inp}}(n,\ \Sigma[n],\ i) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle i,\ o \rangle)}\ \textsc{Input} \qquad \frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}\ \textsc{Msg}$$

$$\frac{p \in P}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus \{p\},\ \Sigma,\ T)}\ \textsc{Dup} \qquad \frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P,\ \Sigma,\ T)}\ \textsc{Drop} \qquad \frac{H_{\text{tmt}}(n,\ \Sigma[n]) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle \text{tmt},\ o \rangle)}\ \textsc{Time}$$

Figure 1: Example Network Semantics.

packet reordering, loss, or duplication. To enable programmers to reason about the correctness of distributed systems in the appropriate environment model, we will provide a spectrum of *network semantics* that encode possible system behaviors using small-step style derivation rules.

We encode nodes as *message handlers* that continuously respond to input from their host and other nodes in the system. These message handlers are reasoned about in the context of a network semantics. The network semantics captures both the way messages are exchanged over the network as well as the faults that nodes and messages may experience.

Figure 1 illustrates an example network semantics which models message reordering, duplication, and drops. A network semantics is defined as a step relation on a "state of the world", which may differ among network semantics, but must always include a *trace* of the system's external input and output, which are used when specifying and verifying a system's behavior. In Figure 1 the state of the world is represented as a tuple (P, Σ, T) where P is a bag of in-flight packets that have been sent by nodes in the system but have not yet been delivered to their destinations, Σ is a map from node identifiers to their local state, and T is the trace of all I/O actions performed by any node of the system. The squiggly arrow $\rightsquigarrow$ between two states indicates that a system in the state of the world on the left of the arrow may transition to the state of the world on the right of the arrow when all of the preconditions above the horizontal bar are satisfied.

Each node in a system communicates with other processes running on the same host via input and output. Nodes can also exchange *packets*, which are tuples of the form (source, destination, message). The behavior of nodes is described by three handler functions H_{inp}, H_{net}, and H_{tmt} which handle input, message exchange, and timeouts respectively.

Verdi's current fault models are not *algebraic*; only one of them can be used at a time, and there is no way to combine them. In our proposed framework, we will design our fault models to be algebraic, so that a fault model which only allows packet drops can be combined with a model that only allows packet duplication to produce a model where either type of failure may occur. This composability will enable users to reason precisely about their environment by choosing the failures their system must tolerate and combining the corresponding fault models.

Verdi's network semantics currently elide some low-level network details. For example, input, output, and packets are modeled as abstract datatypes rather than bits exchanged over wires, and system details such as connection state are not modeled. This level of abstraction simplifies Verdi's semantics and eases both implementation and proof. We will develop lower-level semantics and connect them to the semantics our prototype provides via system transformers, as described below. This will further reduce our framework's trusted computing base and increase our confidence in the end-to-end guarantees our framework provides. We will also explore verification in more idealized models where proofs may be easier to construct, e.g. partially synchronous round-based models [CS09], and determine how such proofs may be soundly transferred to realistic, asynchronous fault models via new verified system transformers.

Verified System Transformers

Verdi provides a *compositional* technique for implementing and verifying distributed systems by sep-

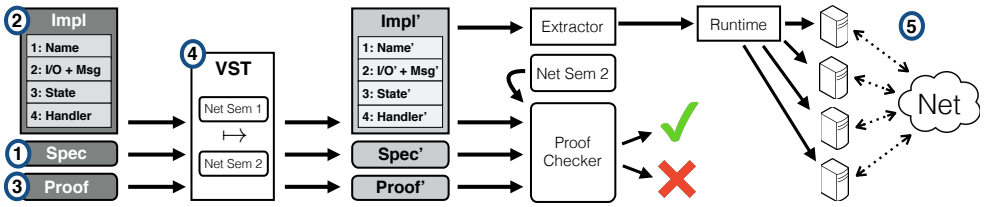

Figure 2: Verified System Transformer Workflow.

arating the concerns of application correctness and fault tolerance. This simplifies the task of providing end-to-end guarantees about distributed systems. To achieve compositionality, we introduce *verified system transformers* (VSTs). A system transformer is a function whose input is an implementation of a system and whose output is a new system implementation that makes different assumptions about its environment. A verified system transformer includes a proof that the new system satisfies properties analogous to those of the original system. For example, a programmer can first build and verify a system assuming a reliable network, and then apply a transformer to obtain another version of their system that correctly and provably tolerates faults in an unreliable network (e.g., where machines may crash).

Figure 2 illustrates the workflow of a programmer applying VSTs. The programmer ① specifies a distributed system and ② implements it by providing four definitions: the names of nodes in the system, the external input and output and internal network messages that these nodes respond to, the state each node maintains, and the message handling code that each node runs. ③ The programmer proves the system correct assuming a specific baseline network semantics. For example, the programmer may choose an idealized reliable model for this proof: all packets are delivered exactly once, and there are no node failures. ④ The programmer then selects a target network semantics that reflects their environment's fault model, and applies a verified system transformer (VST) to transform their implementation into one that is correct in that fault model. This transformation also produces updated versions of the specification and proof. ⑤ The verified system is extracted to OCaml, compiled to an executable, and deployed across the network.

We have already verified a handful of realistic fault-tolerance mechanisms as VSTs including a primary-backup system, a simple key-value store, a lock server, and a retransmission library which correctly tolerates reconnections (a feature which is not provided by raw TCP).

VSTs are also useful during the verification process. A perennial issue in any proof by induction is ensuring that the induction hypothesis is sufficiently strong. All relevant parts of the state of the system must be constrained in order for the proof to go through. For some properties, a strong enough induction hypothesis can only be achieved by adding state to the system, often referred to as "ghost state". Verdi implements ghost state as a system transformer, which, given an original system and a ghost handler, produces a new system whose state consists of both the original state and the ghost state, and whose event handlers modify both the original state and the ghost state. The system transformer also shows once and for all that it is safe to add ghost state without affecting the execution of the original system, that is, it does not change the trace.

Our proposed framework must provide a wider array of network semantics and VSTs and account for the complexities of the real world, techniques to ensure service level agreements and liveness properties are satisfied, facilities to dynamically modify guarantees in response to load spike, and mechanisms for online updated. We discuss some of these challenges below as a sequence of Research Questions (RQs) that our proposed research program will investigate.

RQ 1: How can the framework be extended to support the complexity of real systems?

In practice, distributed systems must account for *reconfiguration* where nodes dynamically enter and leave

the system, *vertical composition* where subtle optimizations are added to achieve the highest performance, and *horizontal composition* where a system coordinates with other distributed systems. Each of these features will require new techniques to add support in our framework.

Reconfiguration. Previous frameworks have only considered verifying distributed systems with an unchanging number of nodes [WWP+15b, CAB+86, SRvR+14]. In practice, these results will not apply because systems must accommodate nodes leaving and joining the system in response to maintenance tasks and demand-based provisioning.

Supporting dynamic system reconfiguration is difficult for two reasons. First, it is not clear how such dynamic behavior should be formalized: can nodes enter and leave the system at any time? How should new nodes be identified, and can node identifiers be re-used? Second, reconfiguration significantly complicates the core invariants used when verifying a system. For example, in the case of consensus, the core invariants involve agreement among a majority of nodes. As the number of nodes in a system dynamically changes, the definition of "majority" shifts as well. This leads to subtler invariants that complicate the linearizability and liveness guarantees provided by consensus [OO14]. These guarantees are not only more difficult to prove for the fault-tolerance mechanism, but also complicate the guarantee provided to applications layered on top of the consensus protocol.

Our past work on verifying servers for protocols such as SSH and HTTP also dealt with an unbounded number of system components [RRJ+14]. In these systems, an application kernel running synchronously on a single machine could dynamically spawn new helper components and coordinate their interaction in servicing client requests. We formalized this unbounded number of system components as a list of channels used for communication coupled with invariants characterizing the status of each component. We also developed rich automation to generalize invariants over an unbounded number of components.

We will investigate applying these techniques to the setting of distributed systems. Unlike the synchronous, single-threaded setting of our previous work, distributed systems are asynchronous and concurrent. Thus, any given node may not be aware of other nodes entering and leaving the system. Handling these challenges will require developing new formalizations of components entering and leaving the system as well as new proof techniques to account for the added complexity of dynamic system invariants.

Optimizing Vertical Composition. Practical distributed systems employ numerous optimizations to achieve the highest performance. For example, "message piggypacking" is used to combine multiple message exchanges between nodes into a single exchange. It is often not clear a priori whether these performance-motivated optimizations continue to satisfy a system's correctness guarantees. Another issue is that distributed algorithms are often specified at a high level and omit details that developers must decide how to implement. For example, it is sometimes unclear how often a system must persist its state to disk in order to preserve crash safety. To simplify proofs, our framework will initially include VSTs that persist state on every operation, which eases the burden of carrying out crash safety proofs but also clearly reduces performance. Adding optimizations to remove unnecessary persistence operations will require adding new VSTs. Since the performance of many distributed systems is bounded by local and network I/O rather than CPU, I/O optimizations such as message piggybacking and efficient persistence will make the performance of verified systems competitive with that of their unverified equivalents. This will encourage adoption of our framework and of verification techniques more broadly.

At a high level, we will adopt the well known technique of *observational refinement* from the compiler community [Mil89, Ler09a]. In this technique, a system may modify or avoid expensive operations as long as a developer can prove that the optimized system's functional behavior is indistinguishable from the correct, unoptimized version. We have previously applied this technique in the context of automated compiler optimization verification [TL10] and will explore how it can be extended to the context of distributed systems.

Unlike the compiler setting, the nature of faults in distributed systems prevents simple "property-

transference style" refinement proofs from applying. In a verified compiler, an engineer typically verifies a transformation t by proving that for any source program s which satisfies any property P, the transformed version of the program $t(s)$ must also satisfy P, formally $\forall s\ P,\ P(s) \rightarrow P(t(s))$. However, this style of guarantee is not generally applicable in distributed systems, because the property P may require a system to successfully accomplish some action that could fail in a lower level network semantics. Thus, we will also explore the notion of *property transformers* which safely translate higher level behavioral specifications to the lower level setting where optimizations are implemented.

Horizontal Composition. Large distributed systems are often composed of smaller distributed systems cooperating to support the application-level logic of a system. For example, a banking system may use a lock server to avoid data races on account information accessed through a distributed key-value store. Ensuring the correct operation of the composed system demands verifying each system against the guarantees provided by the other.

This situation is similar to the setting of concurrent program verification where each thread is only proven to operate safely under the assumption that other threads in the system satisfy certain well-behavedness specifications [OG76]. We have explored this in previous work in the context of aliased mutable data [GEG13].

We will investigate applying such rely/guarantee style reasoning to the context of distributed systems. One major challenge will be formalizing the notion of an "external client" in our framework in order to connect it to the actual implementation of another system. When systems are verified in isolation, external clients are assumed to exhibit arbitrary behavior. This will limit the properties that can be verified about an application which is built on multiple distributed system implementations. To complete the proofs, we will need to define rich interfaces that characterize the behavior of external clients which actually correspond to nodes of another distributed system, and thus will only exhibit the behaviors allowed by that system.

RQ 2: How can service level agreements and liveness properties be verified?

Distributed systems are designed to maintain availability in the face of failures. Service providers typically make availability guarantees to clients in terms of *service level agreements*. For example, Amazon EC2 may guarantee that a customer request will receive a response within 100 milliseconds 99.99% of the time. From a theoretical perspective, availability is a type of *liveness property* since it asserts that a system will eventually display some desired behavior (in contrast to safety properties, which state that the system never displays undesired behaviors). These liveness properties are notoriously difficult to prove, even informally on paper, since in a non-deterministic system (such as a system communicating over an unreliable network), verifying that desired behaviors eventually occur requires reasoning over traces of unbounded length and assuming "fair" behavior on the part of the communication medium. It is thus difficult to formally prove that a system will meet its service level agreements.

To ensure that service level agreements are met, we will extend our framework's support for reconfiguration to reason about load and provisioning. Load corresponds to the rate of incoming client requests relative to the currently allocated resources. Load changes dynamically as traffic drops and spikes and machines crash or come online. Providers must carefully provision resources to ensure that they will always be able to meet their commitments to clients' service level agreements. Fundamentally, service level agreements can only be verified under assumptions that limit the number of faults that will occur. In network semantics that allow arbitrary, unbounded failure, all machines could theoretically crash simultaneously, making it impossible to guarantee availability. Thus we will extend our network semantics with *failure constraints* that characterize maximum failure rates. Failure constraints allow users to express many important guarantees, such as Raft's liveness specification: "Assuming no more than half the nodes in the system fail, the system will remain available." Just as with the network semantics, we will provide an array of flexible failure constraints so that engineers can choose the assumptions most appropriate to their environment.

Guaranteeing service level agreements against failure constraints will also require formalizing provisioning: a provider must be able to express the set of resources they can access and how those resources are allocated in response to changes in load. To support provisioning, we will add *provisioning handlers* to our network semantics which allow system architects to express how new nodes are brought online and integrated into a system when load spikes or failures occur. We will implement this formalism by connecting it to existing techniques for monitoring and estimating system status, e.g. Nagios.

RQ 3: Can we verify dynamically changing consistency guarantees to maintain quality of service?

Distributed collaboration services and control systems need to minimize latency: operations must apply immediately, even when nodes are offline. Traditional *strong consistency* models [Lam98] roughly require that all nodes execute the same operations in the same order. This restrictive model greatly simplifies reasoning, but increases latency and cannot support offline nodes. In contrast, *weak consistency* models [SS05, LFKA] allow nodes to execute operations in different orders, and thus avoid latency and can support offline nodes. However, weak consistency models often fail to prevent certain undesirable behaviors, and thus force application developers to handle anomalous behaviors. This tradeoff between availability and consistency is a fundamental consequence of the CAP theorem [LG02].

Many applications include both operations which must be *responsive*, providing an answer immediately, even if it is incorrect [DHJ$^+$], and also operations which must be *consistent*, where an incorrect response could lead to catastrophic failure. We will explore techniques to verify application-level guarantees that take advantage of weak consistency to continue providing responsiveness without violating consistency of sensitive operations.

Furthermore, in some situations high-level applications may actually be better able to handle failures than lower-level fault tolerance mechanisms. For example, in routing algorithms, application-level code can sometimes make better strategic choices when latency and link-status information are available [KAB$^+$07]. Exposing clients to failure will be especially valuable if the underlying fault-tolerance mechanisms can provide sufficient performance gains in exchange. Such performance benefits are likely since significant complexity arises from ensuring that each operation is always executed correctly.

Developers should ideally be able to achieve the best of both worlds, building systems that are strongly consistent during normal operation but that can safely switch to a weakly consistent model when necessary (for instance, by only making some operations available until the system returns to a known good state). To ensure application invariants are preserved, such systems must notify their clients when operations are only tentative, as opposed to committed in a strongly consistent manner. Once normal operation resumes, the system can return to a strongly consistent state by reconciling any conflicts arising as a result of the weakly-consistent phase. Unfortunately, such systems are currently difficult to develop. We will investigate techniques based on operational transformation [SE98] within our framework to verify that they correctly regain strong consistency after weakly-consistent periods, ensuring that application invariants are maintained.

RQ 4: Can exposing applications to occasional failure improve performance and ease verification?

We will go beyond exploring dynamically changing consistency guarantees, where clients are notified when their operations are tentative, to also explore systems where clients can be exposed to some faults without notification. Past work on distributed systems has focused on providing precise guarantees that insulate applications from all faults by ensuring the correctness of every operation. Such architectures simplify development by abstracting away failure, leading to cleaner designs where all fault handling is factored out of application code.

However, for many systems, e.g. in entertainment and social networking, performance can trump absolute correctness, and engineers may be willing to tolerate application failure at some low rate. Indeed, many applications already tolerate error on common operations; e.g. when incrementing an ad-click counter, a lost

operation may mean lost revenue, but may be justified by improved performance.

The challenge in exposing these failures is that it complicates the interface between the fault-tolerance mechanism and the clients. We will investigate how such interfaces can be formalized, how clients can be constructed and verified against these interfaces, and what performance benefits may be provided for end-to-end system performance. Recent results have explored how consistent transactions can be implemented on top of inconsistent replication [ZSS+]. We will verify a similar system at a much smaller and simpler scale as an initial case study and then investigate techniques to generalize to distributed systems more broadly.

We are particularly interested in interfaces where underlying fault tolerance mechanisms provide probabilistic correctness guarantees as discussed in TACT [YV02]. Past work has shown that probabilistic guarantees can significantly ease *automated* reasoning about a system by allowing for *sampling* to be used in establishing weaker guarantees [SPM+14]. We will explore how to verify implementations of such systems and how a client can cope with weaker probabilistic guarantees.

To explore this avenue, we will determine what fault models are amenable to sampling, and how to formalize randomized sampling to make meaningful, whole-execution guarantees based on sampling from individual failure events. Next we will investigate how application code built on probabilistic fault-tolerance mechanisms may cope with probabilistic guarantees by determining what style of specification and proof is most appropriate. We will characterize the performance benefits that can be achieved by relaxing guarantees to be probabilistic.

From a proof engineering standpoint, exposing clients to probabilistic guarantees seems to entail much more complex proofs. Instead of attacking such proofs directly, we will use sampling to establish probabilistic guarentees to some degree of confidence. A major challenge in this approach will be determining how to sample from complex error distributions. To address this challenge we will develop techniques for overapproximations that allow our sampling based techniques to be carried out while making conservative guarantees.

RQ 5: How can systems be dynamically updated while maintaining safety and availability?

As the software stack evolves, components will require updates and global policies will change. Policy changes typically must be adopted atomically across the distributed components of the system. If only a subset of the nodes are updated, the entire system may exhibit behaviors that are not possible in either the original or updated policies. Such atomic updates have been explored in the single node system setting [HSS+14] and in software-defined networking [MHvF15], but remain challenging in the distributed context because of asynchrony and node failures [Ong14]. For example, a message might be delayed arbitrarily in the network and then delivered after an update has occurred, or a node might crash while applying an update. Furthermore, if updating a node requires taking it offline, then updating all nodes simultaneously will cause the system to become unavailable.

Updating distributed systems has been previously studied [Blo83], but building correct update mechanisms remains challenging in practice. The NYSE failure this summer was due to a failed software upgrade. Because it is such a delicate process, systems may "stop the world" for update, which is often a conservative choice for safety properties, but can violate liveness and availability requirements. Like many code refactoring and maintenance tasks, distributed updates also change system behavior and thus cannot directly apply standard refinement- or equivalence-based proof techniques. Instead, updates should maintain behavior for aspects of the system that are not modified, and should preserve crucial system-wide invariants.

A basic approach to correct distributed update builds on consensus: once an update has been propagated to a majority of the nodes in the system, all interactions switch to the updated version. This technique has been adopted in popular implementations of consensus, e.g., the Raft protocol [Ong14]. Operations originating from out-of-date nodes are ignored, and crashes are handled using Raft's normal fault-tolerance

mechanisms. Note this approach maintains availability so long as updates are performed to a minority of nodes at a time.

A more ambitious approach is to support each node running continuously and applying *adapters* to messages from out-of-date nodes, transformations that translate interactions from old versions of a protocol to safe interactions with updated nodes. This will be crucial for providing continuous operation even for nodes that lag behind or are unable to perform updates (e.g., a legacy system, a mobile node where update is too expensive, or a node under control of a third party). Developing protocols to handle multi-version coordination is extremely complex, and thus their implementations should be formally verified.

3 Tools and Verification Methodology: Planning for Change

Over the past decade, the research community has verified increasingly large and complex systems including compilers [Ler09a, TL10], operating systems [KEH+09], networks [GRF13], browsers [JTL12], and reactive systems [RRJ+14]. Each of these projects required heroic effort from highly-trained teams working at institutions with extensive in-house expertise. For example, the seL4 verified OS kernel took over 20 person-years to verify roughly 9,000 lines of C code [KEH+09], not counting the years of research to lay the groundwork in formalizing C and developing refinement proof techniques.

Not only have past verification efforts been extremely expensive, but even after completion the resulting systems are typically difficult to improve and maintain. The primary challenge is the cost of updating proofs as definitions change: when code or specification is modified, the developer must typically make many updates throughout the verification's proofs. Because mechanical proofs are extremely intricate and not designed for readability, it is difficult to anticipate the consequences of a change.

Scaling verification to systems spanning hundreds of thousands of lines of code and under constant update will require new methodologies and tool support. To address these challenges, we will investigate how tools can help developers cope with change by speculatively exploring the consequences of changes before the developer even makes them; explore techniques to ease both the initial verification burden and maintenance tasks by adapting techniques to verify existing implementations; develop techniques to support modular, local reasoning for verifying distributed systems; articulate a set of design principles for developing robust tactic libraries; and build tools to help developers refine their specifications and code during the iterative development process.

RQ 6: What speculative analyses and development environment features help accommodate change?

Current development environments for proof assistants are extremely primitive. These tools do not automate common tasks such as refactoring. Furthermore, they do not inform the developer of the consequences of changes the developer may make.

As in other development settings, engineers often spend considerable time discovering the consequences of a change. Frequently, one fix leads to the creation of additional problems, but it may take the developer hours, days, or even weeks to discover these consequences.

We have previously developed speculative analyses (see Section 7.1). A speculative analysis tool analyzes potential developer changes in the background. When a developer is about to make a change, they can learn the consequences immediately rather than only after working hours or days with an ill-advised change.

We will explore how speculative analyses can more quickly inform the developer of the consequences of a change during the verification process. In a concurrent project with colleagues at UC San Diego, we built a prototype development environment called PeaCoq [Pea] which performs very local, small-scale speculative analyses. At each step of the interactive proving process, PeaCoq displays a set of potentially applicable tactics along with how each would advance the proof state. When Tatlock used this tool in a

class on programming languages and formal verification, the tool helped Coq novices more quickly learn the available tactics. Expert users have also found PeaCoq valuable as it will often find a clever use of a tactic that completes a proof more quickly or simply.

We will build upon these past successes to further explore how speculative analyses may improve tool support for engineers making higher level changes to a system's code or specification. In addition to adapting traditional development environment features to the setting of proof assistants, we will develop analyses to quickly reflect the consequences of changes, such as reflecting the number of proofs which become invalid as a consequence of an edit or learning from example changes in one proof to automatically update all similar proofs for related theorems as in WitchDoctor [FGL12].

RQ 7: Can we provide guarantees about the behavior of existing, unverified systems?

Our previous work formally verifying web browsers [JTL12] and popular servers like SSH [RRJ+14] developed the *formal shim verification* (FSV) technique. In FSV, an existing, unverified system is partitioned into components which are then each executed in a secure sandbox that prevents them from directly interacting and accessing sensitive system resources. A small *application kernel* is implemented and formally verified in Coq and mediates interactions between the system components and coordinates their interaction with system resources. Using this architecture, we were able to formally verify that the resulting systems satisfied many important security properties including various forms of non-interference, access control, and data integrity [JTL12, RRJ+14]. A key benefit of this architecture is that the sandboxed system components (which may be implemented in notoriously unsafe languages like C) can be *arbitrarily modified* and the updated system will continue to satisfy its security policies *without any changes to proofs*. We will explore applying FSV in the context of distributed systems where these benefits would enable us to make strong guarantees about existing distributed systems implementations and dramatically reduce the cost of change.

A major challenge in applying FSV to distributed system implementations arises from their highly concurrent nature. Application kernels developed in past instances of FSV have been synchronous and single-threaded. Careful design allowed these sequential kernels to achieve good performance on single-host applications, but these designs will not translate to a distributed setting where the kernel must wrap system components on multiple nodes simultaneously and communication may not be reliable. We will explore how this design may be implemented in our proposed framework to enable effective verification that still accommodates easy change to the implementations of system components.

By applying FSV to existing distributed systems we will both make it easier to initially verify systems, since only a simple application kernel needs to be implemented and verified, and also easier to update the system, since changes can be made to the sandboxed components without requiring any changes in proofs. However, FSV is only appropriate for verifying a limited class of specifications [JTL12]. Thus we will also develop additional techniques, as described below, which apply to distributed systems and formal verification more generally.

RQ 8: How to support modular, local reasoning for distributed systems?

As discussed in the previous section, our framework will provide *verified system transformers* (VSTs) to separate fault tolerance mechanisms from application-level logic. By design, this separation will allow changes to either the application or the fault tolerance mechanism *without requiring any changes in the other*. However, we are also interested in providing similar modularity within application or VST implementations.

When re-verifying a system after a change, it is important to take advantage of the implementation's structure and reason about its parts in a modular way. While carefully following good software engineering principles can help enforce modularity, a better approach would be to design a domain-specific language where modules are independent by construction. Changes to systems written in such a DSL are easier to prove correct, because each component can be verified in isolation. To explore this design space, we will develop a

domain-specific language (DSL) for implementing distributed systems that supports modular reasoning.

In our past efforts we have designed custom DSLs for *reactive systems* where a program responds to messages from the outside world in a single event handling loop. By carefully designing this DSL, we were able to achieve complete proof automation for verifying an SSH server and web server. We conducted experiments making changes to the servers and, when the changes were correct, the proofs were automatically reconstructed *with no manual proof effort*.

To develop a similar DSL for distributed systems, we will begin by following the successful methodology we developed for reactive systems, but will adapt our approach for challenges of distributed systems which must deal with asynchrony and failure. We will first investigate system-specific ways of supporting modular reasoning, e.g. by proving that any property preserved by every event handler of a system is preserved by the system as a whole. Then, we will generalize our approach to be system-agnostic instead of system-specific, by leveraging the structure enforced by our DSL. This will support decomposing single message handlers into their components, which can be verified in isolation. We will next apply techniques from concurrent separation logic to create a distributed program logic [SNB15b, SNB15a]. This will reduce global proof obligations about the relationship between nodes to local proof obligations about individual handlers.

RQ 9: What principles underlie the development of general, robust tactic libraries?

The basic building blocks of a proof in Coq are *tactics*: scripts that manipulate the theorem proving context closer to an easily provable goal. Large verification developments typically include extensive libraries of custom tactics used to discharge proof obligations common to the domain. This leads to *proof maintenance* concerns: engineers constantly add new theorems and adjust existing invariants, strengthening them to make them provable by induction, and weakening them to make them true. As definitions and theorem statements change, tactic scripts that use them tend to break. Fixing brittle tactics is difficult and tedious and thus hampers maintenance tasks. We will investigate how custom tactic libraries can be structured to mitigate this burden.

After years of frustration with current practice, we have developed a new design principle for tactic development: *structural tactics*. Tactic libraries often break due to small changes to how theorems are stated or to the shape of a proof context during interactive verification, e.g. when an automatically-generated hypothesis name changes. Our initial experience shows that tactics are more robust when they satisfy the structural properties used in proof system design. The structural properties support patterns such as *weakening* which ensures that a proof will continue to be valid whenever additional hypotheses are added. Structural tactics are also required to never depend on auto-generated hypothesis names or hypothesis ordering. Anecdotally, while developing a prototype version of the framework described here, we designed our tactic library to satisfy these structural properties and found that the benefits to proof robustness far exceeded the initial design costs.

Structural tactics should also help users debug their proofs when automation inevitably fails. While proof automation is extremely useful when it successfully solves a goal, when it fails the results are often opaque, presenting little or no information to the user about what went wrong. Since automation is typically based on heuristics, seemingly irrelevant changes can break or fix proof automation. We will ensure that our tactics are as robust and predictable as possible, erring on the side of failing with a readable error message rather than succeeding in a brittle fashion.

A major challenge with validating our approach is that there are no agreed-upon criteria for comparing the effectiveness of tactic libraries or design principles. This is largely due to the system-specific nature of most tactic libraries. But the lack of any quality metrics or benchmarks hinders the development of a general approach to developing best practices. Thus, to evaluate the benefits of structural tactics, we will first explore techniques to evaluate and compare existing approaches. The POPLmark challenge [ABF$^+$05] provides an ideal starting place for gathering a diverse set of tactics from different groups all solving the same

problem. After developing a set of criteria for evaluating tactic library effectiveness, we will compare our structural tactic approach against (1) traditional tactic development, which relies heavily on built-in tactics and uses auto-generated hypothesis names [PCG+15, Ler09a]; (2) reflection-based techniques, including Ssreflect [GMT09] and MirrorShard [MCB14], which focus on domain-specific decision procedures, (3) type-safe tactic programming languages, such as Mtac [ZDK+13] and VeriML [SS10], and (4) approaches which emphasize heavy automation, as examplified by Chlipala's approach [Chl13].

RQ 10: What tools can help engineers develop specifications and discover inductive invariants?

Conceptually, the first step of verification is developing a specification. However, in practice, specification and implementation often influence one another during the iterative verification process.

Sometimes specifications are too strong, making them logically impossible to implement. Discovering this problem can take weeks or months of effort, during which large parts of the system are implemented and partially verified. When specification issues are uncovered, addressing them often requires re-proving numerous key invariants about all of the partially verified system components. Thus, any techniques to accelerate the process of identifying specification issues will significantly improve the verification process.

Sometimes specifications are too weak, in that they are not inductive, or strong enough to imply itself from all states that satisfy it. Discovering the right way to strengthen a specification to become inductive while remaining implementable is a major challenge that requires numerous expensive iterations and significant proof churn.

To help bootstrap the verification process by providing specific inital guesses, we will apply dynamic invariant detection [ECGN01, EPG+07], a run-time analysis that observes system traces and performs machine learning over these observiations. It produces properties that were always true of the observed executions. These techniques are automated, which drastically reduces programmer effort in comparison to the manual, labor-intensive state of the

Dynamic invariant detection has been combined with multiple theorem-provers to produce a more-automated system [NE02a, NE02b, NEG+04], but not to one as rich as Coq. Furthermore, the general approach has been extended to distributed systems [BBS+11, BBE+11a, BBE+11b, BBEK14, LPB15], but without any connection to a formal proof. We plan to combine these two avenues of research. We also plan to extend Daikon to identify mostly-true invariants [DLE03] that hold for a high percentage of a program's executions and thus are likely to be close to the right specification.

A major challenge in applying our past work to the context of distributed system verification will be extending it to discover *inductive invariants*. Such invariants are not only true of all reachable states, but if the invariant holds on a state *s*, then that fact implies it will hold on the states *s* can step to. Note this inductiveness is usually quite difficult to establish. If the property is too strong, then it will be inductive, but false in the base case. If the property is too weak, it will be true in the base case, but not inductive. A significant portion of developer effort in verification is spent attempting to identify the right level of strengthening to make an invariant inductive while keeping it implementable. We will extend Daikon to automatically find stronger versions of regular invariants which can be suggested to the programmer as likely inductive invariants.

4 Case Studies

We will validate our framework and methodology by verifying practical distributed systems. Below we describe our plan to verify a production-ready implementation of the Raft consensus protocol, a crash-safe distributed key-value store, and a verified implementation of the Border Gateway Protocol (BGP). These case studies will help keep our goals in focus and provide useful examples of how our tools and methodology can be applied to real systems. We plan to develop our framework iteratively, so that we always have a

currently-useful version of our framework and the systems we have verified.

Case Study 1: Raft Raft [OO14] is a *state machine replication protocol*; it ensures that a state machine is consistently replicated across the nodes in a system, which allows external clients to interact with the system as if it were a single node running a single copy of the state machine. This replication enables the system to continue serving clients even in the presence of failures. In particular, Raft guarantees that if a majority of nodes remain available, the system as a whole will remain available. Since consensus provides strong consistency and availability even when machines crash, Raft and other consensus protocols are at the heart of a wide array of crucial distributed systems [Cor14, Bur06, CDG+08].

Raft operates by replicating a log across system nodes and electing a distinguished leader to service client requests and coordinate replication. In Raft, the leader receives state machine operations (e.g. reads and writes) from clients and appends them to a local log of operations. The leader then replicates this log of operations to the other nodes in the system. The leader can only respond to a client with results once all log entries up to and including the client's request have been replicated to a majority of nodes. This ensures that each node will apply operations to its copy of the state machine in the order established by the log.

Raft's primary safety property guarantees that clients see a *linearizable* view of the replicated state; if the system responds to client operation o, the results of all later client operations are guaranteed to reflect the execution of o. Raft also provides a liveness guarantee: if there are sufficiently few failures, then the system will eventually process and respond to all client operations.

We will verify both Raft's safety and liveness guarantees. In our initial prototype, we have already partially verified the safety of a prototype VST (verified system transformer) implementing a simplified version of Raft. Our prototype VST lifts a system implemented in a state machine semantics, in which a single process responds to input from the outside world, into a system where the original state machine is consistently replicated across a number of nodes. The Raft transformer targets a semantics in which all messages can be arbitrarily duplicated, delayed, or dropped, and in which machines can crash and reboot. Completing the verification of Raft in such an adversarial network semantics will provide strong correctness guarantees for operation in real networks. Our prototype does not yet support critical features such as *log compaction*: saving space on disk and in memory by truncating the suffix of the log committed by a majority of nodes. Verifying log compaction will serve as a useful study in adding important, real-world optimizations to verified distributed systems.

Case Study 2: Key-Value Store To validate our methodology, it is important to verify and measure systems composed of verified application code with verified fault-tolerance mechanisms. This will ensure that our systems compose as we intend, that our interfaces are implementable, and that our modularity techniques do not introduce unacceptable overhead. In principle, the theory of linearizability used in reasoning about Raft is part of its specification, and is thus trusted. However, this theory can be removed from the trusted computing base by showing that it implies an application-specific end-to-end specification. To this end we will verify a distributed, crash-safe key-value store.

We will implement a high-performance distributed key-value store on top of our verified Raft implementation. We have explored an initial small prototype that accepts `get`, `put`, and `delete` operations as input. When the system receives input `get`(k), it outputs the value associated with key k; when the system receives input `put`$(k,\ v)$, it updates its state to associate key k with value v; and when the system receives input `delete`(k), it removes any associations for the key k from its state. Internally, the mapping from keys to values is represented using an association list.

To improve performance, we will change the implementation of our store to use a hash table or an efficient tree structure. We will evaluate how our methodology facilitates this change and what impacts the optimization has on proofs. We will integrate lessons learned back into the development of our methodology.

Combining the key-value store with the replication transformer provides an end-to-end guarantee for a

replicated key-value store without requiring the programmer to simultaneously reason about both application correctness and fault tolerance. We aim to develop optimizations that enable our verified key-value to perform comparably to etcd, an industrial key-value store used in many popular web services. Our initial experiments are promising for spinning-disk-based systems, but several low-level optimizations will be necessary to match etcd's performance for modern SSD-based systems. Exploring techniques to prove such low-level optimizations will serve as a stress test of our framework and methodology.

Case Study 3: Verified BGP Routing The Internet is made up of Autonomous Systems (ASes), such as ISPs, that exchange routing information — the paths traffic can takes across the Internet — via the Border Gateway Protocol (BGP). An AS configures its routers to process BGP route announcements so as to maximize the AS's profit, performance, availability, and security. For example, a policy might mandate that announcements from one neighbor must be preferred over those of another.

Writing router configurations to correctly implement an AS policy is challenging because the router configurations are large [Int], frequently changing [TLSS10], distributed, and written in configuration languages with little or no static checking. Goldberg [Gol14] surveys several major outages caused by router misconfigurations. In 2008, YouTube was inaccessible worldwide for 2 hours due to a misconfiguration in Pakistan [Bro08]. In 2010, China Telecom hijacked a significant but unknown fraction of international traffic for 18 minutes [Cow10], which was covered by a Congressional report [Sla10] and mass media [McC10]. In 2014, China Telecom hijacked a similarly unknown fraction of Russian traffic for several hours [Mad14].

We have started to formalize the BGP specification RFC 4271 [RLH06]. We will use this specification, along with our verification framework, to prove the correctness of an AS policy over a network of BGP routers. This will involve stating the correctness of a network of routers with respect to an AS policy, as well as verifying that routers follow the BGP specification. Because BGP operates significantly differently from consensus or applications such as key-value stores, this case study help evaluate the expressiveness of our framework and ensure that we are not overfitting our formalizations to a narrow range of systems. We will also compare the techniques most effective in verifying AS policies to those for Raft and our key-value store and generalize lessons learned to broadly applicable design patterns. AT&T and Microsoft have both expressed interest in using our techniques to verify their BGP policies and configurations.

5 Educational Impact and Curriculum Development

We have already adapted two existing UW graduate courses, CSE 505 (graduate programming langauges) and CSE 506 (advanced graduate programming languages) to be based on the Coq proof assistant. CSE 505 is a "quals class" taken by about 1/3 of all graduate students at UW, and this has led to a surprisingly large group of proficient Coq users among graduate students and several advanced undergraduate students at UW. Inspired by this success, Tatlock along with our colleagues Xi Wang (UW) and Bryan Parno (Microsoft Research) is developing a new graduate course, CSE 599, focused on formal verification of systems to be offered this spring. All our course materials are publicly available.

From these experiences we have learned that students are most excited to work on real projects, and thus we will define small sub-tasks of our research questions as suggested class projects. Not only will this help motivate students, but it will advance our research and provide useful guidance concerning how our methodology can be enhanced to help non-experts in verifying systems.

For advanced students, Tatlock facilitates a student-led reading group which has been exploring recent results in homotomy type theory [Uni13], an exciting new branch of mathematics developed collaboratively at the Princeton Institute for Advanced Study in the context of the Coq proof assistant which explores fundamental issues concerning the notion of equality and its implications for formal proof. While seemingly

esoteric, this challenging material has greatly enhanced the verification skills of our graduate students.

We will also explore introducing more proof assistant-related material at the undergraduate level. Tatlock has taught CSE 341 (undergraduate programming languages) for the past two years and each time devotes one of the final lectures to demonstrating Coq proofs to show students the expressive power of dependent type theory. Several strong undergraduate students have responded by enrolling in the Coq-based graduate programming languages course, and three of these students are now actively involved in research projects within our group, including Steve Anton who is working on the material discussed here.

Tatlock has also been collaborating on PeaCoq [Pea], a new web-based development environment for Coq. In particular, Tatlock used an early prototype of PeaCoq in his graduate programming languages class. Based on this experience, he has become deeply involved in the project and contributed several design improvements to help beginning Coq users more easily learn how to construct proofs.

Ernst taught graduate programming languages, software engineering, compilers, and program analysis at MIT and UW. Some of these offerings have incorporated — or created! — the research tools cited in this proposal. Ernst has introduced use of advanced type systems and verification into CSE 331 (undergraduate software design), leading to greater appreciation of formal methods, and Tatlock will retain this when he teaches the class. Ernst has devised multiple classes and educational activities that are used in colleges nationwide. One example is the innovative "Groupthink Specification Activity" that teaches specifications in a fun interactive environment.

Anderson has taught UW's graduate distributed systems class for over a decade. He will integrate distributed systems verification and other concepts from this grant into his curriculum.

6 Related Work

Extending the related work discussed throughout the proposal above, we briefly survey the most closely related work on distributed system verification, proof-assistant-based verification, and verification methodologies.

Distributed systems verification EventML [Rah12] provides expressive primitives and combinators for implementing distributed systems. EventML programs can be automatically abstracted into formulae in the Logic of Events, which can then be used to verify the system in the NuPRL proof assistant [CAB+86]. The ShadowDB project implements a total-order broadcast service using EventML [SRvR+14]. The implementation is then translated into NuPRL and verified to correctly broadcast messages while preserving causality. A replicated database is implemented on top of this verified broadcast service, but database itself is unverified.

Bishop et al. [BFN+06] used HOL4 to develop a detailed model and specification for TCP and the POSIX sockets API, show that their model implements their specification, and validate their model against existing TCP implementations. This work is complementary to our proposal which focuses verifying applications and fault tolerance mechanisms against network semantics that are assumed to correctly represent the behavior of the network stack.

Ridge [Rid09] verified a significant component of a distributed message queue, written in OCaml. His technique was to develop an operational semantics for OCaml which included some basic networking primitives, encode those semantics in the HOL4 theorem prover, and prove that the message queue works correctly under those semantics. Unlike in Verdi, the proofs for the system under failure conditions were not machine checked.

Ensemble [Hay98] layers simple *micro protocols* to produce sophisticated distributed systems. Like Ensemble micro protocols, our system transformers implement common patterns in distributed systems as modular, reusable components. Unlike Ensemble, our systems transformers come with correctness theorems that translate guarantees made for one network semantics to analogous guarantees for another

semantics. Ensemble enables systems built by stacking many layers of abstraction to achieve efficiency comparable to hand-written implementations via cross-protocol optimizations, a feature we will emulate in our proposed framework. Ensemble micro protocols have been manually translated to IO automata and verified in NuPRL [LKvR+99, HLvR09]. In contrast, our approach will provide a unified framework that connects the implementation and the formalization, thus eliminating formality gaps.

Verified SDN Formal verification has previously been applied to software-defined networking, which allows routing configurations to be flexibly specified using a simple domain specific language (see, e.g. Verified NetCore [GRF13]). Verifying SDN controllers involves giving a semantics for the OpenFlow configuration language, switch hardware, and network communication. Verified NetCore focuses on correct routing protocol configuration, while our framework will be focused on the correctness of distributed systems that run on top of the network.

Specification Checking There are many frameworks for formalizing and specifying the correctness of high-level distributed algorithms [GL00, Pet77, SW01]. One of the most widely used frameworks is TLA, which enables catching protocol bugs during the design phase [Lam14]. For example, Amazon developers reported their experience of using TLA to catch specification bugs [NRZ+14]. Another approach of finding specification bugs is to use a model checker. For example, Zave [Zav12] applied Alloy [Jac12] to analyzing the protocol of the Chord distributed hash table [Zav12]. Lynch [Lyn96] describes algorithm transformations which are similar to Verdi's verified system transformers. In contrast, we will focus on ensuring that *implementations* are correct and show that the actual running system satisfies its intended properties.

Testing and Model Checking There is a rich literature in debugging distributed systems. Run-time checkers such as Friday [GAM+07] and D^{3}S [LGW+08] allow developers to specify invariants of a running system and detect possible violations on the fly or offline. Model checkers such as Mace [KAB+07, KAJV07], MoDist [YCW+09], and CrystalBall [YKKK08] explore the space of executions to detect bugs in distributed systems. These tools are useful for catching bugs and easy for developers to use.

For example, Mace provides a full suite of tools for building and model checking distributed systems. Mace's checker has been applied to discover several bugs, including *liveness* violations, in previously deployed systems. Mace provides mechanisms to explicitly reason across abstraction boundaries so that lower layers in a system can notify higher layers of failures. Our proposed framework will enable stronger guarantees via full formal verification.

Systems verification Several major systems implementations have been verified fully formally in proof assistants. The CompCert C compiler [Ler09a] was verified in Coq and repeatedly shown to be more reliable than traditionally developed compilers [YCER11, LAS14]. Our system transformers are directly inspired by the translation proofs in CompCert, but adapted to handle network semantics where faults may occur.

Bedrock [Chl11], Ynot [NMS+08], and the Verified Software Toolchain [ADH+14] are verification frameworks based on separation logic and are useful for verifying imperative programs in Coq. The Verified Software Toolchain separates proof interfaces from implementations; by exposing only a small set of "API" axioms (for instance, those of separation logic), the Verified Software Toolchain enables automated proofs for many proof obligations. We will adapt this approach in our methodology for verifying distributed systems.

The seL4 OS kernel [KEH+09] was verified using the Isabelle/HOL proof assistant. As in our proposed verification of Raft, the bulk of the effort in verifying seL4 went into proving invariants about the internal state of the system; these invariants are then used to prove a forward simulation from the abstract specification to the C implementation, just as in our verification, Raft's state invariants are used to prove equivalence up to linearizability between the underlying state machine trace and the replicated trace.

Proof engineering There is a small body of work on improving the development of machine-checked proofs. Chlipala's Certified Programming with Dependent Types [Chl13] advocates for heavy proof automation, using powerful, purpose-built tactics to dispatch proof obligations. We see our methodology as a compromise between the "tactic soup" style of many proofs (in which little effort is made to support maintainability or automation) and Chlipala's heavy automation which is difficult to develop and debug.

The Ssreflect library [GMT09] provides an alternative to Coq's default tactic language for developing maintainable proofs. Many of Ssreflect's custom tacticals involve "bookkeeping", that is, managing the hypotheses which appear in the goal and in the context. As in our proof methodology, Ssreflect discourages the use of automatically-generated hypothesis names (hypothesis names generated by Ssreflect's tactics are reserved identifiers, and cannot be used in tactic scripts). Unlike our proof methodology, Ssreflect's tactics focus on *removing* hypotheses from the context when they are no longer needed, and on giving important hypotheses clear names. In contrast, our proof methodology allows users to avoid referring to specific hypotheses at all by using tactics which find the correct hypothesis wherever it is in the context.

7 Results from prior NSF support

7.1 Michael Ernst

Ernst's most recent NSF grant is "Speculation and continuous validation for software development" (CCF-0963757; $766,000; 9/1/2010–8/31/2014, with a no-cost extension to 8/31/2015).

The goal of this project was to discover programmer errors before they occur. We proposed "speculative analysis": program analysis (such as compilation, testing, bug-finding, etc.) that is performed on multiple potential future versions of the software. The analysis guesses how the programmer might change the program, makes such a change on a copy of the codebase, evaluates the consequences of that change, and informs the developer of the consequences before the developer actually makes the change.

Intellectual Merit: In addition to proposing the novel idea of speculative analysis, we implemented six different speculative analyses, plus a framework for creating new speculative analyses.

We created a speculative analysis for version control, Crystal, which unobtrusively informs developers whether their changes are consistent or inconsistent with their teammates' changes [BHEN11b, BHEN11a, BMH+12, BHEN13, MBNC14].

We created a speculative analysis for compilation errors, Quick Fix Scout, which helps programmers decide how to fix compilation errors [MBH+12a, MBH+12b]. It makes the Eclipse Quick Fix menu indicate, for each Quick Fix, how many compilation errors it fixes. Thus, a developer is less likely to choose a Quick Fix with negative consequences that the developer would have to undo later.

We created a speculative analysis for database consistency, Continuous Database Testing, that catches database update errors when they are made, before they cause widespread corruption [MBM13, MBM15].

We created a speculative analysis for test order dependencies, in which one test fails or succeeds based on whether another test has been run [MSW11, ZJW+14]. Techniques such as test prioritization generally assume that the tests in a suite are independent; we showed that this common assumption is untrue in practice.

We created a speculative tool, Cascade, for performing type inference and refactorings [VPEJ15, DDE+11]. Unlike batch-mode tools, Cascade runs incrementally, permitting programmer interaction.

We created a framework that creates a continuous analyses by wrapping an offline analysis — one that requires a user to manually invoke it and wait for the results [MBEN13, Muş13, MBEN15]. This "codebase replication" technique maintain a shadow version of the code that is kept up-to-date with the developer's IDE. We built four continuous analyses — for testing, for bug detection, for code style, and for data race detection

— by wrapping existing offline analyses. We showed that codebase replication enables new approaches to understanding and manipulating the development history of a software project [MSBE15].

For each of the above projects, we implemented the analysis in a publicly-available tool, and we performed extensive empirical evaluation via automated experiments and human studies.

Broader Impacts: For most computer programs, the cost of constructing it dominates the cost of running it. Experimental results demonstrate that our work reduces the cost of creating computer programs. By helping programmers, our work can indirectly improve the lives of everyone who uses computers.

Our work on the Crystal speculative tool inspired development of an internal tool at Microsoft, aimed at the Bing development group. In a recent independent survey of software development practitioners [LNZ15], our Crystal speculative tool was ranked highest of all software engineering research in terms of percent of developers (62%) who thought the work was essential.

The project's publications [DDE+11, BHEN11b, MSW11, BMH+12, MBH+12a, MBH+12b, Muş13, BHEN13, MBEN13, MBM13, MBNC14, ZJW+14, MBM15, VPEJ15, MSBE15, MBEN15] include an ACM Distinguished Paper Award.

We make our tools and data publicly available; see the links at http://homes.cs.washington.edu/~mernst/pubs/.

7.2 Thomas Anderson

Anderson has received several NSF grants over the past five years. None are closely related to the current work, but the closest is "FIA: Collaborative Research: NEBULA: A Future Internet That Supports Trustworthy Cloud Computing" (Award: 1040663, 2010-1013, $753K). Nebula is designed to provide secure, highly available, and robust communication services to critical services in the emerging cloud and mobile environment. Unlike the existing Internet, Nebula provides interdomain route selection and QoS. A spectrum of malicious attacks against the network, such as DoS, is no longer possible with Nebula, and the system allows for rapid repair after unintentional and byzantine failures. Unlike other clean slate projects, Nebula involves the minimal set of changes to the network to accomplish this high assurance.

The publications resulting from this award include four SIGCOMM papers, one NSDI paper, and two HotNets papers: [PJZ+14, LHKA13, HLP+13, PKC+12, SHS+12, LHKA11, PKRS11].

Intellectual Merit: A primary contribution is in the area of interdomain routing. We have developed two concrete proposals. They both bind resources to a path enabling QoS and DoS resilience. The first, called TorIP, achieves reliability by drastically restricting the scope of action for an ISP. Packets are encrypted so as to disclose to each ISP only that the packet is destined for the next hop ISP; thus, an ISP cannot conduct certain Byzantine attacks because it lacks the information to do so. A second proposal is called Arrow. It does not try to hide packet destinations from intermediary ISPs, but like TorIP, it contracts with each ISP in the path to ensure that the packets will be delivered reliably.

In addition, we developed a system, called F10, which can route around failures in 10s of microseconds for highly structured data center networks. This fits with TorIP and Arrow as follows: using resilient intradomain routing protocols, ISPs will be able to offer endpoints resilient service across their networks. TorIP and Arrow can stitch these offerings together to provide resilient end to end service across the future Internet.

Broader Impact: If adopted, NEBULA would have substantial broad impact in terms of building a more reliable and secure Internet. The existing Internet, with several billion users per day, is vulnerable to many different types of misconfiguration, operator error, software outage, and malicious attack. However, it is too early to tell whether our protocols will be adopted in practice.

7.3 Zachary Tatlock

Zachary Tatlock has not received prior NSF funding.

Project personnel

1. Zachary Tatlock; University of Washington; PI
2. Thomas E. Anderson; University of Washington; co-PI
3. Michael D. Ernst; University of Washington; co-PI
4. Suzanne Millstein; University of Washington; Other professional (staff programmer)
5. Calvin Loncaric; University of Washington; Graduate student (research assistant)
6. Chandrakana Nandi; University of Washington; Graduate student (research assistant)
7. Pavel Panchekha; University of Washington; Graduate student (research assistant)
8. Stuart Pernsteiner; University of Washington; Graduate student (research assistant)
9. Konstantin Weitz; University of Washington; Graduate student (research assistant)
10. James R. Wilcox; University of Washington; Graduate student (research assistant)
11. Doug Woos; University of Washington; Graduate student (research assistant)
12. Steve Anton; University of Washington; Undergraduate student
13. Alex Sanchez-Stern; University of Washington; Undergraduate student
14. Daryl Zuniga; University of Washington; Undergraduate student
15. Juliet Oh; International High School; High school student

Collaborators

1. Collaborators for Zachary Tatlock; University of Washington; PI

    1. Sonya Alexandrova; University of Washington
    2. Thomas Anderson; University of Washington
    3. Maya Cakmak; University of Washington
    4. Adam Chlipala; Massachusetts Institute of Technology
    5. Michael D. Ernst; University of Washington
    6. Dan Grossman; University of Washington
    7. Jon Jacky; University of Washington
    8. Dongseok Jang; University of California, San Diego
    9. David Lazar; Massachusetts Institute of Technology
    10. Sorin Lerner; University of California, San Diego
    11. Calvin Loncaric; University of Washington
    12. Eric Mullen; University of Washington
    13. Pavel Panchekha; University of Washington
    14. Stuart Pernsteiner; University of Washington
    15. Daniel Ricketts; University of California, San Diego
    16. Valentin Robert; University of California, San Diego
    17. Alex Sanchez-Stern; University of Washington
    18. Michael Stepp; University of California, San Diego
    19. Ross Tate; Cornell University
    20. Emina Torlak; University of Washington
    21. Xi Wang; University of Washington
    22. James R. Wilcox; University of Washington
    23. Doug Woos; University of Washington
    24. Nickolai Zeldovich; Massachusetts Institute of Technology

2. Collaborators for Thomas E. Anderson; University of Washington; co-PI

    1. Ivan Beschastnikh; University of British Columbia
    2. Ken Birman; Cornell
    3. Matthew Caesar; University of Illinois Urbana-Champaign
    4. Justin Cappos; NYU-Poly
    5. Dave Choffnes; Northeastern
    6. Doug Comer; Purdue
    7. Chase Cotton; Delaware
    8. Mike Dahlin; Google
    9. Colin Dixon; Brocade Networks
    10. Lucas Dixon; Google
    11. Mike Ernst; University of Washington
    12. Nick Feamster; Georgia Tech
    13. Mike Freedman; Princeton
    14. Andreas Haeberlen; University of Pennsylvania
    15. Daniel Halperin; University of Washington
    16. Zack Ives; University of Pennsylvania

17. Ethan Katz-Bassett; University of Southern California
18. Arvind Krishnamurthy; University of Washington
19. Taesoo Kim; Georgia Tech
20. William Lehr; MIT
21. Boon Thau Loo; University of Pennsylvania
22. Harsha Madhyastha; University of Michigan
23. David Mazieres; Stanford
24. Antonio Nicolosi; Stevens Institute of Technology
25. Simon Peter; University of Texas-Austin
26. Dan Ports; University of Washington
27. Sylvia Ratnasamy; University of California at Berkeley
28. Timothy Roscoe; ETH Zurich
29. Jonathan Smith; University of Pennsylvania
30. Ion Stoica; University of California Berkeley
31. Zach Tatlock; University of Washington
32. Robbert van Renesse; Cornell
33. Michael Walfish; NYU
34. Xi Wang; University of Washington
35. Hakim Weatherspoon; Cornell
36. David Wetherall; Google
37. Christopher Yoo; University of Pennsylvania

3. Collaborators for Michael D. Ernst; University of Washington; co-PI

1. Jenny Abrahamson; Microsoft
2. Ruth Anderson; University of Washington
3. Thomas E. Anderson; University of Washington
4. Shay Artzi; Zappix
5. Vinay Augustine; ABB
6. Richard Bailey; University of Washington
7. Magdalena Balazinska; University of Washington
8. Paulo Barros; Federal University of Pernambuco
9. Ivan Beschastnikh; University of British Columbia
10. Ravi Bhoraskar; Facebook
11. Yuriy Brun; University of Massachusetts
12. Yingyi Bu; Couchbase
13. Brian Burg; Apple
14. Juan Caballero; IMDEA Software Institute
15. Aaron Cammarata; VoidALPHA
16. John Cheng; Veracient LLC
17. Raymond Cheng; University of Washington
18. Seth Cooper; Northeastern University
19. Forrest Coward; Microsoft
20. Marcelo d'Amorim; Federal University of Pernambuco
21. Drew Dean; SRI International
22. Werner Dietl; University of Waterloo

23. Stephanie Dietzel; Tableau Software
24. Kellen Donohue; Google
25. Leonard Eusebi; Charles River Analytics
26. Gordon Fraser; University of Sheffield
27. Vijay Ganesh; University of Waterloo
28. Colin S. Gordon; Drexel University
29. Dan Grossman; University of Washington
30. Sean Guarino; Charles River Analytics
31. Philip J. Guo; University of Rochester
32. Seungyeop Han; University of Washington
33. Irfan Ul Haq; IMDEA Software Institute
34. Pingyang He; University of Washington
35. Reid Holmes; University of British Columbia
36. Pieter Hooimeijer; University of Virginia
37. Bill Howe; University of Washington
38. Wei Huang; Google
39. Laura Inozemtseva; University of Waterloo
40. Jon Jacky; University of Washington
41. Darioush Jalali; University of Washington
42. Ralph E. Johnson; University of Illinois
43. René Just; University of Massachusetts
44. Andrew Keplinger; Left Brain Games
45. Adam Kieżun; Broad Institute
46. Gene Kim; University of Washington
47. Andrew J. Ko; University of Washington
48. Karl Koscher; University of California at San Diego
49. Arvind Krishnamurthy; University of Washington
50. Wing Lam; University of Illinois
51. Dominic Langenegger; ETH Zurich
52. Jingyue Li; DNV Research & Innovation
53. Nuo Li; ABB
54. Calvin Loncaric; University of Washington
55. Alberto Lovato; Università di Verona
56. Hao Lü; Google
57. Damiano Macedonio; Julia Srl
58. Thomas Maddern; Veracient LLC
59. Kelly McLaughlin; XPD Analytics
60. Massimo Merro; Università degli Studi di Verona
61. Ana Milanova; Rensselaer Polytechnic Institute
62. Suzanne Millstein; University of Washington
63. Nathaniel Mote; University of Washington
64. John Murray; SRI International
65. Kıvanç Muşlu; Microsoft
66. David Notkin; deceased
67. Robert Ordóñez; Southern Adventist University
68. Johan Östlund; Uppsala University

69. Pavel Panchekha; University of Washington
70. Timothy Pavlik; University of Washington
71. Jeff H. Perkins; MIT
72. Stuart Pernsteiner; University of Washington
73. Paul Pham; The Evergreen State College
74. Amarin Phaosawasdi; University of Illinois
75. Zoran Popović; University of Washington
76. Alex Potanin; Victoria University of Wellington
77. Brian Robinson; ABB
78. Franziska Roesner; University of Washington
79. Todd W. Schiller; Bridgewater Associates
80. Sigurd Schneider; Saarland University
81. Will Scott; University of Washington
82. Michael Sloan; University of Washington
83. Ciprian Spiridon; Julia Srl
84. Eric Spishak; Google
85. Fausto Spoto; Università di Verona
86. Siwakorn Srisakaokul; University of Illinois
87. Luke Swart; University of Washington
88. Zachary Tatlock; University of Washington
89. Javier Thaine; University of Washington
90. Emina Torlak; University of Washington
91. Ben Treibelhorn; Seattle University
92. Mohsen Vakilian; Google
93. Paul Vines; University of Washington
94. Brian Walker; University of Washington
95. Xi Wang; University of Washington
96. Ronald Watro; Raytheon BBN
97. Konstantin Weitz; University of Washington
98. James R. Wilcox; University of Washington
99. Steven A. Wolfman; University of British Columbia
100. Doug Woos; University of Washington
101. Edward X. Wu; Extrahop
102. Jochen Wuttke; Google
103. Cheng Zhang; Shanghai Jiao Tong University
104. Sai Zhang; Google
105. Yoav Zibin; Google

Collaboration plan

The PIs are Zachary Tatlock, Michael Ernst, and Tom Anderson. We have an established, productive history of collaboration and co-authored the PLDI 2015 paper [WWP+15a] that inspired this grant proposal. We all work in the same department, with offices within a few feet of each other. We co-advise students and work together on multiple joint projects across a number of distinct areas of computer science, including verification, software engineering, and distributed systems. We participate in multiple joint weekly meetings, both on the Verdi project and on other current research topics. In particular, Tatlock and Ernst co-advise students Pavel Panchekha and Konstantin Weitz. Anderson and Ernst co-advise student Doug Woos, with Tatlock as an informal advisor. Tatlock is an informal advisor to Ernst's student Calvin Loncaric, and Ernst is an informal advisor to Tatlock's students Stuart Pernsteiner and James Wilcox. All of these students are interested in formal verification of distributed systems and will work on the proposed project. Ernst and Anderson also co-advised Ivan Beschastnikh (who graduated in 2013 and is currently a professor at UBC), whose work spans distributed systems and software engineering.

The PIs will jointly define and participate in the graduate seminars on systems, programming languages, and software engineering, which are held each quarter. In Spring 2015, Zach Tatlock, along with our colleague Xi Wang, will develop and teach a graduate seminar, CSE 599, on systems verification focusing on operating systems, distributed systems, and networking. This seminar will be project-based, and students will be encouraged to use Verdi to verify their own systems or work on further developing a part of the Verdi framework.

The Software Engineering and Programming Languages Research Lab is a place where students "hang out," allowing not only in-depth interaction among the team members but also informal interactions and idea-sharing with other students in these areas. The PIs bring different styles, knowledge, and approach to the research table. This allows us to help students develop along multiple dimensions. We anticipate many technical challenges in this ambitious project — but luckily collaboration won't be an obstacle.

7.4 Division of Work

As described in the proposal, we will investigate 10 research questions. The primary faculty leads for each question are listed below, but we stress that everyone on the team is a generalist. Although we each have different areas of expertise, we all intend to pitch in on whatever sub-project is needed to push the goals of the work forward. We strive to foster the same attitude among our graduate students.

RQ1	(Anderson, Tatlock)	**RQ6**	(Ernst, Tatlock)
RQ2	(Anderson, Tatlock)	**RQ7**	(Anderson, Tatlock)
RQ3	(Anderson, Ernst)	**RQ8**	(Ernst, Tatlock)
RQ4	(Ernst, Tatlock)	**RQ9**	(Ernst, Tatlock)
RQ5	(Anderson, Ernst)	**RQ10**	(Anderson, Ernst)

Each of these questions has a multi-year research and development agenda. Instead of trying to complete one area and then the next, our plan is to work on them in parallel as the results from many of these questions can help developing solutions to the rest. We will split the research agenda in each area into a series of six-month challenges that we can integrate into the mainline Verdi framework if successful, and learn from our failures if not. The idea is to repeatedly course-correct, to go from working system to working system.

7.5 Collaboration Mechanisms

We plan to continue the model we have used in the recent past:

- Graduate students will be co-advised in almost all cases. Not only do students get the benefit of multiple perspectives, but this provides an almost daily opportunity for faculty to discuss issues with respect to the project.
- Graduate researchers will typically work on sub-projects in small teams of 2–3; in our view, student collaboration across project topics is as advantageous as faculty collaboration.
- Undergraduate researchers will be paired with a graduate student mentor, overseen by one other faculty member. We have found that undergraduates benefit from more frequent interaction than is often possible with over-committed faculty. Furthermore, graduate students learn a lot from the experience of being an advisor.
- We will have weekly project meetings, including faculty and students, to discuss research progress and plan future work. We believe students should see the process of how decisions are made, rather than making those decisions behind closed doors.
- We will hold weekly PI meetings to track progress against progress goals, diagnose and fix student issues, and strategize industrial adoption.
- We will organize and hold an annual off-site meeting to coordinate with practioners from our industrial partners.

Data management plan

The goal of this project is to enable the creation of formally-proven distributed system implementations. To evaluate the space of design alternatives, we plan to build proofs and extract runnable source code from them; we will also run experiments on the running systems. All source code, along with configuration files and instructions for reproducing published measurements of the system, will be made public, hosted via GitHub and accessible from the project website at the University of Washington.

The principal investigators have a long history of making research software and data publicly available, so that other scientists can reproduce and build on the work. Anderson has led the development of several widely used open source software artifacts over the past few years, most notably BitTyrant [PIA+07] and OneSwarm [IPKA10]. Both have been downloaded more than a hundred thousand times. Most recently they have led an effort to produce an easy to configure public anti-censorship tool, uProxy, to enable client browsers to work together to evade censorship [Upr]. uProxy is open source software, and it has more than ten thousand enrolled beta users for an initial release in the next few months. They have also led the iPlane Internet topology mapping project [MIP+06]. iPlane has collected and published Internet topology maps continuously since 2006. The iPlane system measures millions of paths per day and the results have been used by more than thirty research groups. Ernst recently led the development of the Checker Framework (http://checkerframework.org/), which is downloaded about 300 times per day, and his Daikon (http://plse.cs.washington.edu/daikon/) and Randoop (http://mernst.github.io/randoop/) projects are even more popular. Dozens more software downloads are listed at http://homes.cs.washington.edu/~mernst/software/ and http://homes.cs.washington.edu/~mernst/pubs/.

As NSDI steering committee chair, Anderson was instrumental in encouraging NSDI to create a special award for those papers whose source code and data sets are placed in the public domain. Tatlock is the Artifact Evaluation Committee Co-Char for PLDI 2016. In this role, he will encourage the authors of all accepted papers to submit their implementations to the committee which will attempt to independently replicate the paper's results before the conference is held.

Policies for access and sharing. We will make the source code for our tools and all test configurations available for public download using the BSD or Apache license. To ensure the broadest impact of our work, we will make the source code available for re-use without restriction.

Reproducibility of experimental results. We will publish our results and algorithms using peer-reviewed venues such as PLDI, POPL, ICSE, FSE, OSDI, and NSDI, as well as through technical reports freely available through our department web sites. All published experimental results will be accompanied by public source code, configuration files, and data sets, allowing others to directly reproduce our results.

Plans for archiving data. Data archival is essential for the repeatability of experiments and longitudinal studies. We will ensure archival of the measured data reported in our published work. We expect these data sets to be small enough to be served out of the same GitHub and department repositories for the project source code.

Standards, data format and content. There is no accepted standard for storing and maintaining proofs (other than the source code of the proof in Coq) and experimental data from distributed systems. We will ensure that the data set and source code distribution includes sufficient documentation to allow for the published data to be reconstructed by a third party.

References

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag.

[ADH+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[Ama11] Amazon. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region, April 2011. http://aws.amazon.com/message/65648/.

[BBE+11a] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. Bandsaw: Log-powered test scenario generation for distributed systems. In *SOSP Work In Progress*, Cascais, Portugal, October 24–26, 2011.

[BBE+11b] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. Mining temporal invariants from partially ordered logs. *SIGOPS Operating Systems Review*, 45(3):39–46, December 2011.

[BBEK14] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 468–479, Hyderabad, India, June 4–6, 2014.

[BBS+11] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 267–277, Szeged, Hungary, September 7–9, 2011.

[BFN+06] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 55–66, Charleston, SC, January 2006.

[BHEN11a] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Crystal: Precise and unobtrusive conflict warnings. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 267–277, Szeged, Hungary, September 7–9, 2011.

[BHEN11b] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 168–178, Szeged, Hungary, September 7–9, 2011.

[BHEN13] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, October 2013.

[Blo83] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. Ph.D., MIT, 1983. Also available as MIT LCS Tech. Report 303.

[BMH+12] Yuriy Brun, Kıvanç Muşlu, Reid Holmes, Michael D. Ernst, and David Notkin. Predicting development trajectories to prevent collaboration conflicts. In *The Future of Collaborative Software Development*, Bellevue, WA, USA, February 12, 2012.

[Bro08] Martin Brown. Pakistan hijacks YouTube. http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1/, 2008. Accessed: 2015-02-20.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T Sasaki, and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice Hall, 1986.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, Portland, OR, August 2007.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, December 2013.

[Cor14] etcd: A highly-available key value store for shared configuration and service discovery, 2014. https://github.com/coreos/etcd.

[Cow10] Jim Cowie. China's 18-minute mystery. http://research.dyn.com/2010/11/chinas-18-minute-mystery/, 2010. Accessed: 2015-03-21.

[CS09] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[DDE+11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[DHJ+] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. SOSP '07.

[DLE03] Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 21, 2003.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[FGL12] S.R. Foster, William G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012.

[GAM+07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 285–298, Cambridge, MA, April 2007.

[GEG13] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 73–84, New York, NY, USA, 2013. ACM.

[GL00] Stephen J. Garland and Nancy Lynch. Using I/O automata for developing distributed systems. In *Foundations of Component-based Systems*. Cambridge University Press, 2000.

[GMT09] G. Gonthier, A. Mahboubi, , and E. Tassi. A Small Scale Reflection Extension for the Coq System. Technical Report 645, Microsoft Research - Inria Joint Centre, 2009.

[Gol14] Sharon Goldberg. Why is it taking so long to secure Internet routing? *Queue*, 12(8):20:20–20:33, August 2014.

[GRF13] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 483–494, Seattle, WA, June 2013.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[Hig11] High Scalability. The updated big list of articles on the Amazon outage, May 2011. http://highscalability.com/blog/2011/5/2/the-updated-big-list-of-articles-on-the-amazon-outage.html.

[HLP+13] Seungyeop Han, Vincent Liu, Qifan Pu, Simon Peter, Thomas E. Anderson, Arvind Krishnamurthy, and David Wetherall. Expressive privacy control with pseudonyms. In Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan, editors, *SIGCOMM 2013: Proceedings of the ACM SIGCOMM 2013 Conference*, pages 291–302, Hong Kong, China, August 12-16 2013. ACM.

[HLvR09] Jason J. Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 119–133, York, UK, March 2009.

[HSS+14] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. *ACM Trans. Program. Lang. Syst.*, 36(4), October 2014.

[Int] Internet2 configurations. http://vn.grnoc.iu.edu/Internet2/configs/configs.html.

[IPKA10] Tomas Isdal, Michael Piatek, Arvind Krishnamurthy, and Thomas E. Anderson. Privacy-preserving P2P data sharing with OneSwarm. In Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker, editors, *SIGCOMM 2010: Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 111–122. ACM, August 30 - September 3, 2010.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, February 2012.

[JTL12] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st Usenix Security Symposium*, pages 113–128, Bellevue, WA, August 2012.

[KAB+07] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 179–188, San Diego, CA, June 2007.

[KAJV07] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 243–256, Cambridge, MA, April 2007.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

[Kin13] Kyle Kingsbury. Call me maybe, May 2013. https://aphyr.com/posts/281-call-me-maybe-carly-rae-jepsen-and-the-perils-of-network-partitions.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, July 2002.

[Lam14] Leslie Lamport. Thinking for programmers, April 2014. http://channel9.msdn.com/Events/Build/2014/3-642.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226, Edinburgh, UK, June 2014.

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[Ler09b] Xavier Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, December 2009.

[LFKA] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. SOSP '11.

[LG02] N Lynch and S Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33:51–59, 2002.

[LGW+08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^3S: Debugging deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 423–437, San Francisco, CA, April 2008.

[LHKA11] Vincent Liu, Seungyeop Han, Arvind Krishnamurthy, and Thomas E. Anderson. Tor instead of IP. In Hari Balakrishnan, Dina Katabi, Aditya Akella, and Ion Stoica, editors, *HotNets 2011: Proceedings of the Tenth ACM Workshop on Hot Topics in Networks*, page 14, Cambridge, MA, USA, November 14 - 15, 2011. ACM.

[LHKA13] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E. Anderson. F10: A fault-tolerant engineered network. In Nick Feamster and Jeffrey C. Mogul, editors, *NSDI 2013: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 399–412, Lombard, IL, USA, April 2-5, 2013. USENIX Association.

[LKvR+99] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert L. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–92, Kiawah Island, SC, December 1999.

[LNZ15] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *ESEC/FSE 2015: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Bergamo, Italy, September 2–4, 2015.

[LPB15] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL specification mining. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, Lincoln, NE, USA, November 11–13, 2015.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mad14] Doug Madory. Chinese routing errors redirect Russian traffic. http://research.dyn.com/2014/11/chinese-routing-errors-redirect-russian-traffic/, 2014. Accessed: 2015-02-20.

[MBEN13] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Making offline analyses continuous. In *ESEC/FSE 2013: The 9th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 323–333, St. Petersburg, Russia, August 21–23, 2013.

[MBEN15] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Reducing feedback delay of software development tools via continuous analysis. *IEEE Transactions on Software Engineering*, 2015.

[MBH+12a] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Improving IDE recommendations by considering global implications of existing recommendations. In *ICSE'12 New Ideas and Emerging Results Track*, pages 1349–1352, Zürich, Switzerland, June 6–8, 2012.

[MBH+12b] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative analysis of integrated development environment recommendations. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2012)*, pages 669–682, Tucson, AZ, USA, October 23–25, 2012.

[MBM13] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *ESEC/FSE 2013: The 9th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 631–634, St. Petersburg, Russia, August 21–23, 2013.

[MBM15] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Preventing data errors with continuous testing. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 373–384, Baltimore, MD, USA, July 15–17, 2015.

[MBNC14] Kıvanç Muşlu, Christian Bird, Nachi Nagappan, and Jacek Czerwonka. Transition from centralized to distributed version control systems: A case study on reasons, barriers, and outcomes. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 334–344, Hyderabad, India, June 4–6, 2014.

[MCB14] Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, pages 374–389, Vienna, Austria, July 2014.

[McC10] Dugald McConnell. Chinese company 'hijacked' U.S. web traffic. http://www.cnn.com/2010/US/11/17/websites.chinese.servers/, 2010. Accessed: 2015-03-21.

[MHvF15] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient synthesis of network updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 196–207, New York, NY, USA, 2015. ACM.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MIP+06] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas E. Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: An information plane for distributed services. In Brian N. Bershad and Jeffrey C. Mogul, editors, *OSDI 2006: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 367–380, Seattle, WA, USA, November 2006. USENIX Association.

[MSBE15] Kıvanç Muşlu, Luke Swart, Yuriy Brun, and Michael D. Ernst. Simplifying development history information retrieval via multi-grained views. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, Lincoln, NE, USA, November 11–13, 2015.

[MSW11] Kıvanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), New Ideas Track*, pages 496–499, Szeged, Hungary, September 7–9, 2011.

[Muş13] Kıvanç Muşlu. Integrating systematic exploration, analysis, and maintenance in software development. In *ICSE'13, Proceedings of the 35th International Conference on Software Engineering*, pages 1389–1392, San Francisco, CA, USA, May 22–24, 2013.

[NE02a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE 2002, Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, pages 11–20, Charleston, SC, November 20–22, 2002.

[NEG+04] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Victoria, British Columbia, Canada, September 2008.

[NRZ+14] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at Amazon Web Services, September 2014. http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf.

[NYT11] NYTimes. Amazon's trouble raises cloud computing doubts, April 2011. http://www.nytimes.com/2011/04/23/technology/23cloud.html.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6(4):319–340, December 1976.

[Ong14] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, August 2014.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[PCG+15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015.

[Pea] The peacoq development environment. http://goto.ucsd.edu/peacoq/. Accessed: 2015-09-20.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, pages 223–252, September 1977.

[PIA+07] Michael Piatek, Tomas Isdal, Thomas E. Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in BitTorrent? In Hari Balakrishnan and Peter Druschel, editors, *NSDI 2007: Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, Cambridge, Massachusetts, USA, April 2007. USENIX.

[PJZ+14] Simon Peter, Umar Javed, Qiao Zhang, Doug Woos, Thomas E. Anderson, and Arvind Krishnamurthy. One tunnel is (often) enough. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *SIGCOMM 2014: Proceedings of the ACM SIGCOMM 2014 Conference*, pages 99–110, Chicago, IL, USA, August 17-22, 2014. ACM.

[PKC+12] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: sharing the network in cloud computing. In Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese, editors, *SIGCOMM 2012: Proceedings of the ACM SIGCOMM 2012 Conference*, pages 187–198, Helsinki, Finland, August 13 - 17, 2012. ACM.

[PKRS11] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: sharing the network in cloud computing. In Hari Balakrishnan, Dina Katabi, Aditya Akella, and Ion Stoica, editors, *HotNets 2011: Proceedings of the Tenth ACM Workshop on Hot Topics in Networks*, Cambridge, MA, USA, November 14 - 15, 2011. ACM.

[Raf] The Raft consensus algorithm. http://raft.github.io/. Accessed: 2015-09-20.

[Rah12] Vincent Rahli. Interfacing with proof assistants for domain specific programming using EventML. In *Proceedings of the 10th International Workshop On User Interfaces for Theorem Provers*, Bremen, Germany, July 2012.

[Rid09] Thomas Ridge. Verifying distributed systems: The operational approach. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 429–440, Savannah, GA, January 2009.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Network Working Group, January 2006.

[RRJ+14] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 452–462, Edinburgh, UK, June 2014.

[SE98] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[SHS+12] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *SIGCOMM 2012: Proceedings of the ACM SIGCOMM 2012 Conference*, pages 13–24, August 2012.

[Sla10] Daniel Slane. 2010 report to Congress of the U.S.–China Economic and Security Review Commission, September 2010.

[SNB15a] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 77–87, 2015.

[SNB15b] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 333–358, 2015.

[SPM+14] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 112–122, New York, NY, USA, 2014. ACM.

[SRvR+14] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 395–406, Atlanta, GA, June 2014.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[SS10] Antonis Stampoulis and Zhong Shao. Veriml: Typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 333–344, New York, NY, USA, 2010. ACM.

[SW01] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[TL10] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[TLSS10] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California fault lines: Understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 315–326, New York, NY, USA, 2010. ACM.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Upr] Uproxy.org. http://uproxy.org/.

[VPEJ15] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. Cascade: A universal programmer-assisted type qualifier inference tool. In *ICSE'15, Proceedings of the 37th International Conference on Software Engineering*, Florance, Italy, May 20–22, 2015.

[WLZ+14] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47, Broomfield, CO, October 2014.

[WWP+15a] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015, Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, pages 357–368, Portland, OR, USA, June 15–17, 2015.

[WWP+15b] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and verifying distributed systems. In *PLDI 2015*, June 2015.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 213–228, Boston, MA, April 2009.

[YKKK08] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 229–244, San Francisco, CA, April 2008.

[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, August 2002.

[Zav12] Pamela Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, April 2012.

[ZDK+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100, New York, NY, USA, 2013. ACM.

[ZJW+14] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 385–396, San Jose, CA, USA, July 23–25, 2014.

[ZSS+] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. SOSP '15. ACM.