The Nearest Replica Can Be Farther Than You Think

Kirill Bogdanov
KTH Royal Institute of Technology
kirillb@kth.se

Miguel Peón-Quirós *
University Complutense of Madrid
mikepeon@gmail.com

Gerald Q. Maguire Jr.
Dejan Kostić
KTH Royal Institute of Technology
maguire@kth.se dmk@kth.se

Abstract

Modern distributed systems are geo-distributed for reasons of increased performance, reliability, and survivability. At the heart of many such systems, e.g., the widely used Cassandra and MongoDB data stores, is an algorithm for choosing a closest set of replicas to service a client request. Suboptimal replica choices due to dynamically changing network conditions result in reduced performance as a result of increased response latency. We present GeoPerf, a tool that tries to automate the process of systematically testing the performance of replica selection algorithms for geo-distributed storage systems. Our key idea is to combine symbolic execution and lightweight modeling to generate a set of inputs that can expose weaknesses in replica selection. As part of our evaluation, we analyzed network round trip times between geographically distributed Amazon EC2 regions, and showed a significant number of daily changes in nearest-K replica orders. We tested Cassandra and MongoDB using our tool, and found bugs in each of these systems. Finally, we use our collected Amazon EC2 latency traces to quantify the time lost due to these bugs. For example due to the bug in Cassandra, the median wasted time for 10% of all requests is above 50 ms.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems; D.2.5 [*Software*]: Software Engineering—Symbolic execution

Keywords Geo-Distributed Systems, Replica Selection Algorithms, Symbolic Execution

* Work done while the author was at IMDEA Networks Institute.

SoCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA.

ACM 978-1-4503-3651-2/15/08…$15.00.
http://dx.doi.org/10.1145/2806777.2806939

1. Introduction

Cloud services implemented on top of third party cloud environments, such as Amazon EC2[1], Microsoft Azure[4], and Google Cloud[2], comprise many useful services and serve hundreds of millions of users spread across the globe. Ensuring low latency when serving user requests is highly important, as it has become one of the differentiating features of the most popular services [20, 21]. Having a reliable service is important, and the service should even survive failures of entire datacenters. Therefore, popular cloud services are typically replicated across geo-distributed datacenters for reasons of performance, reliability, and survivability.

Achieving high throughput and low latency of responses to client requests is a difficult problem for cloud services. Depending on replication policies, the consistency model of a service and the current network state, clients have to choose which replica or set of replicas they will access, using so-called *replica selection algorithms*. These algorithms are expected to consistently make excellent choices in the unstable environment of geo-distributed systems spread across the wide-area.

One of the first questions that we wanted to explore is whether services that are implemented on top of a dedicated cloud infrastructure exhibit the same level of latency variability that is observed across the public Internet (in other words, could we statically configure the replicas once and for all?). To answer this question, we conducted a thorough study of application-level and system-level round-trip latencies across all the Amazon EC2 regions. We find that there is significant variability in network latencies over the course of a day, as well as over several weeks. Next, we evaluated how often a replica selection algorithm would experience a different order of replicas. We find that depending on the number of replicas that are being queried, there could be several tens of reorderings during the course of any given day (even when conservatively removing reorderings shorter than 5 minutes). Thus, static replica selection would be suboptimal even over a dedicated infrastructure.

The replica selection process is inherently hard. The service's clients conduct passive and active measurements of the latency for the served requests, and use this past history

to drive future choices of the replicas to use. Unfortunately, Internet traffic is bursty, and routing (that often does not take network performance into account) frequently changes as well. As a result, end-to-end network bandwidth, latency, and loss rate change dramatically across the wide-area network. To try to compensate for these issues, a replica selection process needs to include mechanisms for filtering and estimating the latency when processing requests. These mechanisms need to answer a number of difficult questions, for example: i) for how long should the past latency samples be kept?; ii) what should be the adaptation rate (new samples preference)?; iii) how to deal with highly variable samples, e.g., outliers (discard some or pay attention to them)?

Unfortunately, errors (suboptimal choices) in replica selection algorithms are extremely hard to find. Such errors usually do not result in critical system failures, and it is hard to determine the optimal behavior in the absence of up-to-date, full global knowledge (needed to detect the presence of an error). Suboptimal replica choices can result in increased latency, and can drive a significant fraction of the customers away. For example, Amazon reports that it loses 1% of sales if response latency increases by 100 ms [27]. Perhaps for similar reasons, Google [20] explicitly tries to eliminate the slowest 1% of request responses in their data centers because even this small fraction of slow responses can cause a large fraction overall of service responses to be delayed when they are composed of hundreds of individual subrequests. While there is a lot of effort to reduce the response latency within a datacenter, this potential advantage can easily be lost by poor (erroneous) replica selection. Thus, it is important that requests are served by replicas that are closer to a client, and the better the selection algorithm the greater reduction in latency.

Debugging replica selection algorithms is difficult for a number of reasons. Bugs can occur due to sampling problems, problems in math calculations, selection logic problems, etc. To thoroughly test the selection algorithms we would need to cover all possible network topologies and their bandwidth, latency, and loss rate characteristics. In addition, we would need to anticipate the exact intensity, duration, and frequency of changes in traffic and routing, which is impossible. [1] Thus, it is unlikely that unit tests and simple (no matter how long) simulations can identify all the bugs. Instead, a *systematic testing tool* is needed.

In this paper we describe GeoPerf, our tool for automating the process of testing replica selection algorithms. We choose to apply symbolic execution [13, 15], because it systematically uses the code itself to identify test inputs that can cause the code under test to examine all branches in the code and ultimately traverse all possible code paths. In our case the inputs are the changing latencies presented to the replica selection mechanisms. As a result, we use the symbolic execution engine to answer the difficult aforementioned questions (e.g., latency change intensity, duration, and frequency) and systematically look for bugs. Using symbolic execution comes with its own set of challenges, however. **First**, detecting errors using symbolic execution requires an application to violate certain invariants, such as an assertion in the code. Specifying these assertions is trivial in the case of memory bugs. Unfortunately, in our case we are dealing with performance deviations and we need the ground truth to define the violation. **Second**, if the symbolic execution engine were to propose inserting latencies that go significantly beyond the observable ones, one could question the relevance of the bugs found. Thus we need a way of using realistic latencies. **Third**, it is easy to run into a path explosion problem due to the large number of possible branches in the code. A large number of symbolic inputs can similarly cause an exponential explosion in state space. Unfortunately, most of the replica selection algorithms contain filtering mechanisms that would require several symbolic variables should symbolic execution be applied blindly.

To provide the ground truth, we use lightweight modeling to approximate what an optimal choice of a replica should be. Our ground truth is a straightforward selection of closest replica(s) based on latencies smoothed-out the same way as in the system under test.[2] This means that the code under test and our model are fed the same latencies in a discrete event simulator, and symbolic execution then tries to find the inputs for the next iteration that would cause a divergence in the choice of replica(s). Any such case is a potential bug. To address the second challenge, we simply reuse the latencies we collected across the wide-area network when running over Amazon EC2 geo-distributed datacenters, and use them to limit the input ranges that the symbolic execution engine is allowed to propose. Finally, to address the third challenge we apply domain-specific knowledge regarding the way the various latency filtering mechanisms work, enabling the symbolic execution engine to work across several iterations. In particular, we make it possible to use only a few symbolic inputs to drive execution along different code paths and produce results in a matter of hours.

We believe this is the first tool for systematically testing geo-distributed replica selection algorithms. We make the following contributions:

1. We conduct thorough round-trip time measurements across *all* geo-distributed datacenters belonging to one cloud provider (Amazon EC2), for several weeks. To the best of our knowledge, no such measurements are publicly available. Using this data, we show that the replica orderings would change up to several tens of times per day, from any given datacenter's viewpoint. Also, we explore the stability of replica positions in the nearest-K order from any given datacenter and find a surprising amount of variability.

[1] If it were possible to do that then we would have the perfect replica selection algorithm!

[2] Unless that module is suspected to be buggy, in which case we change the smoothing mechanism that we use for ground truth.

2. We propose, design, and implement techniques that overcome challenges in applying symbolic execution to testing replica selection algorithms.
3. GeoPerf found bugs in the replica selection of two popular geo-distributed data stores, Cassandra[7] and MongoDB[5]. Interestingly, these bugs belong to different modules. In Cassandra the bug is in the code choosing the replica *after* all the latencies have been examined. We found this bug using an earlier version of our tool, reported it, and it was fixed. In contrast, MongoDB's Java client never clears the buffer that is used to average the latencies. This bug had previously not been reported.
4. We go a step further than the typical bug-finding tools in that we quantify the impact of the bugs that GeoPerf can find. Specifically, we replay the trace of the latencies we collected across Amazon EC2, and compute the cumulative time that is wasted due to the bugs. In the case of Cassandra, the median wasted time for 10% of all requests is above 50 ms.

2. Systems That Use Replica Selection

Cassandra is a highly configurable NoSQL distributed database, designed to work with large datasets in local and geo-distributed environments. Unlike many other distributed databases, Cassandra can perform read operations with a per request variable consistency level, depending on the client's requirements. The client communicates with one of the replicas (presumably the one closest to the client). This node selects a subset of closest replicas (including itself), forwards the client's request to them and waits for their replies. Once enough replicas have replied, a single reply is sent back to the client. The consistency level can vary from LOCAL - *i.e.*, reads from a single node, QUORUM (*i.e.*, $N/2+1$ replicas), to ALL where all nodes have to respond prior to returning an answer to the client.

Generally, a configurable number of nodes will be used to produce an answer, thus allowing a tradeoff between consistency, availability, and performance – as acceptable to the client. While performing a quorum read is well understood, being able to choose a specific subset of nodes opens a number of possibilities, especially if these nodes (replicas) are geo-distributed.

MongoDB is a popular distributed document management database. It supports atomic, strongly consistent write operations on a per document basis and either strong or eventually consistent read operations depending on the client's preferences. MongoDB implements a master-slave replication strategy, with one primary and a number of secondary nodes. All write operations are directed towards the primary node and eventually propagated to the secondary replicas. Replication increases redundancy, data availability, and read throughput for clients that accept eventual consistency semantics. By default, a client's read operations are directed to the primary machine and return strongly consistent data, although the client has an option to use secondary replicas, or to choose the closest node regardless of its current status. We concentrate on the case of choosing the closest node.

2.1 Replica Selection Algorithms

Replica selection systems usually contain two elements: a smoothing algorithm (filter) to get rid of high variability, and the actual replica selection algorithm that acts based on the latencies computed by the filter.

Latency Smoothing In data streaming systems it is common to reduce the weights of older data samples. This is done to smooth out short term high variance in sampled data and gives higher weight to the most recent samples. A well-known example from statistics is a group of Weighted Moving Average (WMA) and Exponentially Weighted Moving Averages (EWMA) functions. This class of functions is commonly used during distance estimation in replica selection algorithms and applied to sampled data, such as RTT or system load.

For latency smoothing, Cassandra uses the off-the-shelf Java Metrics library v2.2.0 [3]. This library implements a special type of a time-decaying function called Forward Decay Sample (FDS) [18] (via the library class ExponentiallyDecayingSample). Similarly to EWMA it gives greater weight to recent samples.

Latency estimation in MongoDB is responsibility of the client's driver, which chooses the closest server to connect to. MongoDB has multiple clients for compatibility with many programming languages, and currently there are more than a dozen different client's drivers (including open source community drivers). Naturally many of them have been implemented by different developers, thus their implementations have significant differences in peer selection. In our work, we concentrate on comparing the C++ and Java drivers, as these are considered to be the most widely used and also represent the most distinct implementations in terms of latency smoothing. Note that the C++ driver uses EWMA with a coefficient of 0.25 applied to new samples, while the Java driver is using a Cumulative Moving Average (CMA) (arithmetic average across all collected values).

Replica selection algorithm Cassandra implements a module called *Snitch* to help each node to choose the best set of replicas to which to direct read operations. There are several types of Snitches [8] that allow administrators to tailor the logic to the deployment environment. In particular, *Dynamic Snitch* automatically chooses the closest node(s), and can be summarized as follows: (1) Asynchronously, each replica contacts every other replica with request messages. The time it took from the request until the reply is received is passed through the Forward Decay Sample smoothing function. This time contains both the RTT network component and the retrieval (service) time at the remote node (the later is an indirect indicator of the load at that node). (2) Periodically, current values from the smoothing filters are collected and normalized, providing a list of scores assigned to each replica. By default, this process is executed every 100 ms. (3)

These scores are used when the local node needs to forward client requests to other replicas. The selection process itself is executed in two stages. First, all replicas are sorted based on their physical location, so that all replicas in the same rack and then in the same datacenter as the source are at the top of the list. Second, the delta of a score is computed from the local node (originator of the query) to all other nodes. If the delta is greater than a threshold (default configuration is 10%) of the difference to the closest node, then all nodes are sorted based on their score. Finally, the top K elements from the list are chosen[3].

Unlike Cassandra, MongoDB drivers (both C++ and Java) rely only on the network RTT to compute network distance to replicas: (1) The collected latency samples are passed through EWMA with the filter coefficient 0.25 and CMA in C++ and Java drivers, respectively. The sampling occurs at a hard-coded heartbeat frequency of 5 s in both drivers. (2) Upon the client's request, the smoothed latencies are used to sort all relevant replicas (i.e. generating a replica set that can answer a query). All replicas that are farther than the default threshold of 15 ms from the closest are filtered out of the list. (3) Finally, one replica is selected at random from the remaining list.

3. GeoPerf

In this section, we describe our approach to systematically test replica selection algorithms. First, we isolate the core logic of the replica selection algorithm. We can use our implementation of the ground truth, which uses light weight modelling, or a different algorithm as a reference. Then, we systematically examine code paths in the original algorithm in an effort to find a case in which the algorithm under test performs worse than the reference one. For path exploration, we use symbolic execution, a common technique for software testing. It incorporates systematic code path exploration with an automatic constraint solver to derive concrete input values that will cause a particular code path within a program to be executed. In our case, the inputs are the latencies that could be observed by the replica selection algorithm under test.

3.1 Symbolic Execution Background

The main concept behind symbolic execution is to replace concrete input values of an application with symbolic variables. A symbolic execution engine is used to control the execution process and operates on symbolic variables. Symbolic variables can be seen as placeholders that can change value along an execution path. An execution path is a sequence of choices taken at each branch point in the code. An example of a branching point in C++ is an *if* statement or a *case* switch. The main purpose of symbolic execution is to test and validate applications, by exploring all reachable code paths and automatically generating test cases for all encountered errors.

For each code path within an application, symbolic execution acquires a set of Path Constraints (PC). This empty set is populated with additional constraints at each branch point in the code. The constraint takes the form of a Boolean equation which is expressed in terms of both symbolic and concrete variables. Automated constraint solvers are used to determine satisfiability of a given PC. The symbolic execution engine maintains a list of all explored code paths and their associated states. The engine iteratively picks a code path to explore. The duration of the exploration can be time bounded or stop when the end of the program is reached or an error is encountered. The choice of the next state to explore can be randomized or driven by heuristics (for example a depth first search would give priority to code paths with a greater depth).

At the start of execution all PCs are initialized with empty sets. At the first branch, the constraint solver evaluates the condition. When the execution of a single code path reaches its termination point such as error, assertions, or end of program, the PC is used to generate a concrete input for every symbolic variable used along that code path. Using the same input will result in the application following the same code path (given that the code is deterministic). Certain PCs may not be satisfiable or not possible to determine due to time limitations or functional limitations of the solver.

3.2 Systematically Exploring Replica Sel. Algorithms

We use the popular open-source symbolic execution engine KLEE[13] with the STP[24] constraint solver. While KLEE (and symbolic execution tools in general) is primarily designed as a test generation tool, it can also be applied to a great variety of other problems. Its ability to generate test cases that satisfy all the constraints can be seen as a solution to a problem defined through code. For example, by symbolically executing code that has a branching point of the form `if ((X+5)*(X-2) == 0){assert(0);}` where `X` is a symbolic variable, the equation will be passed on and solved by an automatic constraint solver. The generated test case will be a solution to the quadratic equation.

Felipe Andres Manzano, in his tutorial[29], demonstrates how KLEE can find an exit out of a classical maze problem by exploring a set of inputs that lead to distinct code paths, and eventually the exit, out of the maze. Thus if program already contains a code path that would solve a particular problem, our task is to define the problem in a clear way so that the symbolic execution tool can isolate a code path (or a set of code paths) that leads to the desired state.

The choice made by the replica selection algorithm is dependent on the system state (i.e., past and current set of network RTTs, time, random numbers) and the algorithm itself as implemented via code. Each potential replica choice is defined by a set of possible code paths that can lead to it. In the compiled programs only a single code path will be

[3] Based on Cassandra V2.0.5.

executed, as determined by the system state at each branch point. However, by symbolically executing this code we can explore alternative code paths that would lead to different choices given the symbolic state of the system. Therefore, for each alternative code path we can determine a set of constraints, and the automated theorem solver will derive a state that would result in the alternative execution path. In other words, we can say what would be the latency over the network paths to cause a replica selection algorithm to pick up one or another subset of nodes.

Forcing a replica selection algorithm to make different choices is not enough to reason about the correctness of that algorithm. To do that, we need to know the ground truth. Alternatively another algorithm can take this role, allowing us to compare two choices in a design exploration.

GeoPerf compares a pair of replica selection algorithms by using symbolic execution and lightweight modeling. It symbolically executes them in a controlled environment, while both algorithms share one view of the network. We use symbolic latencies to check if one of the two algorithms makes a different choice under exactly the same conditions.

However, there are several limitations that need to be addressed. First, symbolically executing an entire distributed system is still difficult and requires inside knowledge of the systems, code modifications and significant computation resources. Unfortunately, this would introduce a lot of code paths that are irrelevant to the replica selection logic. Second, symbolic execution does not have a notion of continuous time, which makes it hard to evaluate replica selection choices. Finally, we want to develop a general purpose tool (a common platform), independent of a single distributed system and suitable for quick prototyping and testing of various algorithms, potentially from different versions of a system or even completely independent solutions.

For all of the above reasons, we isolate the replica selection algorithm from the systems under test, and develop a controlled environment where we can deterministically generate, monitor, and replay events as needed.

The core of the tool is based on our own discrete event-based simulator developed as part of GeoPerf. The setup simulates: i) a set of geo-distributed nodes connected via wide-area network paths, ii) arrival of incoming client requests and iii) a replica selection module that periodically chooses a subset of nodes to serve these requests.

We create two instantiations of the simulator, one using the reference replica selection algorithm and the other the algorithm under test. Both instances run in parallel in identical environments (using synchronized clocks and deterministic synchronized pseudo-random number generators). KLEE is used to drive the exploration of the code paths generating a set of symbolic latencies (i.e., inputs) that characterize the network paths among the nodes. The target of the exploration is to find a sequence of network states that exposes potential weaknesses (bugs) of one of the algorithms

```
1  void main(){
2    Sim A = Sim("algorithmA");
3    Sim B = Sim("algorithmB");
4
5    A.run("RTTs.data");   //preload initial
6    B.run("RTTs.data");   //state
7
8    // declare new symbolic variables
9    int sym_lat = symbolic[nodes][depth];
10
11   timeA = A.run(sym_lat);
12   timeB = B.run(sym_lat);
13
14   assert(timeA < timeB);}
```

Figure 1: Event based simulation pseudocode

by repeatedly demonstrating inferior performance (choices) in the simulated environment.

3.3 Comparing the Selection Algorithms

Consider the code example in Figure 1. At lines 2-3 we create two simulation instances that differ only in the algorithm used for replica selection. On lines 5-6 we preload initial states into both of these simulations by replaying identical sets of latency inputs that we collected earlier. Line 9 declares a set of symbolic variables to be used as an input to future iterations of these two simulations. Lines 11-12 run both simulations with the new symbolic input and collect accumulated request times. Finally, on line 14 we have an assertion that will be triggered when the estimated performance of the simulation guided by the algorithm B is slower than its counterpart A. At this point KLEE will have the set of PCs that led to this state, obtained after exploring the code paths of both simulations.

Next, an automated theorem prover will determine if it is possible to trigger the assertion given the current set of path conditions. If this is the case and the assertion can be triggered, then we can generate a concrete test case that will cause this difference in performance. The concrete values calculated for the symbolic variables are appended to the file "RTTs.data" and will be used during the set-up of the next iteration of the tool.

The pair of algorithms (A and B) are the models of the algorithms from the systems that we test or model. In the process of modeling, we isolate the logic behind the algorithm's implementation and replicate it line by line for our tool. The actual size of such modules is quite small, less than 200 lines of code in both Cassandra's dynamic snitch and MongoDB's drivers.

Ground truth To provide the ground truth when the counterpart algorithm B is not available, we have generated a simplified replica selection mechanism that always selects the node with the lowest delay, without randomization or optimization thresholds. Ground truth is an approximation of the optimal performance and it represents the minimum bound in the achievable latency, provided that there is no caching of requests. We expect that algorithms under

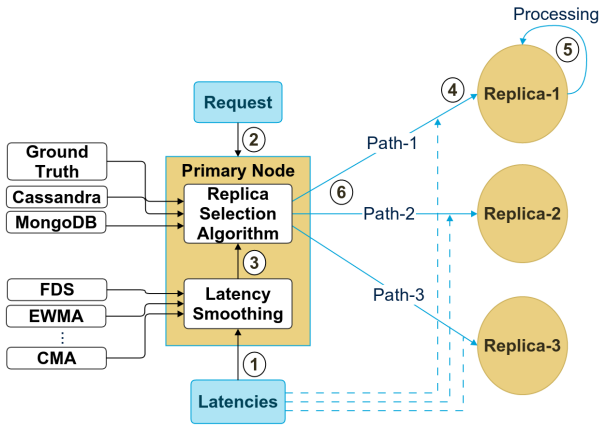


Figure 2: Discrete Event Based Simulation: (1) latencies assigned to inter replica paths and passed through the smoothing filter, (2) client's request generated, (3) the replica selection algorithm is used to chose a closest replica(s) to forward the request, (4) request forwarded to the replica(s) (5) replica processing the request (6) the reply sent to the originating node.

test will demonstrate lower performance than the ground truth in certain cases, but will asymptotically produce comparable results. By default, ground truth uses the same latency smoothing function as the algorithm under test. If the smoothing function itself is suspected to be buggy, it can be replaced.

3.4 Discrete Event Simulation

As a platform for our evaluation we created a discrete event simulator. This simulation models a set of three interconnected geo-distributed replicas and one additional primary node that serves requests (Figure 2). We feed to the simulation the client requests generated at a constant rate, and these are forwarded to the primary node in the set. The primary runs a replica selection algorithm to choose to which replicas it should forward each request. The other nodes in the set are potential replica candidates; they receive the forwarded requests from the primary node, serve them, and send replies back. We also input the set of network latencies that describe RTTs on each path from the primary node to all the replicas. These latencies represent the time it takes for requests to reach the replicas and for the replies to return. As the simulation time progresses, new latency values are introduced to reflect the changing network conditions.

The raw latencies are fed into a smoothing function, such as FDS, EWMA, or CMA according to the specific system being simulated. The output of these functions produces the perceived latency that is used within the replica selection algorithms. Each algorithm makes choices periodically (100 ms for Cassandra and 5 s for MongoDB as per their default configuration). All requests received in that time interval are directed towards the replica set selected at the last decision time. The time it takes to process individual requests is determined by $T_{total} = \frac{RTT_{request}}{2} + T_{processing} + \frac{RTT_{reply}}{2}$

We do not consider request queuing and use a fixed delay of 0.5 ms to generate a response at the replica, i.e., a fixed service time. We obtained this service delay experimentally by running a real system on our hardware, without any load. Such a choice is further motivated by the fact that the processing time is already incorporated into the Cassandra logic, and is completely ignored by the MongoDB logic (in the drivers for replica selection we tested).

3.5 Iterative Search

Ideally, we want to ask the symbolic execution engine to compute the behavior of the system hours ahead of time, and to tell us what sequence of the events and inputs could result in undesired behavior. However, one of the biggest challenges in symbolic execution is the path explosion problem. The number of possible paths grows exponentially and the exploration eventually faces a bottleneck, such as memory or CPU time bounds. The number of possible code paths is exponential in the number of symbolic variables used in the exploration. This becomes the predominant factor that limits the maximum number of symbolic latency variables that we could practically use in our exploration to 9 (3 triplets - 3 latency inputs for 3 replicas in the simulation). The number of triplets defines the depth of the exploration, as each consecutive latency input describes network conditions within the simulation. It is important to note that the number of code paths is also dependent upon the complexity of the algorithm under test. FDS, used in Cassandra's logic, has many more code paths than a simple CMA or EWMA. A single latency triplet input (or a single iteration) corresponds to 100 ms and 5 s of simulated time for Cassandra and MongoDB respectively, as configured in these systems by default.

However, it is not sufficient to show that two algorithms make different choices at a single point in time. It is important to show that this choice exhibits a pathological behavior that can last long enough to cause significant performance degradation for many requests. Thus, we apply an iterative approach by repeating individual explorations.

There are two distinct states to consider. First is the state of the events in the simulation, which describes the queue of the discrete events (*e.g.*, the messages in transition). This state is used to evaluate the performance of the system. Second, the state of the history buffer in the latency smoother is used in the algorithm, for example Cassandra's FDS is configured to remember 100 previous samples. These states may differ from the start of the simulation due to different smoothing filter.

Figure 3 outlines the high level idea behind iterative search. (1) We configure two simulations with a pair of distinct replica selection algorithms, and warm up the system by inserting a set of previously measured latencies to prepopulate the history buffers used by the latency smoothing functions. At this stage, both simulations are in the same state, as replica selection algorithms make identical choices.

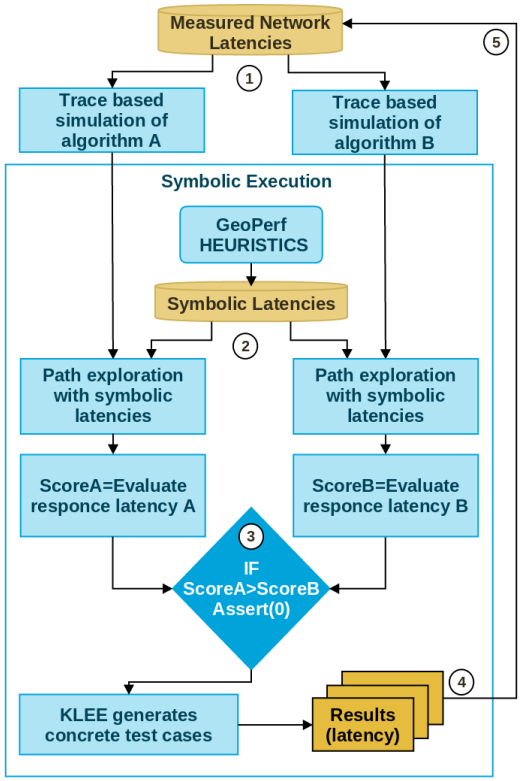


Figure 3: GeoPerf Overview

However since the smoothing techniques can be different (i.e., CMA and EWMA), their states differ as well. Warming up the simulation with the concrete values does not lead to performance penalties, due to the Execution-Generated Testing (EGT) [14] technique implemented in KLEE. EGT allows us to distinguish between concrete and symbolic variables, and avoids creating path constraints if no symbolic variables are involved. (2) As input we introduce a set of symbolic latency triplets, and initiate symbolic exploration. GeoPerf continues the simulation and starts to acquire PCs for all explored paths. After inserting the last symbolic input and finishing all outstanding queries, the performance of both simulations is evaluated. As the scoring function we consider the accumulated time to generate requests initiated during the insertion of symbolic inputs.

At stage (3) we compare the scores. Consequently in the code, we have an assertion checking that the total accumulated time of simulation-1 should be more than accumulated time of simulation-2. The PC is sent to the constraint solver and checked for satisfiability given the current code path. If the PC is not satisfiable then we continue searching until we reach the assertion point through an alternative code path or until we run out of possible code paths, meaning that regardless of the state of the network latencies it is not possible to manipulate the two replica selection algorithms into making different decisions. The continuation uses a different configuration of the search as described in the optimization section. Finally, in the successful case when we reach an assertion, KLEE uses the constraint solver and the obtained PC to generate a concrete test case (i.e., convert symbolic triplets into a concrete set of latencies) that deterministically bring simulation to the same assertion and therefore replica selection choices diverge.

At stage (4) we pick up a newly generated set of latency triplets and append it to our initial set of EC2 latencies. At stage (5) we restart our simulation from the initial point. However in addition to the original set of measured latencies, we also replay the latencies generated in the previous iteration as concrete input.

3.6 Optimizations

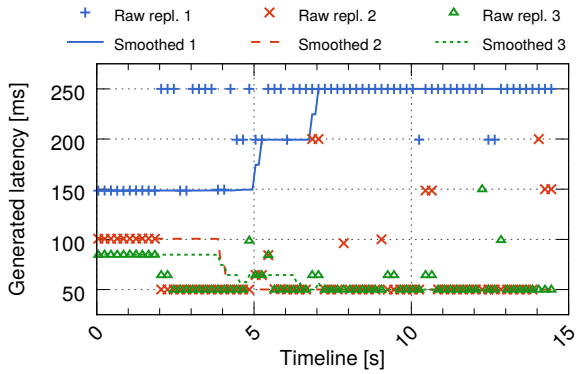
We have developed a heuristic technique to guide the symbolic execution in an effort to find the desired sequence of latency samples. There are three configuration parameters that define the search: maximum search time, number of solutions to find, and number of symbolic variables to use. For each iteration we attempt to find a subset of solutions in the given time limit. If by the end of its time this set is not empty, we sort all our solutions by the improvement ratio that they introduce. However, if we have run out of time or possible code paths, we then attempt to use a different symbolic pattern (SP). In our work we define SP as a set of assumptions (relationships) between the symbolic variables that we input into our simulation. We use the observations of measured samples and reference work [30], to configure SP for each iteration of the search. One observation is that there are different periods of time during which latency does not change significantly (thus we can treat this period of time as if the latency does not actually change). Another observation is that typically only one link will change by a large amount at any given time. By following these observations, we can use a single symbolic variable to define path latency over several iterations of a simulation. Moreover, we can leave the latency of some paths unchanged by reusing values generated in the previous iteration of the search. This technique reduces number of symbolic variables used per iteration and speeds up KLEE code path exploration by simplifying PCs and reducing code path search space.

When our exploration reaches the point when it cannot find a solution with a given configuration, we systematically pick different SPs by re-using symbolic variables over several iterations of the simulation. This optimization allows us to increase the depth of search up to 10 replica selection choices ahead.

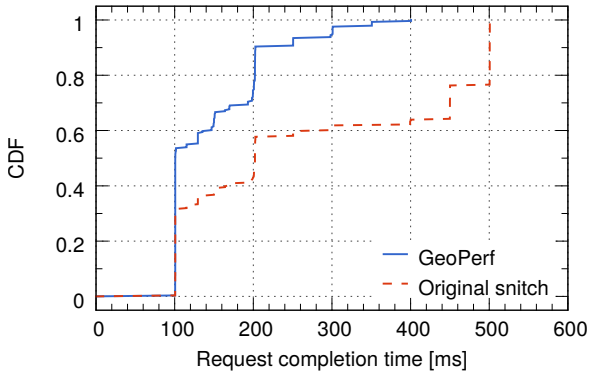
In addition, our experiments show that using integer numbers significantly speeds up constraint solving. Therefore, we replaced floating point operations with their integer equivalent, with three digits of precision.

3.6.1 Limitations

Before we can apply our method, we need to extract the replication selection module from an application in order to plug it into our simulation. If the logic itself is written

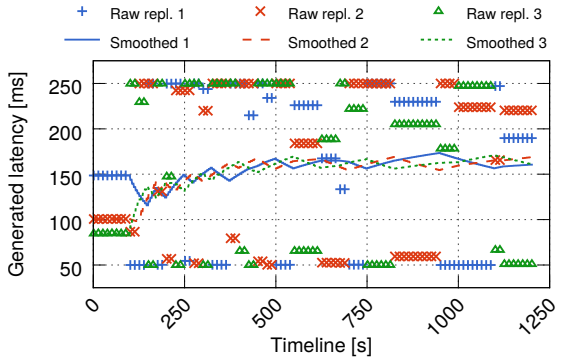


(a) Set of 3 path latencies generated by GeoPerf and the smoothed version of these latencies after FDS

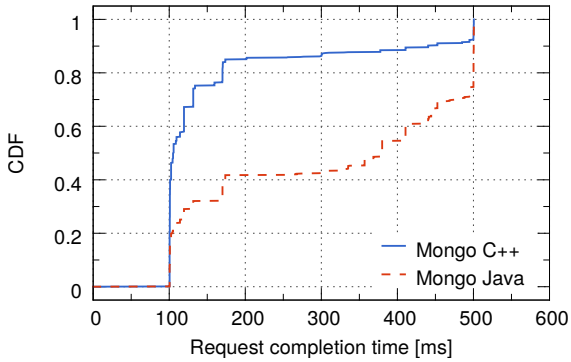


(b) CDF request completion time, Cassandra's Dynamic Snitch compared with GeoPerf's ground truth algorithm

Figure 4: Comparing Cassandra's Dynamic Snitch with the GeoPerf's ground truth



(a) Set of 3 path latencies generated by GeoPerf (points) and the smoothed version of these latencies(lines) after applying MongoDB Java driver's CMA



(b) Request completion time for MongoDB's C++ and Java drivers, as compared by GeoPerf

Figure 5: Comparing MongoDB's drivers using GeoPerf

in a programming language other than the target language processed by the symbolic execution engine, then this module needs to be translated to the target language in order to be executed. Our experience with two different systems implemented in two different programming languages shows that the translation effort is minor.

4. Evaluation

In this section we describe the bugs we found, our latency measurements, and then we quantify the bugs' impact.

4.1 Bugs Found

Cassandra First, we use GeoPerf to evaluate the performance of the Cassandra's Dynamic Snitch against the ground truth model provided with GeoPerf. Both algorithms were configured to use FDS as the latency smoothing function. The GeoPerf's simulations were configured to sample latency at 100 ms intervals. All GeoPerf's explorations were preloaded with 20 samples from EC2 measurements per network path. The RTT range for symbolic latencies was set to 100-500 ms to represent an entire range of observed latencies within EC2. We use fixed request arrival rate of 40 requests per second.

It was sufficient to run GeoPerf for 130 iterations (equivalent to 15 s of real time and 390 symbolic latency inputs) to notice the problem. Figure 4a first, shows the latencies as generated by GeoPerf as well as the smoothed version after passing through FDS. The figure demonstrates that smoothed FDS latencies can follow the general trend and closely represent the real state of the network. However, Figure 4b shows that despite having the correct view of the network, over 20% of Cassandra's requests have RTTs of 500 ms (they are forwarded towards the slowest node of the three). This clearly indicates an issue in the replica selection logic. By examining the code, we have identified that the problem was caused by a bug in the replica score comparison function as follows:
`if ((first-next)/first>BADNESS_THRESHOLD)`. As the value of `next` score could be greater than the `first` score, it results in comparison of the threshold with a negative value. Ultimately, this prevents the sorting function from being called.

MongoDB Next, we compare the performance of the C++ and Java drivers of MongoDB, their only difference was in the smoothing functions used: EWMA(0.25) and CMA for C++ and Java version respectively. In contrast to Cassandra's exploration, GeoPerf was now configured to compare the two provided algorithms (the use of ground truth was not necessary in that case). The system was set with a 5-second sampling interval to match MongoDB's logic. We ran a simulation for a total of 230 iterations (equivalent to 20 min of real time and 690 symbolic entries). First, Figure 5a shows the latencies generated by GeoPerf, and those perceived by MongoDB logic after passing through the Java driver's smoothing function. The latency generated by GeoPerf demonstrates the inability of the Java driver to adapt to periodic latency changes. As time progresses, the Java's CMA smoothing function becomes more resistant to change, until it cannot react to path order changes. As shown on 5b, when Java and C++ drivers are compared under identical conditions, the Java driver demonstrates inferior performance in 80% of the cases. The problem identified here is related to CMA not being reset in the Java driver on a periodic basis.

Although these bugs might appear to be simple, they were not discovered previously. Common software testing often does not provide full test case coverage, thus it is hard to find bugs in application logic that affect performance, particularly in the presence of noisy input (e.g., network latencies). In the case of Cassandra, the official fix for the bug that we found also included a new test case to cover the specific scenario that we pointed out to the developers.

GeoPerf's running time We conduct our explorations using a cluster of 8 heterogeneous machines running Ubuntu 14.04, with a total number of 76 CPU cores and 2GB RAM per core. We use Cloud9 [12] to parallelize and distribute symbolic execution over these machines. Both sets of explorations finish in under 5 hours.

4.2 Geo-Distributed Latencies

In this section, we present our network measurements among the geo-distributed datacenters of Amazon EC2.

Methodology To obtain real world data we instantiated geographically distributed instances of our measurement client on EC2 in all 9 available regions. We measured latencies between each of these datacenters over a period of 2 weeks using t2.micro EC2 instance [4]. We collected samples obtained from 3 different sources: (1) using ICMP ping, (2) application-level TCP ping (Nagle's algorithm was turned off), and (3) application-level UDP ping. These three different sources of samples helped us to understand the potential causes of latency changes. In our measurements, TCP samples demonstrate highest RTTs due to packet re-

[4] Prior to our final measurement, we run a control test using m3.xlarge instances and found that the low bandwidth RTT data does not seem to be affected by the instance size.

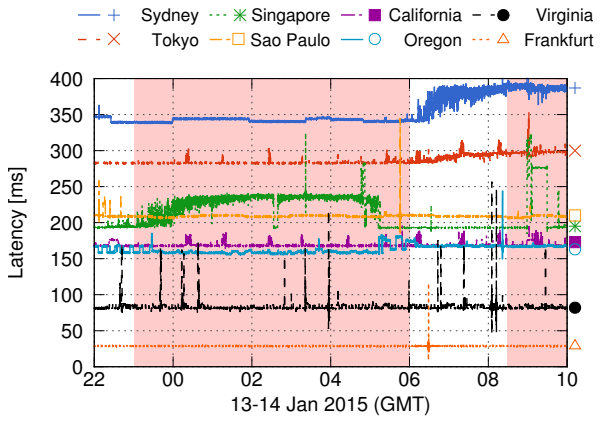


Figure 6: RTT measurement over UDP from EC2 Ireland datacenter to all other 8 regions of EC2 (after low pass filter). Red background highlights the time of the day when the order of replicas change (Singapore-Säo Paulo).

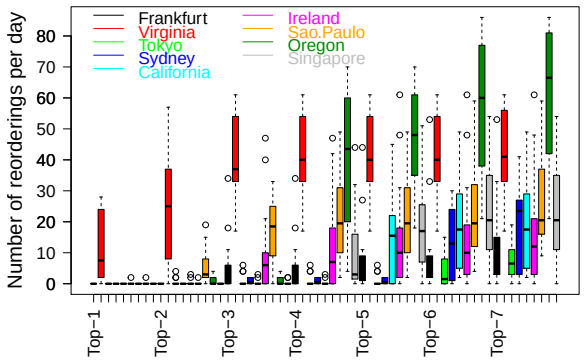
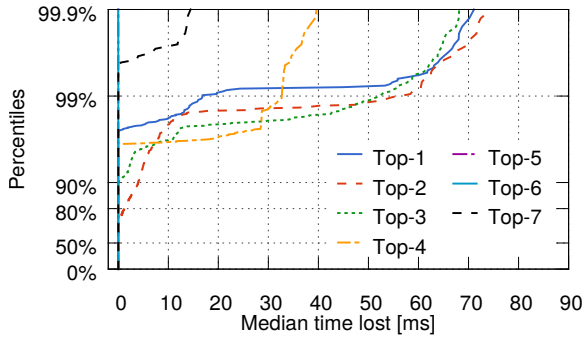


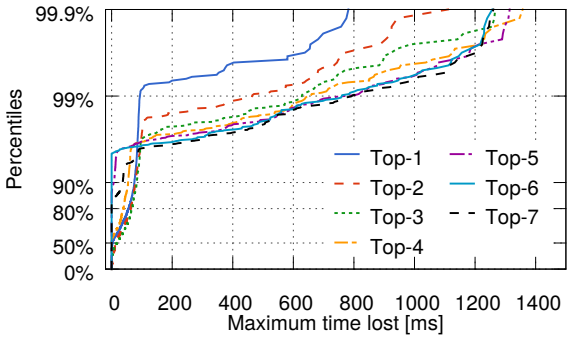
Figure 7: Change of order for the closest K out of total 8 nodes, per region per day over 2 weeks. The color of the boxes corresponds to different EC2 regions. 14 days are aggregated into one boxplot, where the top and the bottom of each box indicate 25th and 75th percentiles, outliers are represented by dots. The horizontal axis indicates the highest indexes of the top K nodes affected. Top-8 reorderings are identical to the Top-7 and thus not shown on the graph.

transmission delays, UDP latencies are equivalent or better than ICMP, potentially due to fast path processing of the UDP packets on many routers. Thus, we chose UDP as our primary dataset for our further analysis.

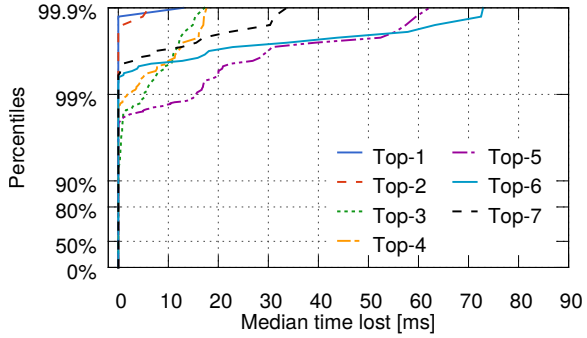
To compute the number of reorderings we perform the following steps. **First:** We group paths based on their source region, such that we consider only realistic subsets of paths to order. For example, from the viewpoint of an instance of a latency-aware geo-distributed system, running in the Ireland datacenter latency is measured from itself to all other 8 regions, while the paths between the other replicas are not relevant. **Second:** We have applied a low pass filter to our samples in order to remove high frequency noise and high variance (we computed an equivalent to Figure 6 for each node). To do that we have converted the latency samples to the frequency domain using the Fast Fourier Transform,



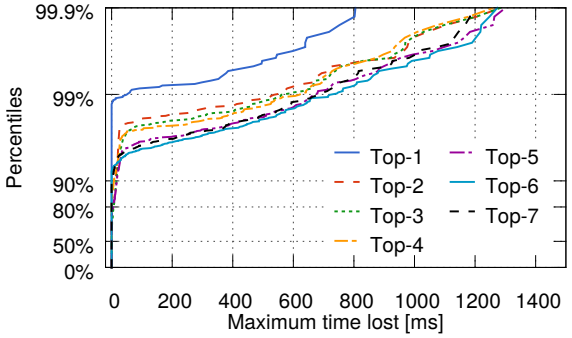
(a) Median time lost from Virginia



(b) Maximum time lost from Virginia



(c) Median time lost from Ireland



(d) Maximum time lost from Ireland

Figure 8: Median and maximum time wasted for the window size of 5 min from Ireland and Virginia. Including more replicas typically increases the maximum penalty, but can produce more stability by going beyond replica positions with high variance. Top-8 configurations use all available replicas and by default perform optimally and thus not shown here.

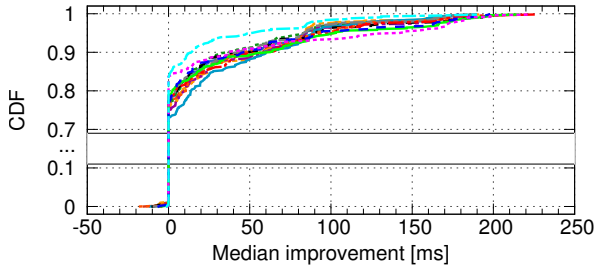
and nullified frequencies higher than 1 Hz. Finally the data was converted back to the time domain using an inverse FFT. **Third:** For each group of links we counted the number of order-changing events and recorded their durations. We ignore events with a duration below a threshold, currently 5 minutes. **Finally:** For each event we identified the indices of the replicas that were affected. The highest index was used to generate Figure 7.

The Case for Replica Selection Figure 6 shows RTT measurements over a period of one day from the Ireland EC2 datacenter to datacenters in all other regions of EC2. For visual convenience, we applied a low pass filter to the raw data to remove high frequency noise. This figure shows that from the point of view of the Ireland datacenter, the order of paths based on their RTT changes periodically. Path reordering occurs when two latency curves cross over each other. The importance of any such event is defined by its duration and the change in the ordering of the top N nodes that are affected by this event. Changing replica ordering too frequently is not desirable, as this may negatively affect caching and other aspects of performance. As a result changing the order or replicates should only be done after some period of time. Additionally, the order of the closest replicas has greater impact than a change in the order of the farther nodes, for example consider the top `3` or `top N/2+1` (where `N` is a total number of replicas) that would be the first candidates for a query. The behavior shown in Figure 6 demonstrates that **any** static configuration will perform suboptimally for a significant amount of time. The magnitude of latency changes in our measurements often exceeds default sorting thresholds in the tested replica selection algorithms, thus making these reorderings significant.

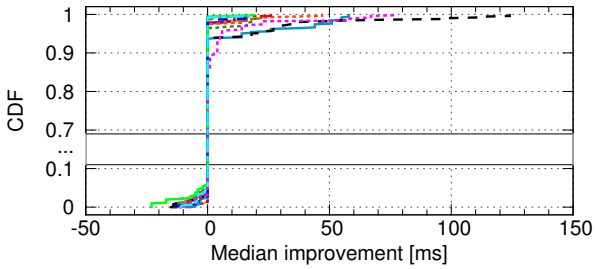
The Global View To achieve a global view, we tried to answer the question of how often *significant* path reorderings occur across all regions of EC2. Figure 7 contains aggregated data showing summaries for two weeks of continuous latency sampling. This figure contains data for different days of the week and different datacenters. The vertical axis shows the number of path reorderings that happened during a day, while the horizontal axis shows the highest index of those paths that have been affected. These plots show how often a developer can expect the replica order to change based on the geographic location of the datacenter and subset of the closest replicas that are relevant for a specific application. For example, if your datacenter is in California and you perform strong consistency reads implemented by a quorum of nodes ($8/2+1 = 5$ nodes) then the median number of reorderings is 15 per day.

4.3 Exploring Replica Position Stability

Next, we explore the stability of replica positions in the nearest-K order from any given datacenter. We apply a low-



(a) Cassandra's buggy version compared against fixed version



(b) MongoDB's C++ and Java drivers compared

Figure 9: The CDFs of median request completion time difference (EC2 latency trace replay via GeoPerf). Each figure contains 14 CDFs, one for each day of the trace of latency samples.

pass filter to the observed latencies to remove frequencies higher than 1 Hz. We divide the sampling period into intervals of 5 min, and at the start of each interval we determine a static replica choice based on the median latencies observed in the previous interval. These choices mimic the typical replica selection algorithm behavior, and remain static for the duration of the 5-min window. For each sample, we then determine the difference between the static choice and optimal choice at points in time separated by 1 second. At the end of the interval, we output the median and maximum difference encountered during that interval. This difference indicates the time one would lose while remaining with a static replica set chosen based on past interval performance. We repeat these computations to cover all possible Top-K combinations, in which we consider the delay between the datacenter at hand and its closest K replicas. Top-2 for example refers to the case of a total of three replicas.

Figure 8 contains two sets of graphs showing the median and maximum delays observed from the Ireland and Virginia datacenters. We chose these datacenters because they are the closest ones to large fractions of users in Europe and the Eastern US. The results are surprising. For example, having only two additional replicas dramatically increases the uncertainty and variance as observed from the Virginia datacenter, as shown by the significantly more time intervals affected (Y-axis) and median time lost (X-axis) in Figure 8a. In particular, more than 29% of time intervals suffer some median time lost, up from 3% for Top-1. Adding more replicas can decrease the uncertainty, for example Top-5 shows only 0.1% time intervals with time lost. This behavior is directly driven by the variability on the paths from the datacenter at hand and other replicas. Going to Top-7 replicas for example again increases the variance (0.3% affected time intervals). However, adding more replicas makes the selection process more vulnerable to unexpected delays and this increases the maximum time lost, as shown in Figure 8b. This figure also shows that roughly 1% of time intervals experience latency penalty of about 600 ms. This is at least an order of magnitude greater penalty than the median time lost (Figure 8a).

The view from the Ireland datacenter is qualitatively the same (Figure 8c), but the unstable replica positions and the times lost differ. Here, Top-4 replicas show fairly low uncertainty with 1.4% time intervals affected. However, uncertainty is harder to eliminate by more than a factor of 10; close to 1.5% of intervals show some median time lost for Top-4, as opposed to 0.1% for Top-5 from Virginia. In contrast, Top-5 from Ireland shows the worst variance.

These findings demonstrate: i) the need for careful adaptation by the replica selection mechanisms, and ii) the difficulty in producing robust algorithms that work well across a variety of network conditions.

4.4 Evaluating the Impact of the Bugs

Here we quantify the potential impact of the bugs found by GeoPerf in both systems under real world conditions. We set up our simulations as in the previous section, and use 14 consecutive days of EC2 latencies (from Tue, 06 Jan 2015, 9GMT) obtained earlier as a concrete input set to GeoPerf. To consider the possible scenarios, first we group the latency samples based on the originating region. Then, from each group we pick up combinations of triplets (3-way replication is a popular, straightforward choice). We have sampled 9 regions, where each region has 8 potential destinations, producing $9\binom{8}{3} = 504$ combinations in total.

Figure 9 shows the median time gained per request per day. For Cassandra's evaluation, instead of using the ground truth model we used the original version of the Dynamic Snitch and the fixed version of the same Snitch. Figure 9a shows that over 20% of all requests were affected by the bug. The median loss for 10% of all requests is above 50 ms.

Next, we compare the C++ and Java MongoDB drivers. Figure 9b shows the effects of using CMA in a dynamic network environment. Over 10% of all requests were unable to react to changing conditions, which resulted in a long tail. The negative CDF tail on both Figures 9a and 9b is explained by two factors. First both systems choose a replica at random if it passes a latency threshold, which accounts for a certain amount of a slightly worse replica to be chosen by the fixed algorithms. Second, when path latency shows a high variance there is a certain amount of inertia in both Cassandra's FDS and MongoDB C++ drivers EWMA, which results in gaining extra time when latency returns to its mean value.

In summary, these findings demonstrate the significance of the bugs found in both systems, and the potential time loss in cloud services due to these bugs.

5. Related Work

Latency Measurements Existing work in the area of latency measurements can be generalized in two categories. The first category [9, 35, 37, 40, 42] concentrates on studying the impact of virtualization on the network and application performance in a cloud environment. These works show that sharing of hardware resources can have a negative effect on latency, throughput, and bandwidth of applications running on these virtual machines. Moreover, these performance artifacts are very different from those that occur in non-virtualized environments, and the artifacts are often hard to predict. Note that all of these studies look at the performance *within* a single datacenter, whereas GeoPerf is concerned primarily with the latency across the wide-area.

The second group of works [10, 30, 32] studies packet loss, delay, and jitter over the geo-distributed WAN network paths. An interesting comparison of cloud providers is done in CloudCmp [26], where the authors computed the CDF for the optimal paths from 260 clients provided by the PlanetLab testbed [16] to the datacenters of four cloud providers; three datacenters from Amazon were tested among other configurations. They report that the latencies among datacenters are highly dependent on their geographic locations. Latencies between different cloud providers are often incomparable due to the different geographic locations of their datacenters. These results suggest that any static configuration of the replica selection algorithms should be always tailored to the deployment on a specific cloud provider.

However, to the best of our knowledge, our study is the first to show the correlation of geo-distributed network paths among all regions of one cloud provider, and to demonstrate how such dynamics can affect the nearest replica selection.

Symbolic Execution Symbolic execution has a long history, and some of the first works are [11, 17]. EXE [15], KLEE [13], JPF-SE [6], CUTE and jCUTE [36] are modern examples of symbolic execution tools, and most of them are being updated by the academia and open source communities.

However, symbolic execution is not used only to discover common bugs in various applications. For example, Siegel [38] uses the SPIN model checker and symbolic execution techniques to compare and verify correctness of the Message Passing parallel implementation of the algorithms given its sequential counterpart. Person [33] proposes differential symbolic execution to determine differences between two versions of the same application. KleeNet [34] applies symbolic execution to a network of wireless sensors. It runs on unmodified applications, while automatically injecting non-deterministic events common for distributed systems. GeoPerf, while performing tests on algorithms that are designed for distributed systems, does not require non deterministic events or event reorderings; the replica selection algorithms are explored in a completely deterministic environment. This dramatically reduces the number of possible code paths to explore, and therefore reduces the search time requirements. Relative to all these approaches, our contribution is applying symbolic execution to a new problem, and overcoming the difficulties that arise in this case. In particular, we deal with the performance-related (latency) issues.

Replica Selection algorithms Geo-distributed storage systems tend to forward client's requests towards the "closest" replicas to minimize network delay and to provide the best performance. This task commonly occurs, e.g., in self-organizing overlays [39]. One of the primary tasks is to correctly compute or estimate the distance among the nodes; various systems have tackled this problem. Vivaldi [19] piggy-backs probes on the application messages in order to infer virtual network coordinates. GNP [31] performs measurements from the dedicated vantage points to compute network coordinates. Alternative approaches for estimating network distance include [22, 25, 28].

The next step is to use estimated distances to select an appropriate replica to contact. Meridian [43] is a decentralized, lightweight overlay network that can estimate the distance to a node in the network by performing a set of pings that are spaced logarithmically from the target. OASIS [23] is a system built on top of Meridian, and also incorporates server load information. DONAR [41] also argues for a decentralized approach, by using a set of mapping nodes, each running a distributed selection algorithm to determine and assign the closest replica for each client. In our work, we do not argue for the best replica selection algorithm, but rather provide a tool that can find flaws in the performance of such algorithms.

6. Conclusion

In this paper, we first demonstrate the need for dynamic replica selection within a geo-distributed environment on a public cloud. Second, we propose a novel technique of combining symbolic execution with lightweight modeling to generate a sequential set of latency inputs that can demonstrate weaknesses in replica selection algorithms. We have created a general purpose system capable of finding bugs and weaknesses in replica selection algorithms. We use our system GeoPerf to analyze the replica selection logic in Cassandra and MongoDB datastores, and find a bug in each of them. We plan to release our software and latency samples as open source.

Acknowledgments We thank our shepherd Douglas Terry, and the anonymous reviewers for their feedback. We are grateful to Peter Peresini for code reviews, and comments on earlier drafts. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

References

[1] Amazon ec2. http://aws.amazon.com/ec2/.

[2] Google cloud. https://cloud.google.com/.

[3] Java metrics library. https://github.com/dropwizard/metrics.

[4] Microsoft azure. http://azure.microsoft.com/.

[5] Mongodb. http://www.mongodb.org/.

[6] ANAND, S., PĂSĂREANU, C. S., AND VISSER, W. JPF–SE: A symbolic execution extension to Java pathfinder. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 134–138.

[7] APACHE. Cassandra. http://cassandra.apache.org/.

[8] APACHE. Cassandra, snitch types. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html.

[9] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 242–253.

[10] BOLOT, J.-C. End-to-end packet delay and loss behavior in the internet. *ACM SIGCOMM Computer Communication Review 23*, 4 (1993), 289–298.

[11] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. Selecta formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices 10*, 6 (1975), 234–245.

[12] BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems* (2011), ACM, pp. 183–198.

[13] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[14] CADAR, C., AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software*. Springer, 2005, pp. 2–23.

[15] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *CCS* (2006).

[16] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review 33*, 3 (2003), 3–12.

[17] CLARKE, L. A. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, 3 (1976), 215–222.

[18] CORMODE, G., SHKAPENYUK, V., SRIVASTAVA, D., AND XU, B. Forward decay: A practical time decay model for streaming systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on* (2009), IEEE, pp. 138–149.

[19] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM Computer Communication Review* (2004), vol. 34, ACM, pp. 15–26.

[20] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[21] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP* (2007).

[22] FRANCIS, P., JAMIN, S., JIN, C., JIN, Y., RAZ, D., SHAVITT, Y., AND ZHANG, L. Idmaps: A global internet host distance estimation service. *Networking, IEEE/ACM Transactions on 9*, 5 (2001), 525–540.

[23] FREEDMAN, M. J., LAKSHMINARAYANAN, K., AND MAZIÈRES, D. Oasis: Anycast for any service. In *NSDI* (2006), vol. 6, pp. 10–10.

[24] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification* (2007), Springer, pp. 519–531.

[25] LEDLIE, J., GARDNER, P., AND SELTZER, M. I. Network coordinates in the wild. In *NSDI* (2007), vol. 7, pp. 299–311.

[26] LI, A., YANG, X., KANDULA, S., AND ZHANG, M. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 1–14.

[27] LINDEN, G. Make Data Useful. https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt.

[28] MADHYASTHA, H. V., ISDAL, T., PIATEK, M., DIXON, C., ANDERSON, T., KRISHNAMURTHY, A., AND VENKATARAMANI, A. iplane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 367–380.

[29] MANZANO, F. A. The symbolic maze! https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/.

[30] MARKOPOULOU, A., TOBAGI, F., AND KARAM, M. Loss and delay measurements of internet backbones. *Computer Communications 29*, 10 (2006), 1590–1604.

[31] NG, T. E., AND ZHANG, H. Predicting internet network distance with coordinates-based approaches. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2002), vol. 1, IEEE, pp. 170–179.

[32] PAXSON, V. End-to-end routing behavior in the internet. *ACM SIGCOMM Computer Communication Review 36*, 5 (2006), 41–56.

[33] PERSON, S., DWYER, M. B., ELBAUM, S., AND PSREANU, C. S. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (2008), ACM, pp. 226–237.

[34] SASNAUSKAS, R., LANDSIEDEL, O., ALIZAI, M. H., WEISE, C., KOWALEWSKI, S., AND WEHRLE, K. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th*

ACM/IEEE International Conference on Information Processing in Sensor Networks (2010), ACM, pp. 186–196.

[35] SCHAD, J., DITTRICH, J., AND QUIANÉ-RUIZ, J.-A. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 460–471.

[36] SEN, K., AND AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification* (2006), Springer, pp. 419–423.

[37] SHEA, R., WANG, F., WANG, H., AND LIU, J. A deep investigation into network performance in virtual machine based cloud environments.

[38] SIEGEL, S. F., MIRONOVA, A., AVRUNIN, G. S., AND CLARKE, L. A. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the 2006 international symposium on Software testing and analysis* (2006), ACM, pp. 157–168.

[39] VAHDAT, A., CHASE, J. S., BRAYNARD, R., KOSTIĆ, D., REYNOLDS, P., AND RODRIGUEZ, A. Self-organizing subsets: From each according to his abilities, to each according to his needs. In *IPTPS* (2002), vol. 2429 of *Lecture Notes in Computer Science*, Springer, pp. 76–84.

[40] WANG, G., AND NG, T. E. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE* (2010), IEEE, pp. 1–9.

[41] WENDELL, P., JIANG, J. W., FREEDMAN, M. J., AND REXFORD, J. Donar: decentralized server selection for cloud services. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 231–242.

[42] WHITEAKER, J., SCHNEIDER, F., AND TEIXEIRA, R. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review 41*, 1 (2011), 38–44.

[43] WONG, B., SLIVKINS, A., AND SIRER, E. G. Meridian: A lightweight network location service without virtual coordinates. In *ACM SIGCOMM Computer Communication Review* (2005), vol. 35, ACM, pp. 85–96.