Building Reliable and Performant Software Defined Networks

Ankush Desai, Wenting Zheng
CS 262A Course Project,
University Of California, Berkeley

Abstract

Distributed systems are notorious for being very difficult to implement and debug. There are no standard approaches or tools for design, implementation and systematic testing of these complex systems. We present a framework based on model based design that unifies design, implementation and testing. We evaluate our framework by building a fault tolerant hash table. Our results show that several non-trivial bugs can be caught by our framework at early stages of implementation and the correct by construction hash-table runs successfully on deployment. We applied the above techniques to a specific problem in software defined networking (SDN). We modeled, verified and generated code for algorithms for solving a specific problem in SDN: availability.

SDN is an instance of a distributed system where controllers are distributed and policies have to be satisfied in the presence of failures. Modern SDN controller designs often do not satisfy the availability requirement, which is an important property for networks to satisfy. In this paper, we present definitions of consistency in SDNs, designs for eventually consistent algorithms that satisfy availability, and analysis of consistency's impact on SDN policies. We present simulation results that show eventually consistent algorithms to be much more available and robust during network failures.

1 Introduction

Fault-tolerant distributed systems are challenging to build, operate, test and debug. A real-world deployment of a distributed protocol consists of many sub-protocols working together to guarantee the correctness of the system. These systems are rarely specified explicitly. Even worse, the specifications keep changing during the implementation phase. Consequently, there is no standard approach for testing distributed systems in the presence of failures. The general approach is using random testing by deploying the protocol on a cluster and injecting random errors in it. But this is not sufficient as it fails to prove the system correct! We need a more systematic approach towards modeling, implementing and testing of these systems.

In this paper, we investigate a new approach based on Model Based Design (MBD) for building fault-tolerant distributed systems. We develop a system compositionally, one component at a time, and continuously test each component against specifications of other components interacting with it. We have used our approach to build a fault-tolerant hash table consisting of various sub-protocols like leader election, multi-paxos [4], and failure detection-recovery etc. Using our framework we were able to verify a small instance of the entire fault tolerant hash table. We considered another case study, Eventual Leader Election protocol from software defined network domain, and modeled-verified the protocol using our framework.

Software Defined Networking (SDN) presents a different approach to networking. Traditionally, networks are managed in a completely decentralized manner. SDN factors out the logical control layer and divides networks into two planes – *control plane* and *data plane*. Control plane consists of the logic that is used to make decisions for the network, and resides in *controllers*. Data plane uses instructions from the control plane and handles packet forwarding. The data plane resides in *switches*.

Modern implementations of SDN [3, 7, 8, 11, 14] have availability issues because most of the controller implementations today applied distributed systems techniques to solve network issues without realizing the weaknesses of such distributed systems. Onix [11], for example, uses state machine replication and consensus protocols to achieve strong consistency among controllers. However, if there is a network partitioning with one or more minority partitions, the consensus protocols will prevent the minority partitions from making progress. One of the questions that we want to answer is: what does consistency really mean in networks?

Contributions: To summarize, the paper makes the following contributions:

- We built and evaluated a model based design framework for distributed systems.
- We built a correct by construction fault tolerant hash table.
- We define consistency in networking, and designed eventually consisten algorithms.
- We analyze the impact on policies with respect to different consistency models.
- We modeled and verified the Eventual Leader Election protocol using our MBD framework.

2 Building Reliable Distributed Systems

Why are distributed systems are hard to design, implement and debug ?

There are various sources of non-determinism in a distributed system like asynchrony, message re-ordering, failures, etc. which makes them harder to reason about.

Asynchrony : Nodes in a distributed system execute asynchronously and communicate by sending message to each other. Guaranteeing safety and liveness in the presence of asynchrony adds one level of complexity to the protocol design.

Failure: what complicates things further in distributed system are failures, in particular, node failure and message loss. Creating fault-tolerant protocols adds another level of complexity to the protocol design.

In the literature a lot of work has been done to propose algorithms that provide desired guarantees, and are accompanied with mathematical proofs. But in practice when a system designer tries to implement these 'proven' protocols, the real implementation deviates from the proposed algorithm. These deviations are mainly because of the custom optimization for performance or adaptation of the protocol in different context. This crucial insight is very well highlighted by the authors of chubby system[4].

> *(1) There are **significant gaps** between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and **the final system will be based on an unproven protocol**. (2) The fault-tolerance computing community **has not developed the tools** to make it easy to implement their algorithms. (3) The fault-tolerance computing community has not paid enough attention to **testing**, a key ingredient for building fault-tolerant systems.*

We will use the observations made by chubby developers as motivation for this paper and explain how we tackle them.

problem 1: Significant gap between the formal proof and the actual implementation.

Solution: We believe that the solution to this problem is to bridge the gap between formal model and the actual implementation. We use model based design approach where the model acts as the implementation.

problem 2: No tools from the community that makes system building easier.

Solution: The aim is to combine following things into a single tool: (1) Design a programming language for modeling complex fault tolerant protocols, (2) build framework for systematic testing and (3) runtime for executing and deploying distributed protocols.

problem 3: Not enough attention paid to rigorous testing of distributed system.

Solution: We use ZING [2] a state of the art explicit state model-checker for performing systematic testing of our implementation.

In the rest of the section we will first explain our model based design framework, followed by brief description of a fault-tolerant hash table based on multi-paxos that we implemented using our framework.

2.1 Model Based Design for Distributed Systems

We designed a framework with the aim of unifying *modeling*, *implementation* and *testing*. Figure 1 gives an overview of the framework.

Modeling: We use the programming language P [5] for implementing distributed protocols. P is a domain specific language for modeling asynchronous systems. Programs in P are collection of state-machines communicating with each other using asynchronous messages. Each state machine has an associated FIFO buffer. Details about the semantics of the language can be found in [5]. P was used in the past for modeling device drivers and didn't have support for complex data-structures. We added complex data-types like maps, sequences and tuples into the language so that complex distributed system implementation can be written in it.

Modeling a complex system in our framework consist of three main parts:

- ***Compositional Protocol Implementation:*** A distributed fault-tolerant protocol consists of many sub-protocols working together to provide the desired guarantees. We implement each of these sub-protocols as separate components in P and test them compositionally to prove properties of the global system. Doing things compositionally helps in scaling the verification as well as provides the capability of replacing components in the future without verifying the entire system again (Details about compositional design is out of scope for this project).

- ***Environmental Modeling:*** Most of the bugs or crashes in complex systems are because of wrong assumptions about other components or environment with which they interact. For building robust systems, we specify the environment behavior in P and close the system. For-example, if our protocol interacts with a database for storing log then it does make certain assumptions about the storage. The programmer should specify the assumptions made about the database. This gives programmer insights about the assumptions made at each interface level and verify the system under these assumption.

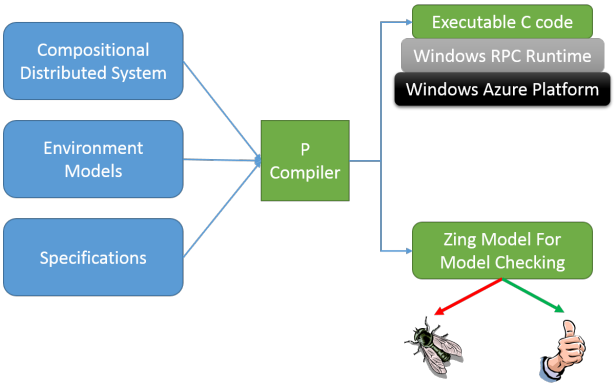

Figure 1: Model Based Design using P

- ***Specifications:*** Specifications for each of the component are written as monitors in P . The framework supports verifying both safety and liveness properties.

P ***Compiler:*** The P compiler takes as input P programs (as shown in Figure 1) and generates executable C code along with models which are verified by ZING . The type checker of P compiler makes sure that the payload value transmitted along with the message is of correct type and can be successfully received at the target machine (in remote node). The compiler also supports serialization and deserialization of these complex data types and thus reduces the load on programmer.

Executable Code and Runtime: The executable C code generated by the P compiler implements the protocol logic specified by the programmer. The generated C code is executed by a distributed runtime. The distrbuted runtime exposes functionalities like creating a remote VM, creating P state machine on a remote VM and sending messages over the network to these state-machines. The network communication is implemented using synchronous Microsoft RPC. We also implemented a deployment process that automatically deploys the distributed protocol on a cluster.

The interface between the generated code and the P runtime is well-defined and hence, can supports any platform dependent runtime as long as it provides the desired interface. This makes it easy to separate the correctness of the protocol which is dependent on the protocol logic and the performance which is mainly dependent on how the program is executed in the distributed setting by the runtime.

2.2 Case Study: Fault Tolerant Hash Table

As a case study for our model based design framework we implemented a fault-tolerant hash table. *Fault-tolerant* means: It should appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail. We built the hash table on top on replicated state machines, and used multi-paxos for implementing consistent log replication.

Figure 2 provides an overview of our hash table implementation. The hash table interface provides two basic operations:

- ***Add(key: int, value: int):*** Add (key, value) in to the hash table. This operation is asynchronous and the hash table sends back a *response:(true, false)* message based on whether the value was successfully committed or not.

- ***Read(key:int):*** Read value corresponding to the key. This operation is asynchronous and the hash table sends back a *readResponse:int* message if the read was successful or no response.

Note that currently the hash table supports only *int* types but extending it to support other complex types is simple.

We use multi-paxos for replicating the state-machine implementing hash-table. As shown in figure 2, multi-paxos uses leader election protocol (which in turn uses basic paxos for electing the leader), and fault detection-recovery protocol.

We implemented a *sequential* multi-paxos with a *distinguished leader*.

- ***Distinguished leader:*** The leader election protocol is such that at any point in time only one node is elected as a leader. All nodes are aware of the current leader, client sends request to any node and the node then forwards it to the correct leader. If the leader fails the client requests are dropped until the failure detection algorithm detects leader failure. The failure recovery unit starts the leader election phase to elect new leader. Hence, only one node can act as a leader at any point in time. *(This is just to simplify things by sacrificing availability).*

- ***Sequential:*** The distinguished leader services requests sequentially, again this is a simplification and can be extended to make it parallel in future. In the current implementation only one operation is active at a time.

- ***Multi-Paxos:*** The multi-paxos protocol is adopted from [4]. Each node in the system acts as *proposer*, *acceptor* and *learner*. Basic Paxos [13] is used to propose values to be committed at each slot position in the log. In out implementation, all the 3 state machines are combined into a single state machine. The learner logic is responsible for executing the replicated state-machine, which in our case is the hash-table logic. All the replicated machines send response back the client and the client has to ignore redundant messages, this can be optimized in future such that only few learners send responses.

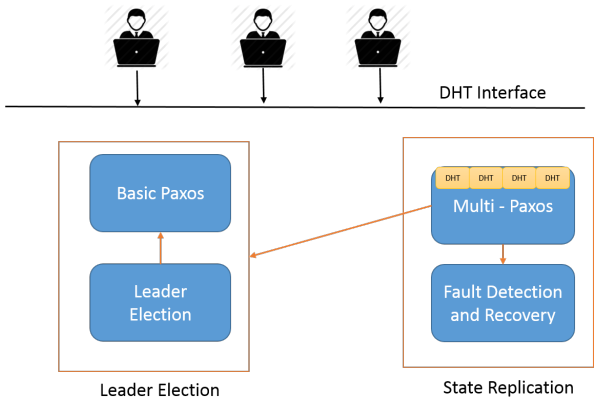

Figure 2: Fault tolerant hash table using Multi-Paxos

During the compositional testing of the system, we verified specification for each of the component separately and using assume-guarantee reasoning [] proved that the global properties hold. Table 1 gives an overview of the properties we checked for the multi-paxos component. We systematically tested our system and also ran it in a distributed setting, evaluation results are described in Section 4.1.

3 Software Defined Networks

There are two different networking "planes." The *data plane* handles the forwarding of packets based on local information, while the *control plane* is responsible for computing and installing that forwarding information in switches. Traditional networks typically combine the implementation of these two planes in routers, so that they both (i) engage in distributed protocols to compute the forwarding state, and (ii) execute data plane actions on arriving packets based on that state.

In SDN, the intelligence of the control plane resides in one or more network "controllers" that run some control software. This control software is responsible for managing the data plane to achieve the desired network behavior, and is often layered into a network operating system (NOS, which handles basic communication with the network switches) and one or more applications written on top of the of this NOS (to determine the network behavior).

3.1 In-band vs. out-of-band

While the SDN paradigm is based on the separation between the control and data planes, it is silent on how connectivity is maintained on the control plane itself. That is, the control plane is responsible for enabling switches to route packets to their intended destination, but SDN does not dictate how the communication between controllers, and between controllers and switches, is maintained. SDN designs have two main approaches control plane connectivity.

In-band Control plane packets are carried over the data plane network.

Out-of-band Control plane packets are carried over a physically separate network. This separate network runs its own routing algorithm.

Out-of-band network is often not viable for a wide-area network, or even in a multi-datacenter setting. The cost of setting up a completely separate network is too high. In addition, the control plane network may have different behaviors from the data plane network, which makes it difficult to predict what will happen if the control plane network acts differently from the data plane network. Moreover, in practice, out-of-band networks are usually not as robust as in-band networks.

3.2 Consistency in networks

In traditional distributed storage systems, the notion of *consistency* is usually clearly defined. For example, distributed databases need to be fault-tolerant. In order to achieve fault tolerance, the databases may decide to perform replication. Replication designs often use consensus protocols to achieve agreement. Replication guarantees that the state stored on multiple machines are *consistent* with each other. That is, each machine can be viewed as a *state machine*, and that these state machines will go through the *same* series of actions and reach the same final state. Consensus protocol is used to agree on a log of actions. Provided that the state machines all begin in the same initial state, the machines will eventually all reach the same state since they have agreed on the same sequence of actions. This type of consistency works well for databases' ACID requirements. However, there is some trade-off here in terms of availability. The famous CAP theorem states that there is a trade-off among consistency, availability, and partition tolerance. A distributed system cannot achieve all three. If a system is consistent and partition tolerant, it cannot be available. In the case of distributed database, this is okay. If the database is storing important data (e.g. bank account), it is important to provide strong data consistency guarantees.

SDN is bringing network design much closer to distributed systems design. However, there are some network-specific requirements that must be taken into consideration when applying distributed systems concepts. Networks are inherently very different from highly consistent systems such as distributed databases. However, controller designs have been applying distributed systems concepts in a strange way. For example, Onix [11] use state machine replication to achieve fault tolerance. This was in response to centralized controller designs [7]. Clearly, a single source of failure is not robust to failures. Paxos was used to achieve fault tolerance by keeping a strongly consistent copy of the

Property	LTL	Description
Safety 1 (Validity)	$\forall pV \in proposedValue, \exists cRV \in ClientReqValue, pV = cRV$. and $\forall cV \in ChosenValue, \exists vP \in proposedValue, cV = pV$.	This property checks the validity of the proposed and chosen value. It asserts that all the proposed values are values requested by the client and all the chosen values are values proposed by the proposers.
Safety 2 (Correctness)	$If\,(s,n,v) \in SeqValues$ then $\forall (s,n',v') \in ProposedValue, if\, n' > n, then\, v' = v$. $\forall (s,n',v') \in ChosenValue, if\, n' > n, then\, v' = v$.	This is the basic paxos correctness property. If a value is being accepted for a slot s then all future values proposed for that slot should be the accepted value. And same is true to chosen value.
Safety 3 (Consistency)	$If\,(s,n',v') \in learntValue_{n1}$, $(s,n'',v'') \in learntValue_{n2}$. then $n = n'$, $v = v'$.	This property asserts consistency of values learnt. If a value is learnt by a learner at node $n1$ for a slot s then value learnt for slot s at any other node $n2$ should be the same.
Liveness Property	$If\, leader \notin FailedNodes$ then $valueProposed_{leader} \Rightarrow$ **Eventually**$(valueProposed_{leader} \in learntValue)$.	If the distinguished leader does not fail, then a value proposed by the leader is always eventually learnt by all the nodes.

Table 1: Properties Verified for Multi-Paxos Protocol

network state. However, networks are different from distributed databases. Replicating a consistent logical network state does not say anything about the state of the *physical network*. The underlying network may have changes, but Paxos only guarantees consistency among the controllers. Controllers being consistent with each other will not help routing in any way. What does consistency really mean in networks? In terms of routing, consistency really means being *consistent with the physical network*.

Strong consistency guarantees under this new definition are not possible. For example, it is impossible to have the controllers be immediately consistent with the physical network. Things get even more complicated under an asynchrounous network. While data consistency could be achieved by using Paxos, it is entirely possible for a link to quickly fail and recover without the rest of the system detecting it. How could we possibly achieve consistency if we want to be consistent with a physical network?

The good news is that routing does not depend on past history. If host A wishes to route a packet to host B, it does not matter to A or B how the network behaved in the past. All they care about is what the state of the network is at this moment. Thus, strong consistency with the physical network is not required.

We formalize the discussion by defining several levels of consistency:

Semantic consistency Controllers' views have to be consistent with each other *and* with the physical network in bounded time.

Controller consistency Replicated state machine type of consistency among the controllers. The controllers all have consistent views. Can be achieved by Paxos.

Eventual partition consistency Controllers should be *eventually* consistent with each other and with the physical network within a partition.

Semantic consistency cannot be achieved under network partition. A controller in one partition cannot know *anything* about the state of another partition. Controller consistency can be achieved at the cost of being unavailable during network partition. In terms of network routing, what we want is *eventual partition consistency*. This means that controllers in different partitions will get out of sync with each other, but we choose availability above controller consistency.

3.3 Eventual Leader Election

This section presents an overview of an algorithm for achieving eventual partition consistency among controllers.

We assume an in-band network where controllers and switches reside on the same network. There are two ways messages can be propagated. One way is flooding packets; another way is propagating using routing tables. Packets are divided into two types: *control packets* and *data packets*. Control packets are flooded throughout the network, while data packets are routed using routing tables. If a packet cannot be sent via a link (i.e. the link through which the packet should be routed is actually down), then the packet will be dropped.

There is a set of controllers and a set of switches.

Controllers Each controller keeps a local *connectivity graph*, which is a representation of what it believes the network looks like at a certain moment in time. In order to retrieve this information, each controller periodically sends a *GetSwitchInformation* control packet to all switches. It will receive link state information from the switches.

We designate one controller to be the *leader*. The leader is in charge of computing rules and pushing those rules to all switches. Each controller computes the leader from its connectivity graph. For simplicity, we designate the controller with the lowest ID to be the leader for a partition. The leader will send routing rule updates to the switches if it detects a change in the network.

Switches Switches route data packets based on its local routing table. When a controller tells a switch to change its routing rules, it will do so immediately. If a switch receives a *GetSwitchInformation* control packet, it will flood its link states in *SwitchInformation* control packets to controllers.

The above is a very simple algorithm design for achieving eventual consistency among the controllers. Note that it is possible that at one time, multiple controllers think they are the leader. This is okay, because the controllers within the same partition will eventually converge to the same network connectivity graph. The algorithm also fulfills the requirement of being available during network partition.

3.3.1 Discussion

There are many algorithms that satisfy our goals, but there are three properties that these algorithms must have:

Periodic refresh The controllers *must* have a mechanism for periodically retrieving the state of the network. In a real network, packets can be dropped arbitrarily over a link, due to network partition, etc. A controller can never be entirely sure that it currently has the correct state of the network. Therefore, each controller must have some way of refreshing its (possibly stale) local copy of the physical network state. To motivate this further, consider the example in figure 3. C1 and C2 are controllers, S1 through S4 are switches, and H1 and H2 are hosts. For simplicity, let us assume that C1 is a leader, and uses master-backup replication scheme to replicate to C2. The switches attempt to notify the current leader when they detect changes. Imagine the following situation:

- Link between S3 and S4 goes down. H1 is now partitioned off from H2.

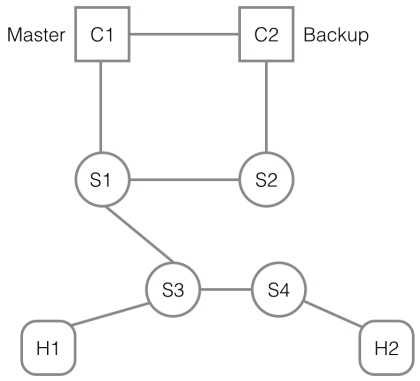

Figure 3: Example of a sequence of failures that will cause the controllers to not converge to the eventual correct state of the network.

- S3 notifies C1 that link S3-S4 is down. C1 notifies C2 of the event. C1 calculates routing rules and sends updates.
- Link S3-S4 comes back up. C1 is notified of the event.
- C1 fails before it has a chance to notify C2 of the new event. C2 realizes C1 is down and takes over as master.

C2 is now out-of-date, unless it specifically queries the state of the network again. C2 will still think that H1 cannot reach H2, even though the link is recovered on the physical network. Either the switch has to *push* information to the controllers periodically, or the controllers have to *pull* information. Since SDN designs have smart controllers, we use the pull mechanism in our design.

Deterministic function In the eventual leader election protocol, each controller eventually converges to a local connectivity graph. The graphs' consistency model is eventually partition consistent, which means that eventually, all of the controllers that belong to a single partition will converge to the same graph. However, to ensure that the controllers reach the same conclusions based on the graphs, we would need them to execute deterministic functions on the local state. For example, in eventual leader election, each controller has to decide whether it is the current leader using local information. If each controller has a different ID and can be ordered, we can guarantee that controllers can choose a single leader within each partition using the eventually consistent local connectivity graph.

Order-independent events Consider the mechanisms that are used to collect information regarding the network state. If the collected information is order-independent, then the order in which we receive the

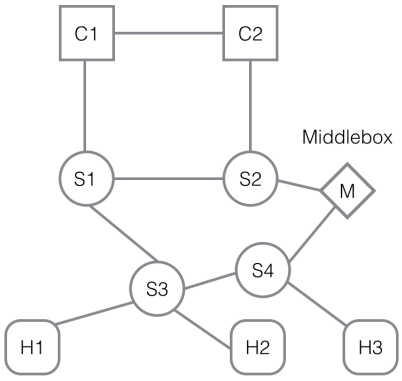

Figure 4: Example layout for middlebox traveral. M represents the middlebox acting as a firewall for all traffic from H1.

events does not impact the state of the final graph construction. This means that the network can reorder the events and the controllers will still be able to converge to the same graph.

3.4 Policy-driven design

In the previous sections, we talked about consistency purely in terms of routing. This is arguably the most basic functionality that networks should support, and the discussion vastly simplified the complexity of real networks. In this section, we present some more complex policies that networks may wish to provide to users, and how they relate to the problems and solutions we talked about in the previous sections.

There are many policies that are implementable by a network. In this section, we will talk about two different policies. For the following discussion, we assume that policies are not changed, while the physical network changes with links failing and recovering.

Middlebox Traversal Middleboxes are data plane elements that perform packet processing services. A very common packet processing service is a firewall that is able to filter out certain traffic. Middlebox traversal policies, therefore, specify that packets originating from a certain source or destined for a certain destination must pass through a middlebox. An example is shown in figure 4. A possible policy in this case would be that all traffic originating from H1 must pass through M before reaching the intended destination.

Traffic Engineering Traffic engineering is a performance-oriented policy. For example, B4 [8] employs traffic engineering techniques to share bandwidth among applications, possibly using different paths for load balancing. A simple version of traffic engineering is using max-main fairness to allocate resources to *flows*. A flow may consist of a source, a destination, and a bandwidth requirement. Depending on the bandwidth requirements and network topology, a single flow may be divided into parallel links for load balancing purposes.

The two policies described above have different requirements in terms of how well they can be achieved. Middlebox traversal can be achieved without global information. Again, consider figure 4, where we wish to enforce the policy that no packet from H1 can be delivered to a destination without passing through middlebox M first. This policy can be enforced without *any* knowledge of the current physical network. The simplest way to implement this policy is to mark the packet. The middlebox M will process the packet and mark it after the processing is done. The destination will only accept the packet if the packet is marked. In the worst case, the packet will be dropped, but the middlebox traversal policy will never be violated. We defined this type of policy to be *local policy*, as the enforcement is performed without up-to-date knowledge of the network state.

For traffic engineering, things are much trickier. Imagine two flows that do not share links. When failures occur, the flows will often have to be allocated to new paths based on the controllers' graphs of the physical network state. We call this type of policy *global policy*, since it needs accurate and timely global information of the network in order to be achieved. Because controllers' states cannot instantaneously reflect the physical state of the network, the traffic engineering requirements cannot be satisfied at all times if the network connectivity changes.

As we can see, local policies are satisfiable no matter what kind of consistency model we pick. Global policies cannot be satisfied because they depend on the physical network state. One of the future works we would like to do is explore the impact of different consistency models on global policies. In the load balancing policy, one can imagine a situation where keeping a consistent state actually works better than an eventually consistent model. The system's behavior is rather unpredictable during convergence, and eventually consistent model might actually overload a link for a short period of time. We would like to experimentally simulate global policies using different consistency models.

4 Evaluation

The evaluation for our paper was mainly directed towards the following goals:

- Implement and Verify a real-world distributed application using our MBD framework.
- Empirically evaluate the performance of distributed system implemented using our MBD framework.

Models	No. Of Nodes	Bugs Found	Verified
Basic Paxos	5	12	Yes
Multi-Paxos	5	20	Yes
ELE in SDN	6	6	Yes

Table 2: Bugs found during Implementation

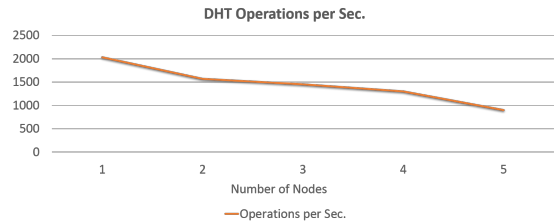

Figure 5: Performance of the Fault Tolerant Hash Table

- Test the performance of SDN algorithms on routing via simluation.
- Model and Verify the Eventual Leader Election protocol.

4.1 Evaluation of Model Based Design

As described in section 2.2, we implemented a distributed hash table using our MBD framework. Using our systematic testing approach we were able to find many subtle bugs in our implementation and fix them. Table 2 shows the number of non-trivial bugs caught and fixed. Note that these bugs where hard to find, for-example: one of the bug was related to failure handling in multi-paxos and occurs only in the presence of n-1 failures. Reproducing this bug using random simulation approach would be very hard but we found this bug within seconds.

Distributed Hash Table. After verifying the hash table implementation for a small instance of 5 nodes, we deployed it on a cluster of 2 physical machines. We deployed the generated code along with runtime on 2 local Windows Server machines in our lab. Figure 5 shows the performance of the hash table when executed with varying number of nodes in the system. You could consider each node as a VM running on physical machine, hence in the case of 5 nodes there were 5 VMs running on 2 physical machines.

We constantly pumped *add* operations into the hash table and measured its performance by varying the number of nodes in the system. The add operation was performed synchronously, in other words, after sending an add request the client waits for a response from the hash table before sending next request. The hash table implementation is based on multi-paxos and hence as the number of nodes in the system increases, the consensus phase takes longer to converge. This can be observed from the graph in figure 5. The take away from this set of results is that the correct by construction hash table works! and comparing the performance with open source implementations is left as future work.

Eventual Leader Election Protocol in SDN. As explained in the previous section, there are two types of architectures in SDNs :

1. **Event Driven** : Switches in the network are event-driven and updates messages are sent to the controller whenever a link-up or link-down event occurs.

2. **Periodic Refresh** : Switches in the network periodically send *current-configuration* messages to the controller or the controller periodically queries all the switches for current link configuration (we implemented the later).

In the context of this paper, the policy that we want the SDN to satisfy is reachability of all the nodes. For reachability it is critical that the controller nodes have consistent view of the connection graph.

Eventual consistency property. In the presence of bounded number of failures, all controllers should have a consistent view of the graph eventually.

If the above property of eventually consistency of controllers holds, then controllers can select an unique leader. Hence, for verifying the eventual leader election protocol it is enough to verify that the system satisfies following property.

$$\textbf{Eventually}(\forall_{i,j}(Controller_i(G) = Controller_j(G)))$$

$Controller_i(G)$ is the view of graph at controller i.

It is quite intuitive that the event-driven approach would fail to satisfy the policy in the presence of a single message-loss. We were able to reproduce a counter example for event driven system. We also implemented the periodic refresh based algorithm and were able to prove that given the network topology, the controllers converge on a unique graph view. We were able to run the generated code from the model on a single server, this can be used for performing simulation of the protocol.

4.2 Evaluation of SDN Algorithm

We used a Python simulator that we built to simluate these algorithms. The simulator is a discrete event simulator, similar to existing network simluators such as ns3. We built this simulator to quickly show the effects of different algorithms under different failure models without the overhead of ns3.

We compared an ideal Paxos implementation, and ideal coordination mechanism, and eventual leader election. Ideal Paxos is implemented with an oracle that simulates the best possible Paxos implemntation and is a lower bound; ideal coordination is a lower bound for any type of coordination protocol. Each simluated network consists of 20 switches, 10 hosts, and 5 controllers.

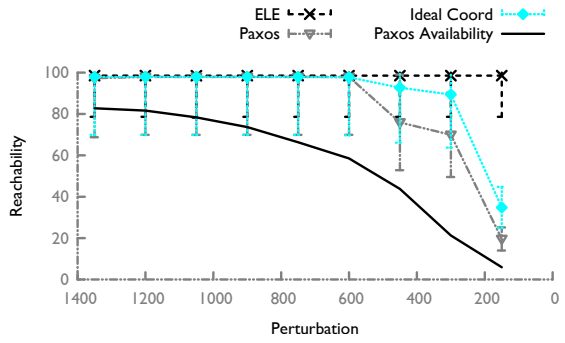

Figure 6: The graph shows three algorithms: eventual leader election, ideal coordination, and ideal Paxos. In addition, Paxos availability calculates how available Paxos is during the test runs. Each point shows the mean for a single simulation run with a particular perturbation parameter, and the error bars show range of values within 1.96 standard deviation of the mean.

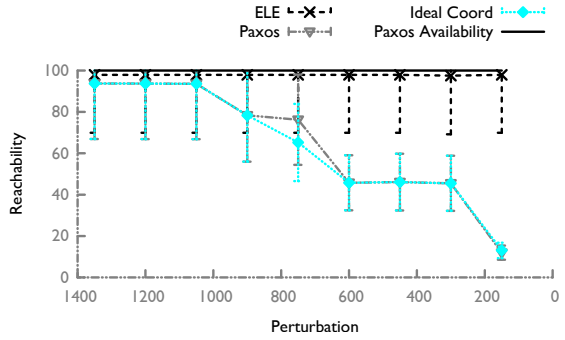

Figure 7: This graph shows what happens when controllers are connected together, but the data plane has a lot of partitions.

The x-axis shows perturbation, where the numbers indicate the mean time to failure (MTTF). Smaller MTTF means that more links are likely to be down at a point in time, and the network is likely to have disconnected components. The y-axis shows the reachability percentage, where 100% means that all *physically reachable* host-host pairs are reachable using this algorithm, and 0% means that the algorithm does not enable any connectivity, even if there are host-host pairs that are reachable using the physical network.

Figure 6 shows the results of simulation on a graph with discrete partitions that are connected together (there are no disconnected components in the network). The controllers and hosts are randomly distributed to the partitions. The links are failed randomly from a set of critical links that connect the partitions together. From the graph we can see that Paxos and ideal coordination do rather well in terms of reachability when there are a few link failures, but perform rather badly when there are a lot of link failures. This makes sense because the network is disconnected, and any coordination algorithm would require that most of the controllers be connected with each other. The Paxos availability line on the graph also shows a drop as the network becomes more unstable.

Figure 7 shows an extreme case where the controllers are always reachable from each other, but not necessarily reachable to the switches. The network is constructed such that the 5 controllers to each other via a single switch, and those links do not fail. Thus, figure 7's Paxos availability results show that Paxos is always available. However, since the rest of the network could be partitioned away, controllers being consistent with each other doesn't mean controllers are consistent with the physical network. The reachability graphs are worse because the controllers are not as spread out.

5 Related Work

The related work for this paper can be split into following categories (1) Programming languages and Verification approaches in the context of distributed systems (2) Software Defined networks and Reachability in SDNs (3) Formal approaches in the context of SDNs.

Programming languages and Testing: Recently, Amazon engineers [1] described their experience using proof engine like TLA+ [12] during the design phase of building distributed web services. But they face the same problem of gap between the design and implementation phase. The closest approach that uses domain specific language with model checking backend is MACE[10], but it is specialized for overlay systems and also they dont use compositional approaches for testing. We provide language features for modeling the environment more precisely. The Ensemble project [15] used layered approach for building communication systems and demonstrated how dividing components into layers helps in building complex systems.

There are testing frameworks like MODIST[16], MACEMC[9] that perform random testing of distributed systems automatically. But as described before, random testing fails to provide coverage guarantees. We use ZING with various search prioritization techniques for finding bugs faster.

Software Defined Networks:

Software defined networks have become very popular in recent years, and arose out of many different research projects research [3, 7, 8, 11, 14]. One example of an early SDN controller system is Nox [7]. Nox consists of a centralized server component that manages the rest of the network. This design is not fault tolerant because the data plane will not be able to react to link failures and recoveries if the centrailzed controller goes down. Onix [11] was a response to Nox's fault tolerance problems by using a replicated state machine mechanism to synchronize the network state, or the Network Informa-

tion Base (NIB). This is a good way to increase fault tolerance of the system, but does not provide availability when the network is partitioned. Google's B4 [8] is an implementation of SDN on a large scale. B4 uses Paxos for leader election – elect a primary instance within a single site.

Formal approaches for SDNs: There has been a lot of work recently towards using programming languages and formal methods for creating robust SDNs. Frenetics(from the cornell)[6] is one such network programming language.

6 Acknowledgments

We would like acknowledge the contributions of Ethan Jackson, Shaz Qadeer and Amar Phanishayee (MSR, Redmond) in building P and Aurojit Panda, Ali Ghodsi and Scott Shenker in solving the SDN availability problem.

References

[1] Amazon. Use of formal methods at amazon web services.

[2] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*. 2004.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM Computer Communication Review*, 37(4):1–12, 2007.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07.

[5] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of PLDI*, 2013.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *SIGPLAN Not.*

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14. ACM, 2013.

[9] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, 2007.

[10] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[11] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[12] L. Lamport. http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html.

[13] L. Lamport. Paxos made simple. http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[15] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*

[16] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09.