Uncovering Distributed System Bugs during Testing (not Production!)

Pantazis Deligiannis[†], Matt McCutchen[◇], Paul Thomson[†], Shuo Chen[⋆]
Alastair F. Donaldson[†], John Erickson[‡], Cheng Huang[⋆], Akash Lal[⋆]
Rashmi Mudduluru[⋆], Shaz Qadeer[⋆], Wolfram Schulte[‡]

[†]*Imperial College London,* [‡]*Microsoft,* [⋆]*Microsoft Research*
[◇]*Massachusetts Institute of Technology*

Abstract

Testing distributed systems is very challenging due to multiple sources of nondeterminism, such as arbitrary interleavings between event handlers and unexpected node failures. Stress testing, commonly used in industry today, is unable to deal with this kind of nondeterminism, which results in the most tricky bugs being missed during testing and only getting exposed in production.

We present a new methodology for systematically testing distributed systems. Our approach involves the use of P#, an extension of C# that combines a flexible environmental modeling approach with a concurrency testing framework, which can capture and systematically explore all sources of nondeterminism Shuo: this paper mainly focuses on the nondet due to scheduling, right? There are other sources, e.g., arbitrary input, system call behavior, clock, random number, etc. I feel that "all sources" is perhaps an overclaim Pantazis: We should clarify: as long as you model a source of nondet using P#, then you can capture it and control it (not automatically). We present two case studies of using P# to test production distributed systems inside Microsoft. Using P#, we managed to uncover a very subtle bug that was haunting developers for a long time, as they did not have an effective way to reproduce it. P# uncovered the bug in a very small setting, which made it easy to examine traces, identify and fix the problem. Pantazis: we now have many bugs in Matt's system too.

1 Introduction

Distributed systems are notoriously hard to design, implement and test [3, 12, 18]. This is due to many well-known sources of *nondeterminism* [4], such as race conditions in the asynchronous interaction between system components, the use of multithreaded code inside a component, unexpected node failures, unreliable communication channels and data losses, and interaction with (human) clients. All these sources of nondeterminism translate into *exponentially* many execution paths that a distributed system might potentially execute. A bug might hide deep inside one of these paths and only manifest under extremely rare corner cases [9, 20], but the consequence can be catastrophic [1, 26].

Classic techniques that product groups employ to test, verify and validate their systems (such as code reviews, unit testing, stress testing and fault injection) are unable to capture and control all the aforementioned sources of nondeterminism Shuo: my comment is similar to the one in the intro: do we claim that we capture all sources of nondet? Pantazis: only the ones that the user has mocked using the P# APIs, some are handled automatically like async/await, we should probably rephrase, which causes the most tricky bugs being missed during testing and only getting exposed after a system has been put in production. Discovering and fixing bugs in production, though, is bad for business as it can cost a lot of money [24] Pantazis: this citation is the famous multi-billion cost for the US economy, not sure if it should go here as we are focusing in distr systems and many dissatisfied customers [1, 26].

We interviewed engineers from the Microsoft Azure team regarding the top problems in distributed system development, and the consensus was that the most critical problem today is how to improve *testing coverage* to find bugs *before* a system goes in production. The need for better testing techniques is not specific to Microsoft; other companies, such as Amazon and Google, publicly acknowledge [21] that testing methodologies have to improve to be able to reason about the correctness of increasingly more complex distributed systems. Pantazis: We are missing a link for Google, but would be good if we can provide such a link, or for other companies like Facebook, I will try find something, but if anyone has a good suggestion please include

Amazon recently published an article [21] that describes their use of TLA+ [15] to detect distributed system bugs and prevent them from reaching production. TLA+ is a powerful specification language for verifying distributed protocols, but it is unable to verify the code that is actually being executed. The implied assumption is that a model of the system will be verified, and then the programmers are responsible to match what was verified with the source code of the real system. Although many design bugs can be caught with this approach, there is *no guarantee* that the real distributed system will be free of bugs.Shuo: we want to avoid making a general point that TLA+ has no guarantee to find all bugs, because our approach cannot either. Instead, we should say something like "as we will show in the paper, many bugs we found are too subtle to be discovered without examining concrete implementations." Pantazis: We should rephrase. By the way the main contrast with TLA+ is that TLA+ works *only* on a model and not the real code, whereas P# works on a real code (with a modeled environment).

In this work, our goal is to *test what is being executed*. We present a new methodology for testing legacy distributed systems and uncovering bugs before these systems are released in the wild. We achieve this using P# [5], an extension of the mainstream language C# that provides two key capabilities: (i) a flexible way of modeling the environment using simple language features; and (ii) a systematic concurrency testing framework that is able to capture and take control of all the nondeterminism in a real system (together with its modeled environment) and systematically explore execution paths to discover bugs. Shuo: my comment is similar to the one in the intro: do we claim that we capture all sources of nondet?

We present three case studies of using P# to test production distributed systems for Windows Azure inside Microsoft: a distributed storage management system and a live migration protocol. Using P#, we managed to uncover a very subtle bug that was haunting developers for a long time as they did not have an effective way to reproduce the bug and nail down the culprit. P# uncovered this bug in a very small setting, which made it easy to examine traces, identify and eventually fix the problem. We show that bugs in production do not need a large setting to be reproduced; they can be reproduced in small settings with easy to understand traces.

To summarize, our contributions are as follows:

- We present a methodology that allows flexible modeling of the environment of a distributed system using simple language mechanisms.
- Our infrastructure can test production code written in C#, which is a mainstream language.
- We present three case studies of using P# to test production distributed systems, finding bugs that could not be found with traditional testing techniques.
- We show that we can reproduce bugs in production systems in a small setting with easy to understand traces.

Shuo: let me give my overall comments about the current version of the paper here. This set of case studies are really cool. The fact that our approach is proven to be effective clearly is the primary selling point of the paper. Reviewers will love the results we have.
About the writing, there may be 3 areas to improve: (1) the word "we" is overloaded, because "we" play multiple roles in this work. "We" built P#; "we", as developers, designed the TableMigration system; "we", as testers, also take the responsibility to test the legacy Azure vNext system. The current writing doesn't clearly differentiate these roles, so readers may wonder how much work does it requires if they want to adopt the approach. (2) The intro makes me feel that we compare with existing logic reasoning approaches such as TLA+, and point out the limitation of TLA+. Until much later in the paper, I realize that our approach doesn't have any logic reasoning ingredient in it. I suggest that the intro makes it clear that our approach is complementary to TLA+, not a competing approach. We should avoid give a comparison with TLA+, because it is not an apple-to-apple comparison. (3) Section 4 is the most important section about the approach. I feel that we spend too much text on section 4.3, which is about building the test harness. My concern is that readers' minds may be dominated by how we mocked all kinds of objects and override real methods by fake methods. They may feel that, in the end, we tested a highly simplified system. This is not true, because the extent manager is a REALLY complicated system (which is shown as a small box in fig.3). I suggest that we zoom into this system to explain why many things can go wrong and it is hard to test in a conventional manner. Pantazis: Really nice suggestions. I totally agree with the last point you make and we should make sure this is addressed (it is super important)!

2 Motivating Example

Microsoft Azure Storage is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data. It has grown from 10s of Petabytes (PB) in 2010 to Exabytes (EB) in 2015, with the total number of objects stored well-exceeding 60 trillion.

Azure Storage vNext is the next generation storage system for Microsoft Azure, where the primary design target is to increase the scalability by more than 100×. Similar to the current system, vNext employs containers, called *extents*, to store data. Extents are typically several Gigabytes each, consisting of many data blocks, and

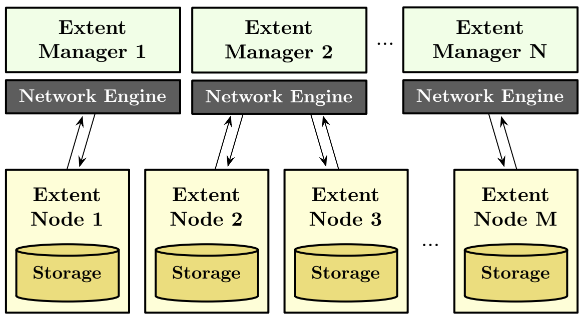

Figure 1: Top-level components of a distributed extent management system for Windows Azure.

replicated over multiple *extent nodes* (ENs). However, in contrast to the current system, which employs a Paxos-based, centralized mapping from extents to ENs, vNext achieves its scalability target by employing a completely *distributed mapping*. In vNext, extents are divided into partitions, with each partition managed by a light-weight *extent manager* (ExtMgr).

One of the many responsibilities of an ExtMgr is to ensure that every extent maintains enough *replicas* in the system. To achieve this, an ExtMgr receives frequent periodic *heartbeat* messages from every EN. The failure of an EN is detected by missing heartbeats. An ExtMgr also receives less frequent, but still periodic, *synchronization reports* from every EN. The sync reports list all the extents (and associated metadata) stored on the EN. Based on these two types of messages, an ExtMgr identifies which ENs have failed and which extents are affected and missing replicas. The ExtMgr, then, schedules tasks to repair the affected extents and distributes the tasks to ENs. The ENs repair the extents from their existing replicas in the system and lazily update the ExtMgr via future sync reports. All this communication between an ExtMgr and the ENs occurs via network engines installed in each component of vNext (see Figure 1).

To ensure correctness, the developers of vNext have instrumented extensive, multiple levels of testing:

1. *Unit testing*, which sends emulated heartbeats and sync reports to an ExtMgr and verifies that the messages are processed as expected.

2. *Integration testing*, which launches an ExtMgr together with multiple ENs, subsequently injects an EN failure, and finally verifies that the affected extents are eventually repaired.

3. *Stress testing*, which launches an ExtMgr with multiple ENs and multiple extents. It keeps repeating the following process: injecting an EN failure, launching a new EN and verifying that the affected extents are eventually repaired.

Despite of the extensive testing efforts, the vNext developers have been plagued by what appears to be an elusive bug in the ExtMgr logic. All the unit test and integration test suites successfully pass every single time. However, the stress test suite fails *from time to time* after very long executions, manifested as the replicas of some extents remain missing while never being repaired. The bug appears difficult to identify, reproduce and troubleshoot. First, it takes very long executions to trigger. Second, an extent not being repaired is *not* a property that can be easily verified. In practice, the developers rely on very large time-out period to detect the bug. Finally, by the time that the bug is detected, very long execution traces have been collected, which makes manual inspection tedious and ineffective.

To uncover this bug and many other similar ones, the developers are in constant search of a generic and systematic approach for testing distributed storage systems.

3 Testing Distributed Systems

Distributed systems typically consist of two or more components that communicate *asynchronously* by sending and receiving messages through a network layer [14]. Each component has its own input message queue, and when a message arrives, the component responds by executing an appropriate *message handler*. Such a handler consists of a sequence of program statements that might update the internal state of the component, send a message to another component in the system, or even create an entirely new component.

In a distributed system, message handlers can interleave in arbitrary order, because of the asynchronous nature of message-based communication. To complicate matters further, unexpected failures are the norm in production systems: nodes in a cluster might fail at any moment, and thus programmers have to implement sophisticated mechanisms that can deal with these failures and recover the state of the system. Moreover, with multicore machines having become a commodity, individual components of a distributed system are commonly implemented using multithreaded code, which adds another source of nondeterminism.

All the above sources of nondeterminism (as well as nondeterminism due to timeouts, message losses and client requests) can easily create *heisenbugs* [9, 20], which are corner-case bugs that are difficult to detect, diagnose and fix, without using advanced *asynchrony-aware* testing techniques. Techniques such as unit testing, integration testing and stress testing are heavily used in industry today for finding bugs in production code. However, these techniques are not effective for testing distributed systems, as they are not able to capture and control the many sources of nondeterminism.

The ideal testing technique should be able to work on unmodified distributed systems, capture and control all possible sources of nondeterminism, systematically inject faults in the right places, and explore all feasible execution paths. However, this is easier said than done when testing production systems.

3.1 Types of bugs

We can classify most distributed system bugs in two categories: *safety* and *liveness* property violations [13].

Safety A safety property checks that an erroneous program state is *never* reached, and is satisfied if it *always* holds in each possible program execution.

Liveness A liveness property checks that some progress *will* happen, and is satisfied if it *always eventually* holds in each possible program execution.

A safety property can be specified using an *assertion* that fails if the property gets violated in some program state. An example of a generic safety property for message passing systems is to assert that whenever a message gets dequeued there must be an action that can handle the received message.

Liveness properties are much harder to specify and check since they apply over entire program executions and not just individual program states. The liveness property that must be always eventually satisfied in the Azure Storage vNext system is that a user-defined *N* number of extent nodes must be always eventually available with the latest extent. There was an actual bug in the system, that led this property to fail for some very rare executions. Although this buggy behavior would be observed from time to time during stress testing, there was no way to reproduce the bug until we tested vNext using the P# systematic testing engine.

3.2 The P# framework

Our goal in this work is to *test what is being executed*. To achieve this we use P# [5], a framework that provides: (i) an *event-driven asynchronous programming* language for developing and modeling distributed systems; and (ii) a *systematic concurrency testing* engine that can systematically explore all interleavings between asynchronous event handlers, as well as other nondeterministic events such as failures and timeouts.

The P# language is an extension of C#, built on top of Microsoft's Roslyn[1] compiler, that enables asynchronous programming using communicating state-machines. P# machines can interact asynchronously by sending and receiving events,[2] an approach commonly used to develop distributed systems. This programming model is similar to actor-based approaches provided by other asynchronous programming languages (e.g. Scala [22] and Erlang [27]).

A P# machine consists of an input event queue, states, state transitions, event handlers, fields and methods. Machines run concurrently with each other, each executing an event handling loop that dequeues an event from the input queue and handles it by invoking an appropriate event handler. This handler might update a field, create a new machine, or send an event to another machine. In P#, a send operation is non-blocking; the message is simply enqueued into the input queue of the target machine, and it is up to the operating system scheduler to decide when to dequeue an event and handle it. All this functionality is provided in a lightweight runtime library, build on top of Microsoft's Task Parallel Library [16].

Because P# is built on top of C#, the programmer can blend P# and C# code; this not only lowers the overhead of learning a new language, but also allows P# to easily integrate with legacy code. Another advantage is that the programmer can use the familiar programming and debugging environment of Visual Studio.

A key capability of the P# runtime is that it can run in *bug-finding mode*, where an embedded systematic testing engine captures and takes control of all sources of nondeterminism (such as event handler interleavings, failures, and client requests) in a P# program, and then systematically explores all possible executions to discover bugs.

P# is available as open-source[3] and is currently used by various teams in Microsoft to develop and test distributed protocols and systems.

3.3 Overview of our approach

In previous work [5], we approached the problem of testing legacy distributed systems as follows. First, we ported the system to P#, then we modeled its environment as P# state machines, and finally we tested the ported system and its environmental model using the P# systematic concurrency testing engine. The limitation of this approach is that it does not allow us to directly test a legacy system, as it has to be re-implemented first in P#. However, such endeavor is very costly and time consuming, and thus is not realistic for testing an existing production system, such as the Azure Storage vNext. Also, unless the code under test is the one that will actually execute, there is no guarantee that the real system will be bug-free.

To solve this problem, and allow P# to be used for testing legacy distributed systems, we decided to take a different approach. Our approach of testing existing dis-

[1] `https://github.com/dotnet/roslyn`

[2] We use the word "event" and "message" interchangeably.

[3] `https://github.com/p-org/PSharp`

tributed systems using P# requires the developer to perform three key modeling tasks:

1. P# operates in the level of communicating state machines, and thus the environment of the system-under-test must be modeled using P# machines, while the real components of the system must be wrapped inside P# machines. We call this modeled environment the P# test harness.

2. The top level asynchrony due to message passing, must be exposed as event sending using the P# APIs. If the communication layer is not based on message passing, then additional effort must be spent into refactoring the system to use message passing. This step allows the P# runtime to capture and control the nondeterminism due to message passing interleavings during systematic testing.

3. Any other source of nondeterminism in the system or the environment (e.g. failures and timers), must be explicitly modeled using the available P# APIs. This would allow P# to capture the nondeterminism and control it during systematic testing.

In the remaining of this section, we will use the Azure Storage vNext case study as a running example of how to model a typical distributed system using P#. In principle, this modeling methodology is not specific to P# but can be used in combination with any systematic testing tool. We decided to use P# as it is a mature tool that provides a lot of modeling power via its C# language extensions and has an embedded systematic concurrency testing engine inside its runtime.

We argue that our approach is *flexible* since it allows the user to model *as much* or *as little* of the environment as required to achieve the desired level of testing. We also argue that our approach is *generic* since a programmer can build on top of it to test other distributed systems besides vNext (see Section 6). Furthermore, the language features that are required to be used to connect the real code with the modeled code, are already being heavily used in production for testing purposes (e.g. *virtual method dispatch*), which significantly lowers the bar for product groups to embrace P# for testing.

In Section 5 we will present more specifics of how the P# bug-finding runtime works and extensions that we did since the original work [5] to be able to use P# to systematically test production code.

4 Testing Azure Storage vNext with P#

To uncover the elusive extent repair bug in Azure Storage vNext, a testing harness is written in P#. The developers of vNext expect the bug more likely in Extent Manager than in Extent Node. Hence, the testing harness focuses on the real Extent Manager with mocked Extent Nodes.

The testing harness consists of multiple P# machines.

1. There is a *Extent Manager* machine that is a thin wrapper for the real Extent Manager and several *Extent Node* machines that represent simple mocked ENs.

2. In addition, there is a *Testing Driver* machine that communicates with all the machines, relays messages between Extent Manager and EN machines, and drives testing scenarios. In one testing scenario, the testing driver launches Extent Manager with three ENs, with a single extent on one of the ENs. It then waits for the extent to be repaired on all three ENs. In another testing scenario, the testing driver fails one of the ENs and launches a new EN. It then waits for the extent to be repaired on all the remaining ENs.

3. Moreover, there is a *Monitor* machine, which collects states from all the ENs and verifies the desired property that the extent is *eventually* repaired.

4.1 Extent Manager Machine

The testing target is the real Extent Manager in vNext, wrapped inside a P# machine in the testing harness.

4.1.1 Extent Manager Internals

Inside Extent Manager, there are two structures related to extent repair: an *Extent Center* and an EN map. The Extent Center maintains mapping records from extents to their hosting ENs. It is updated upon every sync report from ENs. The EN map records the latest heartbeat time from every EN. It internally runs an EN expiration loop that is responsible for removing ENs that have been missing heartbeats for extended period and also cleaning up corresponding Extent Center records. In addition, Extent Manager runs an extent repair loop, which examines all the Extent Center records, identifies extents with missing replica, schedules extent repair tasks and sends the repair tasks to ENs.

4.1.2 Stubbing Network

Extent Manager uses a network engine to send and receive messages from ENs. The testing harness mocks the original network engine in vNext and overrides the networking methods. In this way, it intercepts all outbound messages and relays to the testing driver, which is responsible for dispatching the messages to destination ENs. As shown in Figure 3, the mocked network

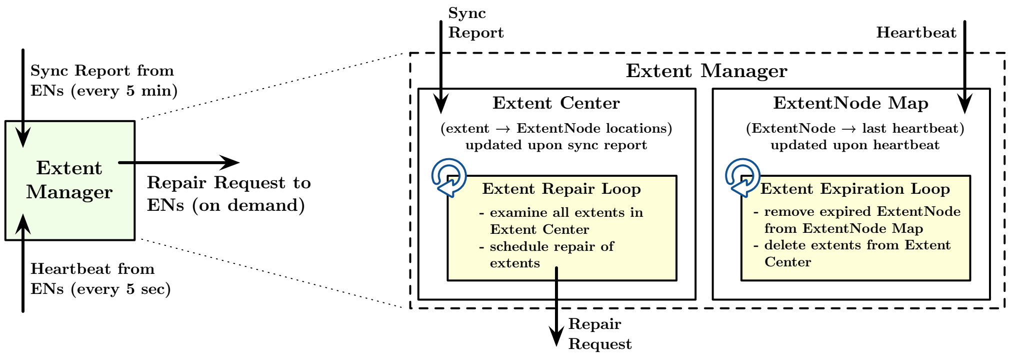

Figure 2: Internal components of the Extent Manager in Azure Storage vNext.

```
// network interface in vNext codebase
class NetworkEngine {
  public virtual void SendMessage(Socket s, Message msg);
}

// mocked network for capturing Extent Manager messages
class MockedNetEngine : NetworkEngine {
  public override void SendMessage(Socket s, Message msg) {
    // capture and relay Extent Manager messages
    PSharpRuntime.Send(this.Environment, new
          MessageFromExtentManagerEvent(), s, msg);
  }
}
```

Figure 3: Mocked Network Engine.

engine captures outbound messages from Extent Manager and invokes `PSharpRuntime.Send(...)` to asynchronously relay the messages to the testing driver.

4.1.3 Extent Manager Machine

Figure 4 shows code snippet from Extent Manager machine, as a thin wrapper of the real Extent Manager. A mocked network engine replaces the real one in Extent Manager. It captures and relays all outbound messages from Extent Manager.

Incoming messages from ENs are delivered to the Extent Manager machine and trigger action. The specific action here relays the messages to the internal Extent Manager by invoking `extMgr.ProcessSingleMessage`. From the Extent Manager's perspective, it is unaware of EN mocking and behaves as if it is communicating with several real ENs.

4.2 Mocking Extent Node

The `ExtentNode` machine is a model (less than 150 lines of code) of the real Extent Node and is responsible for generating synchronization events and sending them to the `ExtentManager` to test the extent manager's repair logic.

briefly describe mocked repair logic.

```
// Wrapping the target vNext component in P#
class ExtentManagerMachine : Machine {
  private ExtentManager extMgr; // real vNext code

  void Init()
  {
    extMgr = new ExtentManager();

    extMgr.netEngine = new MockedNetEngine(); // mock network
    extMgr.SkippingJournal = true; // skip persistence
    extMgr.IsMockingTimer = true; // disable internal timer
  }

  [OnEventDoAction(typeof(MessageFromExtentNodeEvent),
       nameof(DeliverMessageAction))]

  private void DeliverMessageAction()
  {
    var msg = (CommandMessage)this.Payload;
    // relay messages from Extent Node to Extent Manager
    extMgr.ProcessSingleMessage(msg);
  }
}
```

Figure 4: Wrapping vNext Component in P#.

briefly describe mocked sync logic

4.3 Test Harness

The environment of a distributed system might consist of other distributed systems and services, clients, operating system timers, as well as libraries for networking or other purposes. To be able to systematically test a distributed system, this environment must be modeled and all the interactions between the environment and the system, as well as all the nondeterminism, must be captured and controlled by the P# runtime.

4.3.1 Orchestrating Machines

Before being able to use P# to test an existing distributed system, the environment of the components that the developer wants to test have to be modeled using the P# state machine APIs. This involves creating mock classes that inherit from the `Machine` abstract class. The

```
// Mocking Extent Node in P#
class ExtentNodeMachine : Machine {
  // use real vNext component whenever appropriate
  private ExtentNode.ExtentCenter extCtr;

  [OnEventDoAction(typeof(RepairExtentRequestEvent),
        nameof(ProcessRepairExtentRequest))]
  [OnEventDoAction(typeof(TimerTickEvent),
        nameof(ProcessExtentNodeSync))]

  // extent repair logic
  void ProcessRepairExtentRequest()
  {
    var source = GetRepairSourceMachine(this.Payload);
    PSharpRuntime.Send(source, new
          DownloadExtentRequestEvent(), this.Payload);
  }
  ...

  // extent node sync logic
  void ProcessExtentNodeSync()
  {
    var sync = new ExtentNodeSync(this.NodeID);
    extCtr.Extents.ForEach(ext =>
      {
        sync.ExtentRecords.Add(new ExtentRecord(ext));
      });
    PSharpRuntime.Send(this.ExtentManagerMachine, new
          MessageFromExtentNodeEvent(), sync);
  }
}
```

Figure 5: Mocked Extent Node.

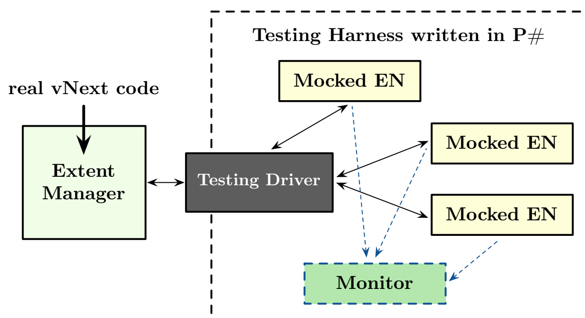

Figure 6: The real environment of the Extent Manager is replaced with a mocked version for testing.

`Machine` class exposes methods for declaring a state machine (e.g. machine states and state transitions) as well as methods for sending events to other machines, declaring event handlers for received events, and accessing the payload from the latest received event.

In the Azure Storage vNext case study, we created the following P# machines: `Environment`, `ExtentNode`, `ExtentManager`, and `Timer`.

The `Environment` (denoted with a dashed line box in Figure 6) is essentially a "god" machine: it is the very first machine that is created; has a handle to all other machines in the harness; and includes the logic based on which the test harness will execute. The programmer is free to create either a finite test harness (e.g. with a predefined number of node failures) or an infinite test harness (e.g. with an unbounded number of failures), as long as any nondeterminism (e.g. when to insert a failure) is captured using the available P# APIs.

Finally, the `Timer` machine models the operating system timer and is responsible for triggering events related to the Azure Storage vNext logic (e.g. synchronization messages, heartbeats and repair logic loops). This is discussed in detail in Section 4.3.4.

```
// Real code for detecting node expiration
public virtual bool IsNodeExpired(string id, DateTime time)
{
  return DateTime.Compare(time, DateTime.Now) <= 0;
}

// Mocked code for detecting node expiration
public override bool IsNodeExpired(string id, DateTime time)
{
  return this.DeletedNodes.Contains(id);
}
```

Figure 7: Abstracting the node expiration logic in the Extent Manager component of Azure Storage vNext.

4.3.2 Modeling and injecting failures

In production, each Extent Node of the Azure Storage vNext system periodically (every 5 seconds) sends a heartbeat to the Extent Manager which notifies that the Extent Node is alive. Because we want to model failures and systematically inject them using P#, we abstract away the heartbeat mechanism in our harness. However, the Extent Manager logic relies on time intervals to detect node failures (see Figure 7). The way to abstract this time-related logic and connect the real code with the modeled code is to use virtual dispatch and override the virtual `IsNodeExpired` method with a mocked version.

Figure 7 presents how we mocked the node expiration detection method. Instead of comparing the time interval as in the original code, we now check if the set `DeletedNodes` contains the id of the Extent Node that we are checking for expiration. If it contains the id, then it means that the node has failed. The `Environment` machine that we have created as part of our testing P# harness, will nondeterministically choose a node to kill, then send the id of this killed node to the Extent Manager wrapper machine, who will in turn add it to the `DeletedNodes` set.

4.3.3 Entry point to a P# test harness

A developer can specify the entry point of a P# test harness similar to how unit tests are typically written. Figure 8 shows the source code of the entry point for the Azure Storage vNext P# test harness. The programmer can declare an entry point method using the `Microsoft.PSharp.Test` attribute. When the P# sys-

```
[Microsoft.PSharp.Test]
public static void Execute()
{
  PSharpRuntime.CreateMachine(typeof(Environment));
}
```

Figure 8: Static C# method acting as the entry point to the Azure Storage vNext P# test harness.

tematic testing engine is invoked, it will scan the binary and attempt to find a static method declared with this attribute. When such a method is found (in our example the `Execute()` method), the testing engine will invoke it and start testing the program. The systematic engine can optionally run multiple iterations of the test harness, each one potentially exploring a different schedule. In each iteration, the program state will be reset (any static fields must be explicitly reset by the programmer) and the entry point will be re-invoked.

4.3.4 Abstracting timers

Distributed systems are often using timers to determine when an event should be send from one component to another. For example, in the Azure Storage vNext system, each Extent Node is associated with a timer that fires of a synchronization message every 5 minutes and a heartbeat every 5 seconds. This timer is related to the liveness bug that we discovered: the synchronization message that gets fired every 5 minutes can potentially race with an Extent Node failure; if it arrives after the node failed, then the bug would manifest. Traditional testing techniques cannot easily find such a bug, due to the very infrequent occurrence of this race due to the timer.

Our methodology in P# to systematically test distributed systems that rely on timers, is to abstract timers away, model them using message passing communication and introduce nondeterminism in their firing. This nondeterminism is introduced using the P# `Nondet()` method, which returns a nondeterministic boolean value that is controlled by the P# runtime during systematic testing.

Figure 9 shows how we modeled a generic timer in the Azure Storage vNext case study. The Extend Manager, as well as each Extent Node in the harness, is associated with a unique `Timer` machine. When creating this machine, we pass as a payload the id of the machine that owns this timer. When the `Timer` machine is created, it stores this id in the `Owner` field and then transitions to the `Active` state. In this state, the `Timer` loops infinitely and nondeterministically (using `Nondet()`) sends a `TimerTickEvent` to `this.Owner`. When the Extent Node owner receives this event, it handles it by generating a synchronization message that is being send to the Extent Manager. Similarly, when the

```
internal class Timer : Machine
{
  MachineId Owner; // Id of the owner machine

  [Start]
  [OnEntry(nameof(InitOnEntryAction))]
  [OnEventGotoState(typeof(Unit), typeof(Active))]
  class Init : MachineState { }

  void InitOnEntryAction()
  {
    this.Owner = (MachineId)this.Payload;
    // triggers state transition to Active
    this.Raise(new Unit());
  }

  [OnEntry(nameof(ProcessTickEvent))]
  [OnEventGotoState(typeof(Unit), typeof(Active))]
  class Active : MachineState { }

  void ProcessTickEvent()
  {
    // Nondeterministic boolean choice controlled by P#
    if (this.Nondet())
      // sends a timer tick event to the owner machine
      this.Send(this.Owner, new TimerTickEvent());
    // triggers state transition to Active
    this.Raise(new Unit());
  }
}
```

Figure 9: Timers in Azure Storage vNext are modeled as nondeterministic P# machines.

Extent Manager receives a `TimerTickEvent` from its own `Timer`, it handles it by nondeterministically invoking repair-related methods in the Extent Repair Center data structure.

5 The P# Systematic Testing Engine

The P# runtime is a lightweight layer build on top of the Task Parallel Library (TPL) of .NET that implements the semantics of P#: creating state machines, executing them concurrently using the default task scheduler of TPL, and sending events and enqueueing them in the appropriate machines. A key capability of the P# runtime is that it can execute in bug-finding mode for systematically testing a P# program to find bugs, such as assertion violations and uncaught exceptions. We now give an overview of how this works, while more details can be found in the original paper [5].

When the P# runtime runs in bug-finding mode, an embedded *systematic testing engine* captures and takes control of all sources of non-determinism that are *known* to the P# runtime. The engine attempts to detect asynchronous bugs by systematically scheduling machines to execute their event handlers in a different order. This approach is based on systematic concurrency testing [8, 20, 7] (SCT) techniques that have been previously developed for testing shared memory programs.

The systematic testing engine serializes the program execution, takes control of the synchronization and nondeterministic choice points, and at each *step* of the execution it invokes a systematically chosen P# machine to execute its next event handler. The engine will repeatedly execute a program from start to completion, each time exploring a potentially different set of interleavings, until it either reaches a bound (in number of iterations or time), or it hits an assertion failure. The testing is fully automatic, has no false-positives (assuming an accurate environmental model), and can reproduce found bugs by replaying buggy schedules.

We have implemented two schedulers inside the P# systematic testing engine: *random* and *probabilistic concurrency testing* (PCT) [2]. The random scheduler takes a completely random decision at every scheduling point, whereas the PCT scheduler uses randomization in a disciplined fashion to provide a probabilistic guarantee of finding a bug. It is straightforward to create a new scheduler by implementing the `ISchedulingStrategy` interface exposed by the P# libraries. The interface exposes callbacks that are invoked by the P# runtime for taking decisions regarding which machine to schedule next, and can be used for developing both generic and application-specific schedulers, although we have experimented only with generic schedulers so far. Exposing an easy-to-use interface for creating new schedulers is inspired by previous work [6].

We designed the systematic testing engine in a way that enables easy debugging: after a bug is found, the engine can generate a trace that represents the buggy schedule. Note that this trace (in contrast to typical logs generating during production) is sequential.

5.1 Checking safety specifications

Safety property specifications can be encoded in P# by using the provided `Assert(...)` method. This method takes as argument a predicate, which if evaluates to false denotes a safety property violation. The programmer is also free to use other custom assertion APIs (as P# executes actual code), but to enable P# to recognize such APIs, the suggested approach is to override them to call the P# `Assert(...)` method.

P# also provides a way to specify global assertions by using *monitors*, special state machines that can only receive events, but not send. To declare a monitor, the programmer has to inherit from the P# `Monitor` class. A `Machine` does not need to have a reference to a `Monitor` to send an event to it; as long as a `Monitor` has been created, any `Machine` can invoke it by calling `Monitor<M>(...)`, where `M` is the identifier name of the `Monitor`, and the parameter is an event and an optional payload. This call is also synchronous, in contrast to the regular asynchronous `Send(...)` method calls, as the monitor has to be called deterministically (and not be interleaved).

5.2 Checking liveness specifications

Liveness property specifications are in principle harder to encode since they apply over entire program executions instead of individual program states. Normally, liveness checking requires the identification of an infinite fair execution that never satisfies the liveness property [23, 19]. Prior work [23] has proposed that assuming a program with finite state space, a liveness property can be converted into a safety property. Other researchers proposed the use of heuristics and only exploring finite executions of an infinite state space system using random walks to identify if a liveness property is violated [11].

The P# developer can write liveness properties using a *liveness monitor*, which is a special type of monitor that can contain three types of states: regular ones; *hot* states; and *cold* states. A state annotated with the `Hot` attribute denotes a state where the liveness property is not satisfied (e.g. a node has failed but a new one has not come up yet). A state annotated with the `Cold` attribute denotes a state where the liveness property is satisfied.

One can imagine a liveness monitor as a *thermometer*: as the program executes and the liveness property is not satisfied, the liveness monitor stays in the hot state, which infinitely raises the temperature. If the liveness property is ever satisfied, and thus the liveness monitor transitions to a cold state, the temperature is instantly reduced to normal levels. Encoding liveness properties using hot and cold states enables the programmer to specify arbitrary LTL properties. The liveness monitors in P# are based on previous work [**?**].Shuo: at this point, I realize that our work (perhaps P#) doesn't have any symbolic reasoning ingredient in it, right? so the comparison with TLA+ is not an apple-to-apple comparison.

Pantazis: Discuss the liveness checking algorithm: are we going to mention the terminating-harness random walk one? or the infinite-harness lasso one with partial caching?

Pantazis: Show how we specified the liveness property in vNext?

5.3 Handling intra-machine concurrency

Vanilla P# is able to capture and take control of the *inter-machine concurrency* due to message passing, but is unable to systematically explore any interleavings due to *intra-machine concurrency* (e.g. async/await or TPL). This is problematic as nowadays, with multicore machines being a commodity, programmers tend to write multithreaded code to exploit shared memory architectures and increase the performance of individual components of a distributed system.

As an example, the Live Azure Migration system uses the `async` and `await` C# 5.0 language primitives. Asynchronous code using async/await is more readable because it looks like traditional procedural code, but it is translated by the compiler to an event-driven state machine that is built on top of TPL to achieve performance. The key idea behind async/await is that a method declared as `async` can use internally the `await` keyword which allows the thread executing the method to wait on a TPL task, or any other *awaitable* object, *without* blocking. This is achieved as follows. When an `await` statement is executed, the code following the await is wrapped as a TPL task *continuation* and the method returns to the caller. This continuation executes when the awaitable object has completed.

Developing a fully automatic and universal approach to handle intra-machine concurrency in .NET is very challenging, because there are many APIs that can be used for concurrency and synchronization (e.g. `System.Threading`, TPL, async/await, locks, semaphores) and each one has its own complexities. Although developers are willing to model the top-level message passing communication using P#, they are resistant in modeling the low-level threading and synchronization methods as this would be a very invasive procedure and such refactoring is unlikely to scale for legacy code.

To test our case studies, we decided to focus our efforts in handling async/await as providing a robust solution for a subset of the .NET threading APIs is more feasible than trying to handle arbitrary threading and synchronization. Our solution involves using a custom task scheduler whenever the bug-finding mode of the P# runtime is enabled, instead of the default TPL task scheduler. The P# task scheduler inherits from the `TaskScheduler` class, a low-level API that is responsible for enqueueing TPL tasks into threads.

Our approach works as follows. We start the task of the *root* P# machine in our custom task scheduler. As long as any child tasks spawned by the root machine task do not explicitly start in another scheduler (including the default TPL scheduler), then they will be scheduled for execution in our custom task scheduler. Our custom scheduler intercepts the call to enqueue a task, creates a special machine called `TaskMachine` and *wraps* the enqueued task inside this machine. The constructor of a `TaskMachine` takes as an argument a TPL task and stores it in an field. The `TaskMachine` has only a single state and when it is scheduled for execution by the P# systematic testing scheduler it will start executing the wrapped task and then immediately wait on its completion. This forces the lifetime of the task to become the lifetime of the machine and vice versa. Thus, when the P# systematic testing scheduler schedules the `TaskMachine`, it will also schedule the corresponding task. This means that when a P# program makes a call to an `async` method, then the task created in the backend will be automatically wrapped and scheduled.

For `await` statements, no special treatment is required because `await` statements are compiled into a continuation. The task associated with the continuation will also be wrapped and scheduled accordingly.

Regarding support for handling non-async/await concurrency primitives, we are currently investigate feasible approaches.

6 Experience Report

6.1 Azure Storage vNext

We used P# to test the *distributed extent management* component of the Windows Azure vNext distributed storage system. This component is responsible for managing the partitioned extent metadata and works as follows.

vNext bug goes here.

6.2 Live Azure Table Migration

The Live Azure Table Migration is a library capable of transparently migrating a data set between tables in the Windows Azure storage service while an application is accessing the data set. MigratingTable provides a *virtual table* with an API similar to that of an ordinary Azure table, backed by a pair of *old* and *new* tables. A background *migrator* job moves all data from the old table to the new table. Meanwhile, each read or write issued to the virtual table is translated to a sequence of reads and writes on the backend tables according to a protocol we designed, which guarantees linearizability of operations on the virtual table across multiple application processes assuming that the backend tables respect their own linearizability guarantees.

The initial motivation for MigratingTable was to solve a scaling problem for Artifact Services, an internal Microsoft system with a data set that is sharded across tables in different Azure storage accounts because it exceeds the limit on traffic supported by a single Azure storage account. As the traffic continues to grow over time, the system needs to reshard the data set across a greater number of Azure storage accounts without interrupting service. During such a resharding, our sharding manager will identify each key range that should migrate to a different table, and we will use a separate MigratingTable instance for each such key range to actually perform the migration (Figure 10). MigratingTable may also be useful to migrate data to a table with different values of configuration parameters that Azure does not support

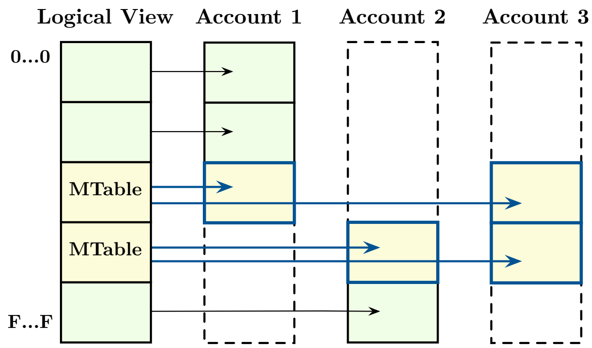

Figure 10: Resharding a data set when a third Azure storage account is added. Two key ranges are each migrated to the new account using a MigratingTable instance (abbreviated MTable).

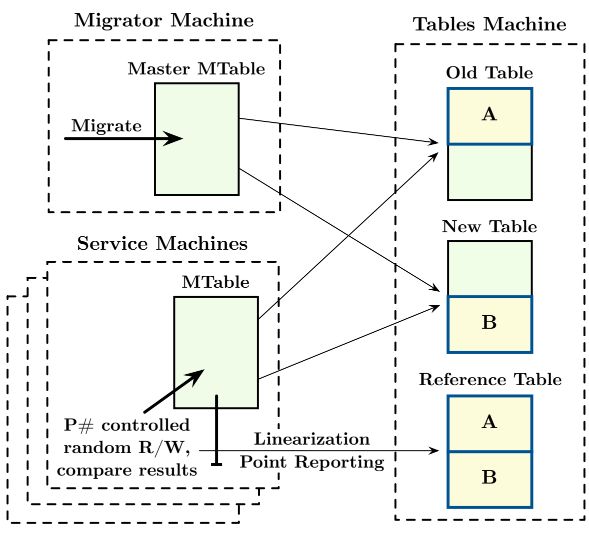

Figure 11: MigratingTable P# correctness test environment.

changing on an existing table, such as geographic location.

We also used P# to test the MigratingTable library, which is capable of transparently migrating a data set between tables in the Windows Azure storage service while an application is accessing the data set.

Since we were designing a new concurrent protocol that we expected to become increasingly complex over time as we add optimizations, we planned from the beginning to maintain a P# test harness along with the protocol to maintain confidence in its correctness.

MigratingTable implements an interface called IChainTable2, which provides the core read and write functionality of the original Azure table API with one exception: it provides *streaming reads* with a weaker consistency property than multi-page reads in the original API, since the original property would have been difficult to achieve for no benefit to applications we could foresee. MigratingTable requires that its backend tables also implement IChainTable2, and we wrote a simple adapter to expose physical Azure tables as IChainTable2. Our goal was then to verify that when multiple application processes issue "input" read and write calls to their own MigratingTable instances with the same backend tables, the behavior complies with the specification of IChainTable2 for the combined input history.

6.2.1 Input generation

Of course, there are many possible input histories to test. Since we originally hoped for a comprehensive form of verification and we didn't feel we had good intuition to prepare a set of specific test cases that would make us confident of having caught any and all concurrency bugs in the protocol, it was natural for us to sample from a distribution of input histories that we defined to exercise all the features of IChainTable2 within certain bounds. Furthermore, it was natural to let P# control the choice of input history as well as the machine interleaving so we could reproduce both using a single random seed.

All of our input histories include two application processes. Each process performs either a single streaming read or a sequence of two atomic calls, each a read or a batch write. Each batch write call includes one or two operations, where the operation type is chosen from the set supported by IChainTable2 (Insert, Replace, Merge, Delete, InsertOrReplace, InsertOrMerge, DeleteIfExists) and the row key is chosen from $\{0,\ldots,5\}$. If the operation requires an If-Match value, it is equally likely to be `*`, the current ETag of the row (if it exists), or some non-matching value. Finally, the new entity includes a user-defined property `isHappy` whose value is equally likely to be true or false. For both atomic and streaming reads, the filter expression is equally likely to be empty (i.e., match everything), `isHappy eq true`, or `isHappy eq false`.

6.2.2 Model structure

To comprehensively verify the behavior of MigratingTable under an arbitrary input history, we needed some formulation of the IChainTable2 specification to which to compare it. Since the specification is deterministic under sequential calls except for the results of streaming reads, we decided the easiest approach was to write an in-memory reference implementation called SpecTable. Given a streaming read call, SpecTable can produce a set of all results that are compliant with the specification. Our correctness property is then:

> For every execution trace of a collection of MigratingTables backed by the same pair of *old* and *new* SpecTables (where P# chooses the actual result of each streaming read from the valid set) in parallel with the migrator job, there exists a linearization of the combined input history such that the output in the original trace matches the output of a "reference" SpecTable on the linearized input.

We instrumented MigratingTable to report the intended *linearization point* of each input call, which in our setting is always one of the corresponding *backend calls* to the backend tables (often the last). Specifically, after each backend call completes, MigratingTable reports whether that call was the linearization point, which may depend on the result of the call. This makes it possible to verify the correctness property as the model executes. The model consists of a P# *tables machine* containing all three SpecTables; a collection of *service machines* containing identically configured MigratingTables; and a *migrator machine* that performs the background migration (Figure 11). Each service machine issues a random sequence of input calls to its MigratingTable, which sends backend calls to the tables machine. When MigratingTable reports the linearization point of an input call, the service machine sends that input call to the reference table. When an input call completes, the service machine checks that the results from the MigratingTable and the reference table agree. P# controls the interleaving of the backend calls. To ensure that the reference table is never observed to be out of sync with the backend tables, after the tables machine processes a backend call, it enters a state that defers further backend calls until MigratingTable has reported whether the backend call was a linearization point and (if so) the call to the reference table has been made. We use the P# random scheduling strategy; we were afraid that an exhaustive strategy would only be feasible within bounds so low that we would miss some bugs.

We wanted to implement the core MigratingTable algorithms in C# "async/await" code, like most of Artifact Services, to achieve both good readability and good performance. We used a method similar to that described in Section 5.3 to bring the generated TPL tasks under the control of the P# scheduler. Then we implemented an "async" RPC mechanism based on the .NET RealProxy class that automates the generation of proxies for objects hosted by other P# machines (in our setting, the service machines use proxies for the SpecTables and various auxiliary objects hosted by the tables machine). When a machine calls a method on a proxy, the proxy sends a P# message to the host machine, causing it to execute the method call on the original object and send back the result, which the proxy then returns. Thus, the use of these proxies as IChainTable2 backends is transparent to the MigratingTable library, thanks to dynamic dispatch.

6.3 An example bug in MigratingTable

Matt: Move wherever appropriate

One of the bugs in MigratingTable that we found using the P# test stands out because it reflects the type of oversight that tends to occur as designs evolve and it's unclear whether we would have been able to find it by any other method (Matt: revise remark when we have stress test results?). This bug, which we named QueryStreamedBackUpNewStream, is in the implementation of a streaming read from the virtual table, which should return a stream of all rows in the table sorted by key. The essential implementation idea is to start streams s_O, s_N from the old and new backend tables and merge the sorted streams by keeping track of the next row in each stream and returning the row with the lesser key. In parallel, the migrator job is concurrently copying rows from the old table to the new table; we had satisfied ourselves that this concurrency would not cause any problems. However, then we added support to the migrator job to delete the old table when it finishes copying, which triggers the virtual stream to close s_O. Suppose the virtual stream is in a state in which the next row in s_O has key k_O and the next row in s_N has key k_N, where $k_O < k_N$. Further suppose that before the next read from the virtual stream, the migrator job copies a row with key k ($k_O < k < k_N$) from the old table to the new table and then deletes the old table. Since s_O has not yet returned this row when it is closed and s_N has already advanced to k_N, the row with key k will be missed by the virtual stream. A similar problem can occur if s_N does not reflect rows inserted into the new table by the migrator job after s_N is started, as allowed by the IChainTable2 specification. Restarting s_N when the old table is deleted fixes both variants of the bug.

Matt: Add a figure based on migration-bug3-explanation.pptx (probably a hybrid of slides 4 and 5, assuming we want only one figure).

6.4 Azure Service Fabric (change to CScale?)

Pantazis: I guess we should write stuff about CScale here based on our last discussion: overview of the system (collection of Fabric services connected via TPL dataflow?), of its environment (Fabric), how they interop?, why its very challenging to test? how they test currently? I am not sure if all these should go here or be split here and the experience report, we will see ... same for the above case studies

Azure Service Faric (or *Fabric* for short) is a platform and API for creating reliable services that execute on a cluster of machines. The developer writes a service that receives requests (e.g. from some client program via HTTP requests) and mutates its state based on these requests. In order to make the service *reliable*, Fabric launches several copies (*replicas*) of the service, where each copy runs as a separate process on a different node in the cluster. One replica is selected to be the *primary* which serves client requests; the rest are *secondaries*. The primary replicates state changes to the secondaries by sending *replication requests* so that all replicas eventually have the same state. If the primary fails (e.g. if the node on which the primary is running crashes), Fabric elects one of the secondaries to be the new primary and launches another secondary; the new secondary will receive a full or partial copy (depending on whether persistent storage is used) of the state of the new primary in order to "catch up" with the other secondaries. Fabric provides a name-resolution service so that clients can always find the current primary.

Fabric services have a lot of asynchrony, which make them interesting targets for systematic testing with P#. Our primary goal was to create a P# model of Fabric to allow thorough testing of services, where Fabric's asynchrony is controlled by the P# runtime. The system that we wished to test is *CScale* [**?**], a big data-stream processing system that chains multiple Fabric services in a directed acyclic graph. Prior work [5] created a model of Fabric with limited functionality; it used a mixture of C# and P# internally, only supported one in-flight replication request (which restricts the asynchrony that can be tested), and only supported one Fabric service. Our new Fabric model was re-written to use only P# internally, support an arbitrary number of Fabric services and in-flight replication requests, and in general be a more complete model of Fabric. Note that C# code is still required to interface with the existing C# service code. We refer to our C# code as the *translation layer* and the user-written C# service code as *user code*.

An overview of our Fabric model is shown in Figure 12. The `ClusterRuntime` machine handles the creation and management of one or more Fabric services, as well as service resolution requests which allows for client-service and inter-service communication within the model. Each Fabric service instance is managed by a `ServiceRuntime` machine, which in turn manages several `ReplicaRuntime` machines. Each `ReplicaRuntime` communicates with the user code via several machines and interfaces from the translation layer (only the translation layer for the primary is shown, but every `ReplicaRuntime` has its own instance of the translation layer). Note that communication between machines is hierachical; thus, communication between

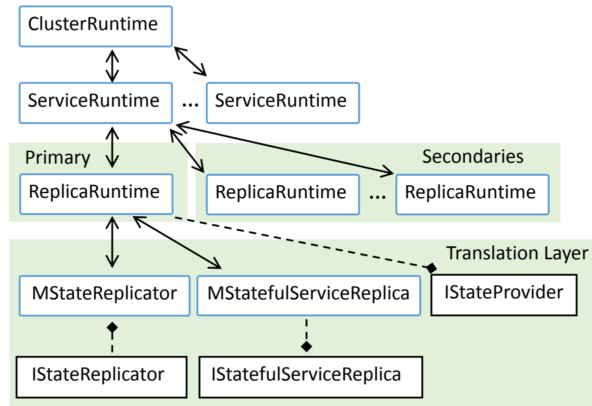

Figure 12: Overview of the key machines (rounded boxes) and interfaces (boxes) in our Fabric model.

`ReplicaRuntimes` (such as the sending of replication requests) is via the `ServiceRuntime` machine for that service. This approach does not necessarily reflect how Fabric works in practice. Instead, we chose an architecture that keeps the model simple while still allowing (what we believe to be) realistic asynchrony and failure scenarios.

7 Evaluation

We compared the bug-finding effectiveness of P# systematic concurrency testing to traditional "stress testing" without systematic concurrency control on several benchmark bugs, described below. We report the length of time it took for the test to fail and remark on the effort needed to diagnose the bug from the test output. Matt: Overview of actual results?

7.1 Experimental Setup

We performed all experiments using the Windows PowerShell tool on a 2.50GHz Intel Core i5-4300U CPU with 8GB RAM running Windows 10 Pro 64-bit.

Because we fixed all bugs that we discovered using the P# systematic testing engine in the case studies, for the evaluation we added an option to conditionally reintroduce each one of the bugs so that we can measure how fast and how often each bug can be discovered with the P# random and the PCT scheduler.

Cheng's case study with the actual bug

After fixing all the bugs we found in MigratingTable, we added an option to conditionally reintroduce each of the following bugs:

1. Filtering on a user-defined property did not retrieve non-matching rows from the new table that might shadow a matching row from the old table.

Azure Systems	#LoC Real	#LoC Model	P# statistics #M	P# statistics #ST	P# statistics #AB	#Bugs found
Azure Storage vNext	0	684	5	11	17	1
Live Table Migration	0	0	0	0	0	>10
Fabric User Service	0	0	0	0	0	0

Table 1: Statistics.

2. ...

7.2 Systematic testing with P#

Table 2 presents the results from running the P# systematic testing engine on each case study with an enabled bug using the random and the PCT schedulers. We configured the engine to perform 100,000 iterations. The random seed for both schedulers was generated in each iteration using the `DateTime.Now.Millisecond` API which represents the current time in milliseconds. The PCT scheduler was configured with a bug depth of 2 and a max number of scheduling steps to execute of 500. All reported times are in seconds.

How long you had to run the test to find the bug (if you can find it) and how long are the traces

How easy it is to pinpoint the error using the P# traces, comparing to traditional stress testing

Effort required to use the system

Controlled random scheduling has proven to be efficient for finding concurrency bugs [25, 5].

8 Related Work

A significant amount of research has been conducted on how to analyze and test distributed systems [15, 23, 11, 10, 29], but a lot of these techniques either have significant limitations, or cannot be easily applied in a production environment, due to many complexities that are outside the scope of a research project.

A completely different approach for reasoning about the correctness of distributed systems is to use formal methods. A notable example is TLA+ [15], a formal specification language that can be used to design and verify concurrent programs via model checking. Amazon recently published an article describing their use of TLA+ in Amazon Web Services to verify distributed protocols [21]. A limitation of TLA+, as well as other similar specification languages, is that they are applied on a model of the system and not the actual system. Even if the model is verified, the gap between a real-world implementation and the verified model is still significant, so implementation bugs are still a realistic concern.

In Verdi [28], a distributed system is written and verified in Coq, and then OCaml code is produced for execution. Verdi cannot find liveness bugs. P# is also more production-friendly (works on a mainstream language). Pantazis: say few more stuff

D3S [17] Pantazis: should read the paper, they say they are debugging running systems

9 Conclusion

Draft.

References

[1] AMAZON. Summary of the AWS service event in the US East Region. http://aws.amazon.com/message/67457/, 2012.

[2] BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010), ACM, pp. 167–178.

[3] CAVAGE, M. There's just no getting around it: you're building a distributed system. *ACM Queue 11*, 4 (2013), 30–41.

[4] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[5] DELIGIANNIS, P., DONALDSON, A. F., KETEMA, J., LAL, A., AND THOMSON, P. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 154–164.

[6] DESAI, A., QADEER, S., AND SESHIA, S. Systematic testing of asynchronous reactive systems. Tech. Rep. MSR-TR-2015-25, Microsoft Research, 2015.

[7] EMMI, M., QADEER, S., AND RAKAMARIĆ, Z. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2011), ACM, pp. 411–422.

[8] GODEFROID, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (1997), ACM, pp. 174–186.

[9] GRAY, J. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems* (1986), IEEE, pp. 3–12.

[10] GUPTA, D., VISHWANATH, K. V., AND VAHDAT, A. DieCast: Testing distributed systems with an accurate scale model. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2008), USENIX, pp. 407–421.

[11] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (2007), USENIX, pp. 18–18.

[12] LAGUNA, I., AHN, D. H., DE SUPINSKI, B. R., GAMBLIN, T., LEE, G. L., SCHULZ, M., BAGCHI, S., KULKARNI, M., ZHOU, B., CHEN, Z., AND QIN, F. Debugging high-performance computing applications at massive scales. *Communications of the ACM 58*, 9 (2015), 72–81.

		P# Systematic Testing (Random)				P# Systematic Testing (PCT)			
CS	**Bug Identifier**	**Time to Bug (s)**	**#SP**	**%Buggy**	**BF?**	**Time to Bug (s)**	**#SP**	**%Buggy**	**BF?**
1	ExtentNodeLivenessViolation			x%	✓			x%	✓
2	QueryAtomicFilterShadowing	157.22	165	0.018%	✓	350.46	108	0.220%	✓
2	QueryStreamedLock	2121.45	181	0.001%	✓	6.58	220	0.664%	✓
2	QueryStreamedBackUpNewStream	-	-	-	✗	5.95	232	0.458%	✓
2	DeleteNoLeaveTombstonesEtag	-	-	-	✗	4.69	272	1.187%	✓
2	DeletePrimaryKey	2.72	168	3.895%	✓	2.37	171	4.137%	✓
2	EnsurePartitionSwitchedFromPopulated	25.17	85	0.076%	✓	1.57	136	1.491%	✓
2	TombstoneOutputETag	8.25	305	0.370%	✓	3.40	242	0.473%	✓
2	DeleteIfExistsNotLinearizable	-	-	-	✗	3.19	242	0.195%	✓
◇2	QueryStreamedFilterShadowing	0.55	79	24.908%	✓	0.41	79	24.998%	✓
◇2	MigrateSkipPreferOld	-	-	-	✗	1.13	115	25.043%	✓
◇2	MigrateSkipUseNewWithTombstones	-	-	-	✗	1.16	120	x%	✓
◇2	InsertBehindMigrator	0.32	47	72.534%	✓	0.31	47	x%	✓

Table 2: Results from running the P# random and PCT systematic testing schedulers for 100,000 iterations. We report: time in seconds to find a bug (Time to Bug); number of scheduling points when a bug was found (#SP); percentage of buggy iterations (%Buggy); and if a bug was found with a particular scheduler (Bug Found?).

[13] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (1977), 125–143.

[14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[15] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (1994), 872–923.

[16] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (2009), ACM, pp. 227–242.

[17] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), USENIX, pp. 423–437.

[18] MADDOX, P. Testing a distributed system. *ACM Queue 13*, 7 (2015), 10–15.

[19] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), ACM, pp. 362–371.

[20] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 267–280.

[21] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How amazon web services uses formal methods. *Communications of the ACM 58*, 4 (2015), 66–73.

[22] ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala*. Artima Inc, 2008.

[23] SCHUPPAN, V., AND BIERE, A. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer 5*, 2-3 (2004), 185–204.

[24] TASSEY, G. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3* (2002).

[25] THOMSON, P., DONALDSON, A. F., AND BETTS, A. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), ACM, pp. 15–28.

[26] TREYNOR, B. GoogleBlog – Today's outage for several Google services. `http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html`, 2014.

[27] VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall International, 1996.

[28] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 357–368.

[29] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 213–228.