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-

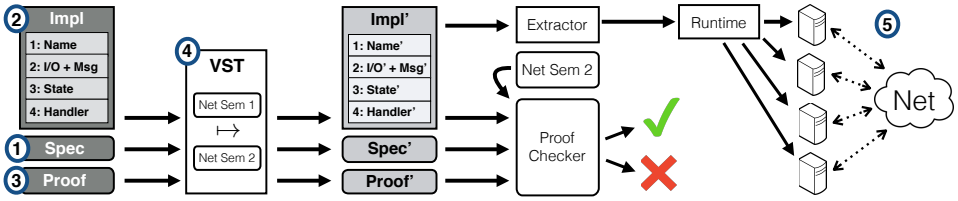

Figure 2: Verified System Transformer Workflow.

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