Modelling of OpenFlow Controllers with Alloy:
A Load-Balancing Protocol

Spring '13 Semester Project for CS512: Formal Methods

Charalampos Mavroforakis     Dimitris Papadopoulos

1 Introduction

OpenFlow is a communications protocol that manages the forwarding plane of a network consisting of routers or switches by programming their flow-tables. It consists of two discrete types of entities, switches and controllers, with controllers being highly customized nodes that are used to enforce policies and implement services and protocols throughout the network. These entities operate in distinct planes, the data plane and the control plane respectively. Messages travelling through the network are forwarded among switches primarily according to the maintained flow-tables. These flow-tables are maintained and modified according to update messages received by controllers. Since controllers are customizable, this framework provides a flexible model that allows network administrators to implement a wide-range of network primitives such as firewalls, QoS, NAT, statistics collection etc. On the other hand, it allows researchers to analyse and evaluate experimental protocols and services without affecting the rest of the network traffic since it “effectively” separates the network (based for example on various protocols and applications being executed) without affecting its topology. This least property is what has made OpenFlow so popular, in particular in the Software Defined Networks community. For the rest of the report, we assume a level of familiarity with basic OpenFlow concepts. We defer interested readers to [3] for a more detailed introduction to OpenFlow.

Here we are interested in modelling a load-balancing protocol implemented with OpenFlow, presented in [4], which constitutes a modification and improvement over protocols introduced in [2] and [1]. Nowadays most online services, offered by web entities such as search engines, Web sites and social networks, are hosted at multiple servers. Within a single data center or enterprise, a front-end load balancer directs incoming client request to a particular server. One of the currently most common approaches to load-balancing is a dedicated hash-based load-balancer, which however has the disadvantage of being an additional piece of hardware running in the network and is generally non-customizable. In [4] the authors propose an OpenFlow based approach to load-balancing that does not suffer from these disadvantages. In its most basic form, the network consists of an entry point switch that communicates with a single controller that is in charge of producing appropriate
forwarding rules for the switch’s flow tables in a way so that it matches a target distribution in the replica servers’. At any given time, each replica server is associated with a percentage of the incoming queries that it must handle and the goal of the controller is to match this percentage at all times with its “knowledge” of the computational capabilities of each server.

In this report, we construct an Alloy module that models the behaviour of this load-balancing protocol. In section 2 we present a detailed description of the protocol in [4]. Following that, in 3 we present an analysis of the properties guaranteed and the assumptions made (implicitly or explicitly) by the authors and we discuss a number of abstractions and implementation choices we made when making an Alloy module to model their protocol’s behaviour. Finally, 4 contains an in-depth presentation of our Alloy code, the way we tested the proposed properties, and a discussion of more complicated issues arising with the implementation.

2 An OpenFlow based load-balancing protocol

Conceptually, load-balancing is a desired property for a service network. Having multiple copies of a resource allows incoming queries to be partitioned and handled by multiple servers, thus splitting the overall processing time for a given amount of requests. Moreover, dynamic load-balancing is very useful to avoid having unique points-of-failure in a network, backing up of proprietary datasets and allowing flexibility in server downtimes (e.g. for maintenance). OpenFlow treats load-balancing as a network primitive and there are many considerations when building such a model. Here we present only a few of them as discussed in [4], [2], [1].

Pro-Active vs. Reactive Load-balancing decisions can be made pro-actively or reactively upon seeing a request. For example, some of the earliest OpenFlow based approaches where reactive, in the sense that for every fresh flow seen by a switch, a message to the controller is being sent and a new rule is added to all tables regarding this new flow. On the other hand, the model we are studying is pro-active. The domain of all possible flows is partitioned beforehand based on the clients IP address and for each range of addresses a rule that maps it to a certain replica is added to the flow-tables initially.

Distributed vs. Centralized If load-balancing switches can handle load-balancing themselves, then this yields a very low-cost and highly scalable load-balancing protocol. However, for an OpenFlow based protocol, switches are limited to forwarding based upon lookups to local flow tables, thus load-balancing decisions cannot be made by them. In the model we are studying here, load-balancing is defined in the control plane. Decisions regarding it are made at the controller and replication can be achieved by having more than one of them.

Static vs. Dynamic Many of the currently used load-balancing systems are oblivious to the current state of the network, such as hash-based load balancing. On the other hand, we are more interested in schemes that react to updated knowledge about the current state of the network and the servers, making dynamic decisions to split the load.
Individual vs. Aggregated As mentioned above, controllers need to have knowledge about the current state of the servers. In the studied model, we assume direct interaction only between controllers and switches and switches and replica-servers. Thus the controllers rely upon messages they receive from the switches in order to update their load-balancing rules. Periodically switches send aggregate statistics regarding their view of the state of the replicas. For example they may report that one replica appears to be offline or that, based on the current rules, a replica appears to be receiving a disproportionally large amount of traffic. In this sense, the protocol here is aggregated since decisions are taken based on some aggregate info for the state of the network.

The model discussed here, at its most basic form, consists of a single point-of-entry switch that handles incoming client requests from a large number of clients and distributes the among a number of replica servers (all of which contain the same content) based on some rules sent to it from a single controller. Initially, rules that partition the domain of clients (based on their IP address) are issued from the controller and afterwards the client periodically sends reports regarding his view of the servers. The controller then updates the set of flow rules based on the message he received, in order to approximately match a target distribution of requests among the replicas. This desired target distribution periodically changes (for example when a server is taken offline for maintenance) and the goal of the controller is to approximate this updated distribution at all times.

2.1 Assumptions of the Model

The model introduced in [4], entails a number of assumptions about the behaviour of the network. Some of them are explicitly stated by the authors whereas others can simply be inferred from the expected behaviour of a well defined load-balancer. Here is a list of these assumptions:

**Uniform Traffic** For the greatest part of the paper, the authors assume that the IP addresses of the clients issuing requests are uniformly distributed among the IP address space. The reason for this is that, using OpenFlow, they must base their routing rules in the prefix bits of the client IP address (e.g. “All requests by clients with IP’s starting with 00 are sent to replica 4”). Hence, to achieve some level of distribution among replicas, they must assume some similar distribution in the source IP addresses. Otherwise, they could simply look at the low-order bits of source IP’s (which are very likely to have high entropy). At the last part of the paper, they discuss how this assumption can be dropped by imposing an artificial distribution and taking routing decisions based on it.

**Congestion at Replicas** Throughout the paper, it is assumed that the most heavily congested part of the network takes place at the replicas. For example message transmission times are supposed to be much smaller than processing at the replicas (which is a realistic assumption for most real-world scenarios).
No Transmission Failures  It is implicitly assumed that all messages are successfully sent and received and not corrupted. For example, in the case of multiple switches, it is never the case that messages from a replica arrive normally at one of the switches but their are lost when sent to another, hence it is never the case that two switches report different views regarding the state of a replica.

3 Desired Properties and Modelling Abstractions

There is a number of desired properties that the authors of [4] guarantee with their protocol. Namely fairness and minimality of rule changes. We hereby explain these properties and discuss how we integrated them in our Alloy model (making some necessary abstractions). We first discuss explicitly stated properties and following that, we identify a set of properties that, while not explicitly addressed in the paper, are nonetheless of utmost importance for the correct operation of the protocol.

3.1 Explicit Properties

Fairness. The load-balancing protocol guarantees that at all times, the load will be split among the replicas under some desired distribution. Some servers might be known to operate faster than others, therefore they can process a larger ratio of the overall traffic than slower ones. This desired distribution changes over time to represent congestion at servers or taking some replica offline. The challenge comes from wanting able to approximately map this distribution at all times. However, in the experimentation part of the paper, the authors only deal with servers being taken offline and not with changes in their performance. We incorporate this property in our model, under the relaxing assumption that all servers are of equivalent processing power and the desired distribution is, at all times, splitting the traffic evenly among the operating ones which is similar to the experiments run by the authors. The reason for this relaxation is the lack of expressiveness of Alloy with respect to numerical values (in particular ratios as is the case here).

Minimizing Rule Changes. The authors are interested in minimizing the number of rule changes following some update in the goal distribution. For example, if a replica goes offline, it is desired that IP ranges that where routed to other replicas are unaffected (i.e., related rules in the flow tables do not change). We integrate this property in our model.

There are other properties that are discussed by the authors that we abstracted away from our model. For example, the authors wish their protocol to achieve a minimal number of rules for each table since this would make table lookups significantly faster. They achieve that, by assuming a fixed number of IP address ranges for the set of rules and then splitting or merging rules for these ranges. In practice, since each rule is associated with a number of prefix bits of the IP address of its source client, a number of three bits will yield 8 ranges. Following that, one can define a binary tree with 8 leafs, each corresponding to a particular three-bit word and write appropriate rules for all of them. Heuristically, if both IP’s starting with 001 and 000 are routed to the \( i \)-th replica, then instead of having two separate rules,
one can write a rule stating “All requests by clients with IP’s starting with 00 are sent to replica $i$” hence lowering the number of overall rules. By applying a heuristic argument (fill replicas with IP ranges, taking “faster” replicas first) they can guarantee that the overall set of rules is minimal. Another property they are interested in is making sure that not packets are lost between transitions. We did not incorporate this in our model, since in our case time is “discretized” and no packets are sent between these time fragments.

3.2 Implicit properties

There is a number of properties that are necessary in order to guarantee the normal execution of the protocol and are implicitly assumed by the authors.

- At all times, all IP ranges of client requests are covered by some replica. It is obvious that otherwise no protocol would be well defined, since some clients would not be served.

- Each IP range is served by exactly one replica at any given time. Otherwise there would be redundancy in the flow tables, with multiple entries some of which would never be used.

- No requests are forwarded to a replica that is currently offline.

4 Our Alloy Model

In this section we describe the model we constructed using Alloy to model the above load-balancing protocol. Our goal is to model correctly all the involved parties and the necessary interactions among them, while verifying (or invalidating) the properties identified above.

Our model consists of network entities captured by the abstract signature Node. These entities can be of three kinds, namely Controller, Switch and Replica. We are modelling a front-end load-balancing service that is being offered at an entry-point of a Web-service provider. The most reasonable model for such a framework consists of a single controller operating at the control plane and one or more switches operating on the data plane. The controller’s goal is to split incoming traffic as evenly as possible among replicas in order to balance their workload. The dynamic component of the model is that these replicas periodically go offline and routing rules need to be recomputed.

In our model, communication is only being done between switches and controller and switches and replicas. That is, the controller is only learning about the status of the replicas through messages he receives from the switches. Incoming traffic in the network comes with client requests, asking for some content stored in the replicas. It should be specified that we consider full content duplication in the replicas, i.e., all replicas store all content. Following the protocol discussed in [4], incoming traffic is split in a number of ranges of IP addresses, each of which is captured by the signature Range. We abstract away the particular way the separation is being done and simply allow for an arbitrary number of ranges to be specified.
Once this number is specified in the scope of a particular execution, it remains static. Finally, we make the assumption (and provide it as a guarantee that can be checked by an assertion in Alloy) that at all times, at least one replica is online. This closely models real-world practices where a downtime of mere minutes can result to losses of millions of dollars, hence enterprises take measures towards being able to provide content at all times.

We follow an event-driven modelling strategy made possible by Alloy’s `util/ordering` module. The events that set our model in motion are messages that are being sent from one of the switches to the controller to report a change in the behaviour of one or more of the replicas and corresponding messages from the controller to (all of) the switches that contain the necessary updates in the routing rules. These messages are captured by the abstract signature `Message` that has an ordering imposed to it. Extending this signature, we have two abstract signatures `SwitchToControlMessage` and `ControlToSwitchMessage` that model these two message types. Finally, the first category of messages is extended to `OnlineMessage` and `OfflineMessage` and the second is extended to `UpdateMessage`. The first two capture that the event taking place is a switch sending a message reporting that a number of replicas has gone online or offline and the last that the event taking place is the transmission of a table update message from the controller towards the switches.

In the following we present a detailed description of our code.

4.1 Signatures of Network Nodes

```alloy
abstract sig Node {}
sig Switch extends Node {}
one sig Controller {
    rules: (Range -> one Replica) -> Tick
}
sig Replica extends Node{}
sig Range {}
```

Most of the signatures defined above do not have fields related to them. Since we are only interested in modelling the controller, we added a field to it that contains the routing mappings for a given time. We are not interested to the internals of switches and replicas and we assume that each range has a pre-defined range of IP addresses tied to it. A very important fact here is that we request each rule to bind a range to a single replica, thus achieving the required property that, at all times, client requests originating from a particular IP range, are routed to exactly one replica. At the last part of our report we will include a case study where we removed this quantifier and present the produced counter-example.

We also want to specify that for our model, objects of the signature `Replica` should be inferred not as the actual replica servers but rather as the view of the replicas that the controller has at a given time. Similarly, the field `offlineReplicas` which is related to a `Tick` object (as will be explained later) refers again to the view of the controller about which servers are offline at any point in time. The reason for this is that, as discussed above, there exists no direct communication between replicas and controller (they essentially duel
in different planes –data plane and control plane respectively) and, since we are interested in modelling the controller’s mode of operation, we do not wish to explicitly include in the model communication between replicas and switches. Consequently, the controller relies on information he receives from the switches in order to infer the latest state of the replicas. As long as no messages are lost or corrupted in the network, as discussed in the assumptions section above, the controller’s view is consistent with the actual state of the network at all times, hence this modification holds no real meaning and certainly does not weaken our model.

4.2 Signatures of Messages and Time Intervals

```plaintext
open util/ordering[Tick]
sig Tick{
    offlineReplicas: set Replica
}
{
    all r: offlineReplicas, rng: Range |
        r not in Controller.rules.this[rng]
}
abstract sig Message {
    pre, post: Tick,
}
{
    post != first
    post = pre.next
}
abstract sig SwitchToControlMessage extends Message{
    r: some Replica,
    sends: one Switch,
    receives: one Controller
}
abstract sig ControlToSwitchMessage extends Message {
    sends: one Controller,
    receives: set Switch
}
sig OfflineMessage extends SwitchToControlMessage {}{
    no repl: r | repl in pre.offlineReplicas
    post.offlineReplicas = pre.offlineReplicas + r
}
sig OnlineMessage extends SwitchToControlMessage {}{
    r in pre.offlineReplicas
    post.offlineReplicas = pre.offlineReplicas - r
```
In the above we made extensive use of the signature fact technique offered by Alloy to model the desired properties for each of these signatures. Observe that objects of the signature `Tick` are ordered. Ticks in our model are variable durations of time that are used to define when a message was sent. To be a little more formal, a *pair* of consecutive ticks defines a duration of time when a message was sent. Each of them has a field that contains an arbitrary set of replicas that are offline during the time period related to that particular tick. Messages are related to two ticks the pre and post time points. In the signatures fact the properties that no event can happen before the beginning of the simulation and that each post must be immediately following the corresponding messages post (no message can span more than one time period).

Objects of the signature `SwitchToControlMessage` are related to a (non-empty) set of replicas and are separated into two categories, those reporting that this set of replicas has gone offline and those reporting that this set of replicas has come back online. Each of these messages is sent by exactly one switch and received by the unique controller. The reason for this is that our strategy for reporting (see below for a more detailed discussion on event simulation) is essentially “report as soon as you see”. Finally, in the appropriate signature facts declarations, we capture the properties that no offline messages can be sent for replicas that are known to be offline –and the same holds for the online case.

We also specify the way in which these messages affect which nodes are offline for that particular time period (addition of the replicas from the `offlineReplicas` field for the case of offline messages and removal for the online ones). Let us reason a little about this implementation choice. When dealing with modelling of sequential procedures, there are generally two approaches for defining a time segment. Either fixed-time or event-driven. Simply put, in the first approach each event happens at a fixed time interval whereas at the second the time segments are of various size. In the case of a network, the first approach would correspond to a protocol where switches ping replicas at fixed times to verify their liveness and report to the server changes in their status at fixed intervals (say, a number of minutes). The second approach (which is the one we take here) corresponds to a protocol where a switch reports an offline replica whenever a transmission (or a series of transmissions) towards it fails. Likewise, each replica sends a “Hello” message whenever it comes back online hence, upon receiving this message, the switch reports this change to the controller. Observe that because of this modelling assumption we made (i.e., to follow an event-driven strategy) we can only have a single type of event (online or offline) reported at any given time. The reason why we chose, nonetheless, to allow for multiple replicas going offline or
online simultaneously is that it is common practice in environments where content replication takes place, to physically locate multiple servers at the same location (indeed at the same rack) thus a change may affect more than one replicas simultaneously (for example a power outage may take offline all servers located in a particular building).

The objects of the UpdateMessage signature and its parent signature, ControlToSwitchMessage constitute a different type of messages. These are related to messages that are sent to switches from the controller, following some change in the forwarding rules that was triggered by a message the former received. Observe that these messages are of the “broadcast” type, i.e., they have all the switches as recipients (a property that is captured at the corresponding signature fact). Also, these messages do not contain a re-computation of all routing rules but instead only contain changes in the routing rules as imposed by the declaration of the corresponding field. The last holds exactly as in real-world OpenFlow protocols in order to minimize the size of sent messages. Finally, no such message can be sent during the first time period, since no change in the replica states has taken place yet.

An important property of our model is defined and captured in the Tick signature. Namely, in the signature facts declaration, we state that no client requests are routed to a replica that is offline.

### 4.3 Event Simulation

This code snippet contains some of the most important techniques used in our model.

```
pred init [t: Tick] {
    no t.offlineReplicas
}

fact traces {
    first.init
    PropertyVerification
    all t: Tick - last | let t’ = t.next |
    some m : Message {
        m.pre = t and m.post = t’
        {t.offlineReplicas = t’.offlineReplicas} =>
        m in UpdateMessage else
        {t.offlineReplicas in t’.offlineReplicas} =>
        m in OfflineMessage else
        {t’.offlineReplicas in t.offlineReplicas} =>
        m in OnlineMessage
    }
}
```

The predicate init contains the only necessary additional setup condition for our model. We make the structural assumption that initially all replicas are online (i.e., the network is at a “healthy” state). The fact traces is the backbone of our simulation. It specifies that an event must take place between each pair of consecutive ticks (in a sense, the fact that
an event took place is what caused the time to progress—recall our discussion above about event-driven simulation) and it furthermore defines what type of event takes place based on the changes on the states of the replicas before and afterwards. Moreover, the predicate PropertyVerification, which will be discussed in detail in 4.5, provides a way to run and validate the main properties of our protocol.

4.4 Model Facts

fact AlwaysResponding {
  all t: Tick | 
  some r: Replica | 
  r not in t.offlineReplicas
}

fact UpdateMessageAfterStateChange {
  all t: Tick - last.prev - last, m: SwitchToControlMessage | 
  let t’ = t.next, t'' = t’.next { 
    {m.pre = t and m.post = t’} => { 
      some m’ : ControlToSwitchMessage { 
        m’.pre = t’ and m’.post=t''
      } 
    } 
  }
}

fact NoUpdateAfterUpdate {
  no m,m’ : UpdateMessage | 
  m.post = m’.pre
}

The above are some required facts that are imposed by the implicit properties discussed in section 3.2 and are generally related with the smooth operation of our protocol. The AlwaysResponding captures that in our model there will always be at least one replica server online. The fact UpdateMessageAfterStateChange captures that every message sent by a switch, that triggers some changes in the routing tables of the controller, must be immediately followed be a broadcast message with the corresponding updates. The next fact is related to the fact that no two update messages can be issued sequentially. To understand the reason for this, one must view it in conjunction with the previous fact and the signature fact in the UpdateMessage signature that disallows update messages at the first tick. These three facts together, guarantee that there is a regular flow of messages throughout the protocol execution, namely that each state change reported by a switch is followed by a corresponding set of rule updates and no rule updates take place as long as no such change is reported.
4.5 Model Verification

pred PropertyVerification {
  all m: Message {
    let t = m.pre, t' = m.post {
      {m in OnlineMessage} => {
        all r: Replica - t'.offlineReplicas{
          {r in Replica - t.offlineReplicas} => {
            Controller.rules.t'.r in Controller.rules.t.r
          }
        }
      } else
      {m in OfflineMessage} => {
        all r: Replica - t'.offlineReplicas{
          {r in Replica - t.offlineReplicas} => {
            Controller.rules.t.r in Controller.rules.t'.r
          }
        }
      } else
      {m in UpdateMessage} => {
        all r: Replica - t'.offlineReplicas{
          Controller.rules.t'.r = Controller.rules.t.r
        }
      }
    }
  }
  all t: Tick, r1, r2: Replica - t.offlineReplicas{
    let rng1 = Controller.rules.t.r1, rng2 = Controller.rules.t.r2 |
    (#rng1 = #rng2)
    // uncomment the part of the code below to
    // allow more flexible distributions than
    // even split among the Replicas
    /* or minus[#rng1, #rng2] = 1 or minus[#rng2, #rng1] = 1 */
  }
}

/*
* When running the example, the number of Ranges needs to be divisible by
* all numbers in [1, #Replicas]. To relax this requirement, uncomment line 246.
*/
run execute for 10 but exactly 4 Replica, exactly 12 Range, 3 Switch

The above predicate captures both of the main properties of our model. The first one is fairness, i.e., that all traffic is evenly distributed among replicas that are online for that time period. We have an additional line (currently commented out) that allows one to relax the required property of exactly even splitting. In real-world implementations this wouldn’t
be an issue, but since here we are modelling a fixed number of IP ranges and a small number of replicas, in order to achieve exactly even split it must that the number of IP ranges is a multiple of the number of replicas (and any number less than that since at any time it may be that an arbitrary subset of them is online). For example, running our example for 4 replicas and 12 ranges would execute correctly without this additional line and it would, at all times, evenly split the ranges among online replicas (since 12 divides 4, 3, 2 and of course 1). Just by relaxing this property by 1, we can accommodate many more combinations of ranges and replicas cardinalities.

The second main property of our model is that the number of rule changes included in each update is minimal. This is captured in the above predicate by allowing changes in the rules only for the ranges that were associated with replicas that went offline, in the case of an offline report, and by forcing replicas that were already online in the previous time segment to maintain as many of their client IP ranges as possible (while at the same time achieving an even distribution) in case some replicas come online.

Running for the execute predicate produces a valid snapshot of our model, that achieves all the desired properties. Keep in mind that, unless the distribution-relaxation part of the code is un-commented, the restriction on the number of ranges discussed above must hold. Two good candidate values for execution are 12 ranges with 4 replicas and 6 ranges with 3 replicas. The numbers of switches, messages and ticks can grow arbitrarily and the model runs very fast (less that 5 seconds) for up to 10 messages. Figures 4.5 and 4.5 include snapshots of our model in full deployment for distinct cases of consecutive stages.

In Figure 4.5, observe how, at the figure on the left representing the initial state, each of the replicas has exactly three ranges associated with it and all of them are online as specified by our modelling assumptions. Following that, in the transition between the first and the second tick, Switch0 reports to the Controller that Replica0 and Replica3 have gone offline. Consequently, the Controller computes the necessary rule changes and assigns ranges accordingly. Observe that Replica2 and Replica1 (the ones that remained online) maintained all of their previously routed ranges, as dictated by our minimum rule changes property. Finally between the second and third tick, the Controller broadcasts to both switches an UpdateMessage with the necessary changes in the rules. Observe that at all times, all ranges were evenly split among the online replicas hence satisfying the fairness property.

On the other hand, Figure 4.5, depicts two consecutive time segments, the first of which corresponds to some intermediate step of the protocol execution where exactly one replica is online thus receiving all of the traffic, having all ranges assigned to it. Following that, on the right, one can see that two of the previously offline replicas come back online and the load is again shared with four of the ranges assigned to each of them. Observe how fairness and minimal rule changes are satisfied again.

We also present in Appendix A a number of assertions we included that can be optionally tested by running corresponding checks. Most of them are self-explanatory so more detailed description of them is omitted. The following figures depict how removing some of the constraints of our model, immediately produces counter-examples that break some of our
Figure 1: Snapshot of three consecutive time segments of our model. Initial tick on the left and last on the right.
Figure 2: Snapshot of two consecutive time segments of our model. Observe how on the left there are three offline replicas and on the right two of them come back online.
assumptions and fail to achieve the desired properties.

In Figure 4.5, we removed the one quantifier from the field declaration of \texttt{rules} in the \texttt{Controller} signature. Immediately this causes problems, since by running a check for any of the assertions \texttt{RangeServedOnce} and \texttt{AllRangesServed} produces counter-examples such as the one seen in Figure 4.5. It is obvious from the figure that there exist IP ranges that are not being served at all (such as \textit{Range1}) and others that are being mapped to more than one replicas (such as \textit{Range3}) which leads to redundant flow rules that are not being used.

Figure 4.5 depicts a counter-example that breaks the fairness property of the protocol and which was produced by removing the important predicate \texttt{PropertyVerification}. An immediate by-product is that the main goal of balancing the load among online replicas is no longer achieved, since there exists an online replica that is not serving client requests (\texttt{Replica0}) and furthermore ranges are not equally split between the other two.

References


Figure 4: A figure depicting how removing the PropertyVerification predicate produces a counter-example that violates the fairness property of our model. Clearly, the distribution of ranges is far from uniform for this snapshot, with one online replica not receiving any incoming traffic at all.


Appendix A  Model Assertions

/*
 * We want at least one of the Replicas to be online, so
 * that our system can always handle requests.
 */
assert AtLeastOneOnline

/*
 * We require all the different IP ranges to be handled by a Replica
 */
assert AllRangesServed
/*
 * Each range gets served by exactly one Replica.
 */
assert RangeServedOnce

/*
 * There is no range that is being served by an offline Replica
 */
assert NoOfflineRanges

/*
 * There cannot be an OfflineMessage reporting that an offline node went offline.
 */
assert OfflineReplicasCannotCrash

/*
 * An online Replica cannot be reported again as online in an OnlineMessage.
 */
assert OnlineReplicasNotInOnlineMessages

/*
 * We require that all Replicas serve the same number of ranges
 */
assert FairSplitOfRanges