# Verification in the Age of Microservices

Aurojit Panda
UC Berkeley

Mooly Sagiv
Tel Aviv University

Scott Shenker
UC Berkeley and ICSI

## ABSTRACT

Many large applications are now built using collections of microservices, each of which is deployed in isolated containers and which interact with each other through the use of remote procedure calls (RPCs). The use of microservices improves scalability – each component of an application can be scaled independently – and deployability. However, such applications are inherently distributed and current tools do not provide mechanisms to reason about and ensure their global behavior. In this paper we argue that recent advances in formal methods and software packet processing pave the path towards building mechanisms that can ensure correctness for such systems, both when they are being built and at runtime. These techniques impose minimal runtime overheads and are amenable to production deployments.

## CCS CONCEPTS

• **Computer systems organization** → **Reliability**; • **Networks** → *Middle boxes / network appliances*;

**ACM Reference format:**
Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2017. Verification in the Age of Microservices. In *Proceedings of HotOS '17, Whistler , BC, Canada, May 08-10, 2017,* 7 pages.
https://doi.org/http://dx.doi.org/10.1145/3102980.3102986

## 1 INTRODUCTION

Web applications are increasingly built and deployed as sets of isolated services which interact with each other through remote procedure calls (RPCs). The advent of lightweight virtualization techniques — *e.g.,* containers — has enabled the use of increasingly larger numbers of fine grained services which are commonly referred to as *microservices*. Decoupling applications in this manner yields several benefits: it simplifies scaling (each service can be scaled independently), provides greater flexibility in resource allocation and scheduling, allows greater code reuse, enables new fault tolerance mechanisms, provides better modularity, and allows application developers to take advantage of services from other provides *e.g.,* Amazon S3. As a result this architecture has been widely adopted by both startups and large established companies (*e.g.,* Uber [27] and Netflix [28]), and is being deployed at significant scale (*e.g.,* Uber's application is composed of over 1000 microservices [27]).

As deployments grow in size and complexity, it has become harder for operators to ensure the correctness for these microservice-based applications. The properties of such applications depend on the behavior of each constituent microservice and how they are configured to interact. Understanding whether certain *invariants* are upheld during their operation is nontrivial for small applications but downright daunting for ones involving hundreds of microservices. For example, consider a simple application with three services — a webserver, an authentication service, and a database. Ensuring the invariant that only authenticated users can update the database requires accessing state at both the webserver (to determine what request resulted in an update) and the authentication service (to check if the requester was authenticated), in addition to accessing database state. Even if all microservices provided mechanisms to access local state, naively checking the invariant would require coordination — to get a consistent snapshot of system state — whenever the invariant needed to be tested, negatively impacting performance. Scaling such techniques to large applications would be untenable.

In this paper we propose a system, `ucheck`, that checks and enforces invariants in microservice based applications. In `ucheck` invariants (§2.2) specify sequences of RPC calls which indicate erroneous behavior — *e.g.,* insertion calls to the database before a corresponding authentication call. We designed `ucheck` so that it requires no coordination, and imposes minimal performance overheads, and as a result it is amenable to being deployed in production.

`ucheck` is designed around modular microservice models (§2.3) which specify the set of messages a microservice can send, and how its local state changes in response to receiving a message. Given these models our approach is simple: we first use formal verification to check whether a given invariant would hold based on the models (§3.1), and then we use programmable virtual switches (vswitches, *e.g.,* Bess [11] or VPP [29]) to detect cases where a microservice's behavior deviates from what is specified by the model. This allows


*HotOS '17, May 08-10, 2017, Whistler , BC, Canada*

ACM ISBN 978-1-4503-5068-6/17/05…$$15.00
https://doi.org/http://dx.doi.org/10.1145/3102980.3102986

ucheck to detect *potential* invariant violation at runtime without requiring coordination. However, because ucheck does not access any microservice state, it cannot detect all invariant violations – we discuss this in greater detail in §5.1 and also present possible mitigations.

Given the increasing scale and complexity of microservice-based applications, we believe it is necessary to help operators reason about and ensure the correctness of their applications – ucheck represents a first step towards this goal.

## 2 APPLICATIONS AND INPUTS

We begin by providing an overview of microservice based applications, and the invariants and models that are the inputs to ucheck.

### 2.1 Microservice Based Application

ucheck is designed to be used with application built by composing multiple *microservices*. Each microservice runs in a single *container*, and each container can communicate with others through a virtual network [15]. We assume that each containers is connected to the virtual network through a software virtual switch. Microservices interact with each other by sending messages through the virtual network. We assume these messages are sent using a RPC mechanism *e.g.,* GRPC [10] or Thrift [26].

We use a simple web forum (Figure 1) as a running example through the rest of this paper. We assume the web forum is built using an authentication service, a key-value store (kv-store) microservice and a frontend microservice. The authentication service provides a single RPC endpoint, authenticate, that can be used to check if a client's credential are correct. The key-value store provides RPC endpoints that can be used to insert, modify, get and delete key-value pairs. The frontend microservice receives and processes three types of HTTP requests – authenticate requests that a client can use to establish its identity, get requests in response to which it gets values from the kv-store, and post requests which can result in it inserting a value into the kv-store.

### 2.2 Invariants

In this paper we focus on enforcing safety invariants, which identify sequences of RPC calls (including predicates on inputs and outputs) and results that are prohibited. An invariant is violated whenever such a sequence of calls is observed. For example, an application developer might require that posted messages are never deleted or modified. This can be expressed as a (stronger) invariant requiring that no modify or delete calls be made to the key-value store. Similarly, the application developer might also require that users authenticate before messages can be posted. This can be expressed as

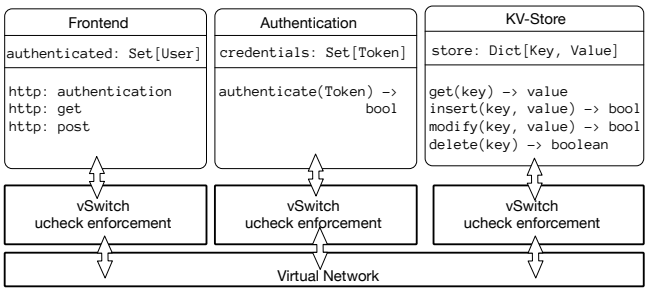


**Figure 1: We use a web forum as a running example throughout this paper. The webforum is comprised of three microservices: a key-value store (kv-store), an authentication service, and a frontend webservice. The frontend webservice receives HTTP request (indicated by the `http:` prefix) and makes RPC calls to the key-value store (kv-store) and authentication service. In this figure we specify the local state and RPC endpoints for each microservice. The web forum is correct if two invariants hold: (a) posted messages are never modified or deleted, and (b) only authenticated users can post messages. All microservices interact by seending messages, which must pass through a virtal switch. `ucheck`'s enforcement mechanism is implemented in the vswitch.**

the invariant that the frontend sends no *insert* message to the key-value store, before sending an *authenticate* message and receiving a response indicating the client is authenticated. Observe that in general invariants can specify a sequence of RPC calls across multiple microservices. As a result, enforcing an invariant might require coordination across microservices – which in the worse case requires communication between all microservices in an application – adds unacceptable performance overheads. As a result, our design does not directly enforce invariants, and instead uses abstract models (§2.3) and static verification (§3.1) to translate these to restrictions on individual microservices which can be checked *without coordination*.

### 2.3 Microservice Models

ucheck assumes that users provide it with a model for each microservice. This model serves two purposes – first we rely on formal verification to ensure that if all supplied models were true the invariant would hold, and second we enforce that the model itself is correct. A model must therefore provide enough semantics about a services' behavior to allow supplied invariants to be verified and must also specify constraints on RPC calls and results that can be used by our enforcement mechanism. Prior work has show than modular reasoning techniques such as rely-guarantee [14] reasoning are well suited to verifying invariants in concurrent systems such as

the applications we consider in this paper. Furthermore, prior work [9] has looked at decomposing rely-guarantee conditions when verifying concurrent programs, allowing verification to be performed on each function. Our architecture builds on this work – our models are specified as four tuples each of which contains a set of *global preconditions* (the next message processed by the microservice), *local preconditions* (local state at the microservice), *local post condition* (resulting local condition from processing a message) and *global post condition* (new messages generated by the microservice). We use global preconditions and postconditions for enforcement (§3.2). Note, that since we ignore local state, our enforcement is necessarily approximate and we might miss cases where invariants are violated. We discuss this limitation and possible mitigations in §5.1.

As an example the model for the frontend service in the web forum would include the following:

(1) When an authentication request is received from a client (a *global precondition*), issue an `authenticate` RPC for the authentication microservice (a *global postcondition*).

(2) When the authentication microservice response is pending and it indicates the authentication succeeded (a *global precondition*), update the set of authenticated clients (a *local post condition*) and send an indication to the client (a *global postcondition*).

(3) When a post request is received (a *global precondition*) and if the client has been authenticated (a *local precondition*) then issue a *insert* request to the kv-store microservice.

**Who writes the models?** Writing and maintaining models is an important concern with systems such as `ucheck`. This is especially true when models cannot be derived from code, since any model written by a programmer is likely to be wrong as code evolves. In this work we assume that models are used by an application developer – who is combining multiple microservices – to specify their beliefs about each microservice. In the common case we therefore expect these models to be supplied by the application developer, and we rely on our enforcement mechanisms to catch a mismatch between a microservice's model and its actual behavior. Models can however come from other sources including – service developers who might produce models as specification; third parties *e.g.,* auditors, who might use models to record conditions that are required for security; and finally models might be produces as a part of writing a service in frameworks such as Verdi [30], IronFleet [12], and Yggdarsil [24]. We discuss `ucheck`'s relation to these works in greater detail in §5.2.

# 3 PREVENTING INVARIANT VIOLATIONS

## 3.1 Static Verification

`ucheck` relies on a combination of static verification and enforcement to ensure correctness. We require developers to verify each invariant given the supplied models, this serves the dual purpose of ensuring that (a) the invariant would actually hold given how services are thought to act, and (b) ensure that the models (and hence the preconditions and postconditions) are strong enough to prove the invariant. Beyond requiring the use of models that can be decomposed into local and global rely-guarantee constraints, `ucheck` imposes no restrictions or requirements on the verification process. As a result verification can be performed by either manually generating a proof and relying on Coq [18] or other theorem provers to check the proof, or using traditional model checking tools such as NuSMV [4].

We also note that the performance of our enforcement mechanism's worsens as the size of the model grows. One might be able to use tools such as MAX-SAT solvers [17] or CEGAR [7] to simplify models during this static verification step. Investigating these techniques and their potential benefit is left to future work.

## 3.2 Runtime Enforcement

While static verification ensures that invariants are upheld assuming microservices behave as modelled, it cannot prevent violations in cases where a microservice's behavior diverges from what is allowed by the model. This can happen for a variety of reasons including errors in the input model, due to bugs (such as buffer overruns or underruns) or malicious attacks that change executing code, etc. Therefore, we rely on runtime enforcement mechanisms that can detect and handle cases where a microservice's behavior diverges from what has been modelled.

Our enforcement mechanism is largely designed to run at the virtual network layer. All microservices in an application are connected through a virtual network, which is generally implemented using one or more virtual switches *e.g.,* OVS [20] or the Linux Bridge [16]. Virtual switches have visibility into all network traffic received or sent by a microservice, based on our assumption this means they have access to all RPC requests and response, along with user requests (*e.g.,* HTTP requests to the web forum front end). Finally we observe that increasingly vswitches, *e.g.,* Bess [11] and VPP [29], provide mechanisms to perform complex processing on network traffic. In both Bess and VPP operators specify a packet processing pipeline – which consists of a sequence of modules written in a regular programming language (C++) – through which all traffic is sent. We implement `ucheck`'s

runtime enforcement mechanism as one such module, that we then ensure has visibility into traffic sent by all microservices.

The `ucheck` enforcement module must implement four basic mechanisms – first, it must be able to distinguish between external communication (*i.e.,* communication between the application and external client, *e.g.,* web browsers) and internal communication (*i.e.,* communication between microservices belonging to the same application); second, it must be able to convert raw network traffic (*i.e.,* bytes) into semantically meaningful messages; third, it must be able to associate each message with a particular application – this is required since a single microservice might be shared by multiple applications, and each application might assume different models for the same microservice; and fourth, it must detect situations where messages sent or received by a microservice do not correspond to its model. We assume that we can distinguishing between external and internal communication by looking at the source and destination for each packet. To ensure correct routing a container orchestration service (*e.g.,* Kubernetes [3]) must configure the virtual network with information about the location and address of all microservices, the `ucheck` module merely reuses this information to distinguish between external and internal traffic. We rely on TCP byte stream reconstruction and the deserialization functionality implemented by the RPC library to convert raw bytes into messages. TCP byte stream reconstruction is widely used (in systems such as Bro [19]) and prior work has looked at efficient and safe reconstruction techniques [13]. We assume that all messages carry metadata associating them with an application, this is required both by logging services (*e.g.,* X-Trace [8]) and for billing in shared services.

We detect divergence between observed behavior and models for a microservice by comparing messages sent by the microservice against what is allowed by its model (§2.3). To do this we compute a static set of messages that can be plausibly sent (or received) by the service – this is equivalent to computing the set of all messages specified in the model. Whenever a microservice sends (or receives) a message we check to see if an equivalent can be found in the set of plausible messages (which we represent as a compact predicate, rather than as an actual set) – if so we allow it through and raise an exception (indicating an invariant violation) otherwise. Note that this enforcement mechanism checks a weaker model than is provided by the user – for example if we consider the frontend microservice (model in §2.3) we can see that `insert` calls should only be generated in response to post messages from authenticated client. However, our enforcement strategy would allow all `insert` calls through, regardless of the causal sequence leading up to the call being made. Such an approximation is necessary for two reasons: (a) first we assume no access into a microservice's private data, and (b) inferring local state from previous messages can require looking through a potentially unbounded sequence of messages, which can severely slow down enforcement. Investigating techniques that allow us to make trade-offs between performance and accuracy in enforcement is left to future work. We discuss strategies for mitigating the effect of this approximation later in §5.1. In addition to restricting messages to those that can be plausibly sent by a microservice model, we also impose additional message restrictions based on analyzing the entire application. In the example web forum application, one such constraint is that the kv-store should receive no `modify` or `delete` requests. Discovering such additional constraints is a standard step in verifying invariants using rely-guarantees.

**Is this form of enforcement feasible?** Our preliminary investigations seem to show yes – with 64-byte packets, Bess can forward upwards of 15 million packet per second to a container, this drops to approximately 2.5 million packets per second when the Linux stack is used. Efficient key-value stores (which we use to benchmark peak service performance) such as Redis can only process between 100k and a 1 million operations per second [21], where each request and response fits within one packet. As a result we observe that the application is the bottleneck, and additional network processing should not drastically reduce performance. In our tests we observed little or no performance degradation for Redis, even when a 100 cycles of latency was imposed on each packet. While 100 cycles might not be sufficient for all enforcement tasks, we believe that these results indicate that in many cases enforcement can be performed with no degradation in performance.

**How to respond to invariant violations?** When an invariant violation is detected, `ucheck` drops the request and logs the incident. Not that safely dropping a request requires that we reset the TCP connection between a pair of microservices, and we assume that the RPC layer is resilient to such disconnections. Furthermore, our module can be configured to make an RPC request whenever a violation is received, and this mechanism can be used when debugging an invariant violation, as described next.

**Why implement at the network layer?** An alternative approach we could have adopted would be to implement this enforcement mechanism in the RPC library. However, in this scenario any memory corruption – due to bugs or exploits – can impact enforcement. On the other hand virtual switches are generally isolated from the microservices, and do not run this risk.

**Challenges due to encryption:** Our enforcement mechanism assumes access to message contents, an assumption which is violated when encrypted channels, *e.g.,* ones built using TLS, are used. This is a limitation of our approach,

and while it can be addressed by placing ucheck's mechanisms at a higher layer this fundamentally changes the design presented here. We note however that at present most microservices do not make use of such encrypted channels, and in general the overheads associated with encryption and decryption make it challenging to use such channels in systems consisting of many small services, and as a result we do not believe this poses a tremendous barrier to production deployments of ucheck. More generally, analyzing system behavior when control or data flow is encrypted remains an open problem, both in the case of microservices and single machine applications.

**ucheck's impact on placement:** Our enforcement mechanism requires no coordination between microservices and we do not require microservices to be connected to the same vSwitch instance (as long as they are connected to a vSwitch). As a result we impose no restrictions on container placement, or resource allocation.

# 4 DEBUGGING VIOLATIONS

How should application writers respond to invariant violations? While tools such as X-Trace [8], Dapper [23], etc. can be used to analyze logs and discover causal relations (which are roughly analogous to stack traces in sequential code) in microservice applications, this is often insufficient to debug problems. We observe that we can use our enforcement mechanism, along with the additional metadata used by X-Trace and Dapper to provide live debugging support that is roughly analogous to that provided by tools like GDB and LLDB. In this section we present mechanisms that allow application to step through distributed RPC calls and to set breakpoints. Other debugging mechanisms can be implemented similarly.

**Breakpoints**: A common way to use a debugger is to set a breakpoint, which is triggered when certain conditions are met and pauses a particular thread of execution. A breakpoint might pause program execution when control reaches a certain line of code (*i.e.,* the program counter reaches a specified value), on variable access, on exceptions, etc. In sequential programs breakpoints are implemented by adding instructions that are run whenever these conditions are met, *e.g.,* a breakpoint might be inserted by replacing instructions at particular location with an interrupt exception. We use a similar strategy to implement breakpoints in ucheck. We extend the ucheck enforcement module (§3.2) to accept a series of rules in addition to plausible messages – each rule is a predicate that identifies a set of messages. The enforcement module raises an exception whenever a message matches a rule. When a debugger is connected, the enforcement module notifies it of any exceptions (through an RPC call) – this notification includes the message that triggered the exception. Given this mechanism, ucheck can insert breakpoints by adding appropriate rules to all vswitches, and reporting to the user whenever an exception is triggered.

**Stepping:** After a breakpoint is reached (*i.e.,* the program has been broken into) it is often useful to execute individual statements and observe program state after each statement is executed. In the context of distributed applications we would like to allow developers to step through the processing of a *single external request*, while allowing other requests to be processed unmodified. Such functionality is useful for two reasons: (a) it allows the debugger to be used in production and (b) many distributed systems depend on keep-alives (or heartbeats) to detect failures, and delaying these messages can change application behavior. The key challenge in implementing stepping at a per request level lies in associating each message (RPC call) with the external request that resulted in the call. We address this by requiring that application add enough metadata to map each message to a particular request. Such metadata is required by X-Trace and Dapper, and our debugger can easily reuse this metadata.

Combining the ability to set distributed breakpoints, and stepping with the ability to reconstruct causal behavior using X-Trace or Dapper therefore allows ucheck to function as a debugger. Note however that the ucheck debugger can only operate at the level of RPC messages, visibility into the state within a microservice requires the use of GDB or another traditional debugger.

# 5 DISCUSSION

## 5.1 Approximate Enforcement

Our enforcement mechanism is approximate, and can sometimes fail to report invariant violations. This is because our mechanism as described in §3.2 does not have access to the local state of any microservice. There can be mitigated in three ways (a) by providing mechanisms that can be used by the enforcement mechanism to access local state; (b) by having the access state use local postconditions to maintain a view of local state and (c) by augmenting messages to include enough information about the service's state. We reject the first mitigation strategy since it can negatively impact both the correctness and performance of an application, since such access requires synchronization between each microservice and ucheck's enforcement module. The second mitigation similarly imposes severe performance penalties, in particular it can require up to twice as much computation. The last mitigation seems the most promising: many existing and commonly used protocols (*e.g.,* Kerberos) already require embedding enough information in every RPC request to allow ucheck to check whether some local condition is met (*e.g.,* in Kerberos [25] whether a requester is authenticated) in every RPC request. Such checks impose limited additional overheads and require no additional synchronization. The challenge with this

approach is of course in ensuring that only a limited amount of local information is needed. We believe this is the case for many common invariants, but can make no guarantees about the size of this additional metadata in the general case. In the future we also plan to investigate whether it is feasible to determine the minimum amount of metadata required for accurate enforcement during verification. If such information can be derived during verification, we could potentially add such metadata either by modifying each microservice or through mechanisms implemented in the virtual network.

### 5.2 Provable Correctness vs Enforcement

Recently IronFleet [24], Verdi [30], and others have suggested techniques for writing provably correct distributed systems. This can be done in several ways, in the case of IronFleet this is done by having programmers provide both a TLA+ specification and code, and the verifying both that the TLA+ specification upholds all invariants and that the code meets the specification, while Verdi does this by requiring developers to provide mechanically checkable proofs of correctness (written in Coq) and generating code corresponding to this proof. Both of these works are motivated by the observation that correctly implementing distributed systems is hard, which is similar to our motivation. One might ask whether our techniques are useful when deploying systems whose implementation is provably correct?

We believe this is in fact the case – in particular adding new invariants to either IronFleet or Verdi might require generating new proofs, and hence changing the system. `ucheck` by contrast allows new invariants to be added, checked and enforced without requiring any changes to running services. Furthermore both Verdi and IronFleet rely on a few underlying assumptions about the network and system they run on, and `ucheck` can also be used to ensure that those assumptions hold for a deployment.

## 6 RELATED WORK

In §5.2 we have compared `ucheck` to distributed programming systems that provide provable correctness. Other work on testing and debugging distributed systems has focused on two aspects:

**Randomized testing** Quickcheck [5, 6], DeMI [22], etc. investigated using randomized testing (*e.g.,* fuzz testing) to discover invariant violations. These techniques require visibility into the local state of all processes, and are most useful during development. These tools are thus complimentary to `ucheck`– they help in the development process rather than in the deployment process.

**Reconstructing errors using logs** ShiViz [2], X-Trace [8], Dapper [23], etc. allow developers to combine multiple concurrent logs into a single causal log. Developers can then use this causal log to identify and debug bugs. These tools focus on post facto analysis while `ucheck` can detect invariant violations in real time. `ucheck` makes use of these systems for debugging (§4) and prior work has shown that log analysis might be better suited at identifying liveness issues including performance bugs [1].

## 7 CONCLUSION

Microservice based applications represent an increasingly common class of non-trivial distributed systems built from heterogeneous components. Checking and enforcing correctness for such systems is both crucial and hard. In this paper we demonstrated that by leveraging advances in formal methods and software packet processing one can provide efficient mechanisms that ensure correctness.

## ACKNOWLEDGMENTS

We would like to thank Shivaram Venkatraman for the many insightful discussions that helped improve this paper. Chris Rossbach, and Katerina Argyraki provided invaluable comments on the initial drafts of this paper. This work was supported in part by a grant from Intel Corporation, and by the European Research Council under the EU's Seventh Framework Program (FP7/2007–2013) through ERC Grant 321174-VSSC.

## REFERENCES

[1] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[2] Beschastnikh, I., Wang, P., Brun, Y., and Ernst, M. D. Debugging distributed systems. *ACM Queue 14* (2016), 50.

[3] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Commun. ACM 59* (2016), 50–57.

[4] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NUSMV: A New Symbolic Model Checker. *STTT 2* (2000), 410–425.

[5] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP* (2000).

[6] Claessen, K., Palka, M. H., Smallbone, N., Hughes, J., Svensson, H., Arts, T., and Wiger, U. T. Finding race conditions in erlang with quickcheck and pulse. In *ICFP* (2009).

[7] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50*, 5 (Sept. 2003).

[8] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[9] Gavran, I., Niksic, F., Kanade, A., Majumdar, R., and Vafeiadis, V. Rely/guarantee reasoning for asynchronous programs. In *CONCUR* (2015).

[10] GRPC: A high performance, open-source, universal RPC framework. https://grpc.io, retrieved 01/21/2017.

[11] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. Softnic: A software nic to augment hardware. In *Technical Report UCB/EECS-2015-155*. EECS Department, University of California, Berkeley, 2015.

[12] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S. T. V., AND ZILL, B. Ironfleet: proving practical distributed systems correct. In *SOSP* (2015).

[13] JAMSHED, M. A., MOON, Y., KIM, D., HAN, D., AND PARK, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *NSDI* (2017).

[14] JONES, C. B. Specification and design of (parallel) programs. In *IFIP Congress* (1983).

[15] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., INGRAM, P., JACKSON, E. J., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *NSDI* (2014).

[16] LINUX FOUNDATION. networking:bridge. https://wiki.linuxfoundation.org/networking/bridge, retrieved 01/22/2017.

[17] MARTINS, R., MANQUINHO, V. M., AND LYNCE, I. Community-based partitioning for maxsat solving. In *SAT* (2013).

[18] THE COQ DEVELOPMENT TEAM. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[19] PAXSON, V. Bro: A system for detecting network intruders in real-time. *Computer Networks 31* (1998), 2435–2463.

[20] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E. J., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *NSDI* (2015).

[21] REDIS.IO. How fast is Redis? https://redis.io/topics/benchmarks, retrieved 01/22/2017.

[22] SCOTT, C., PANDA, A., BRAJKOVIC, V., NECULA, G., KRISHNAMURTHY, A., AND SHENKER, S. Minimizing Faulty Executions of Distributed Systems. In *NSDI* (2016).

[23] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[24] SIGURBJARNARSON, H., BORNHOLT, J., TORLAK, E., AND WANG, X. Push-button verification of file systems via crash refinement. In *OSDI* (2016).

[25] STEINER, J. G., NEUMAN, C., AND SCHILLER, J. I. Kerberos: An authentication service for open network systems. In *USENIX Conference* (1988).

[26] Apache Thrift. https://thrift.apache.org/, retrieved 01/21/2017.

[27] TODD HOFF. Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. https://goo.gl/1MRvoT, retrieved 01/21/2017.

[28] TONY MAURO. Adopting Microservices at Netflix: Lessons for Architectural Design. https://goo.gl/DyrtvI, retrieved 01/21/2017.

[29] What is VPP? https://wiki.fd.io/view/VPP/What_is_VPP%3F, retreived 01/21/2017.

[30] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. E. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI* (2015).