A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications

Jonathan Goldstein (jongold), Ahmed Abdelhamid (samy)[†], Mike Barnett (mbarnett),
Sebastian Burckhardt (sburckha), Badrish Chandramouli (badrishc), Darren Gehring (darrenge),
Niel Lebeck (nl35)[ʃ], Christopher Meiklejohn (cmeiklej)[Γ], Umar Farooq Minhas (ufminhas),
Ryan Newton (rrnewton)[χ], Rahee Ghosh Peshawaria (raghosh), Tal Zaccai (talzacc), Irene Zhang (irzha)
Microsoft (@microsoft.com), Purdue University (@purdue.edu)[†], University of Washington (@cs.washington.edu)[ʃ],
Carnegie Mellon University (@andrew.cmu.edu)[Γ], Indiana University (@indiana.edu)[χ]

ABSTRACT

When writing today's distributed programs, which frequently span both devices and cloud services, programmers are faced with complex decisions and coding tasks around coping with failure, especially when these distributed components are stateful. If their application can be cast as pure data processing, they benefit from the past 40-50 years of work from the database community, which has shown how declarative database systems can completely isolate the developer from the possibility of failure in a performant manner. Unfortunately, while there have been some attempts at bringing similar functionality into the more general distributed programming space, a compelling general-purpose system must handle non-determinism, be performant, support a variety of machine types with varying resiliency goals, and be language agnostic, allowing distributed components written in different languages to communicate. This paper introduces Ambrosia, the first system to satisfy all these requirements. We coin the term "virtual resiliency", analogous to virtual memory, for the platform feature which allows failure oblivious code to run in a failure resilient manner. We also introduce novel programming language constructs for resiliently handling non-determinism. Of further interest is the effective reapplication of much database performance optimization technology to make Ambrosia more performant than many of today's non-resilient cloud solutions.

PVLDB Reference Format:
Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, Irene Zhang. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *PVLDB*, 13(5) : 588-601, 2019.
DOI: https://doi.org/10.14778/3377369.3377370

INTRODUCTION

When writing today's distributed programs, which span both devices and cloud, programmers are faced with complex decisions and coding tasks around coping with failure, especially when applications are stateful: Consider an application consisting of two objects, Client and Server, where Server keeps a counter, initially 0, and exposes a method called Inc() to increment the counter and return the new value. Furthermore, assume Client calls Inc() twice and prints the value of the counter after each call. If both objects are run in a single process, the outcome is clear: the values 1, and 2 are displayed in Client output. In contrast, consider the possibilities when Client and Server run on different machines, where state is maintained locally, and method calls are performed through an RPC (remote procedure call) mechanism.

First let's consider possible outcomes when Client fails and is naively restarted from scratch and reconnected: If Client fails after the first call and after the return value is received, the output will instead be 2, 3, which is incorrect. If Client fails after successfully issuing the RPC request, but before receiving the return value, Server will initially try to provide to Client an unexpected return value, which is problematic. Even worse, consider that Client may be restarted on a different machine, with a different IP address.

Outcomes when Server fails are further complicated by the loss, and subsequent reinitialization of the counter. If Server fails after Client has completed the first RPC, the output will be 1, 1, which is incorrect. Furthermore, if Server fails after receiving the first RPC request, but before communicating the return value, Client is left waiting for a return value which never arrives.

In order to get the answer consistent with no failures occurring, developers face varying challenges, depending on the type of application they are writing.

If a task is pure data processing, it benefits from the past 40-50 years of work from the database community, which has shown how declarative database systems, which produce deterministically replayable behavior through logging, along with technology to make database sessions robust ([1], [30]) can completely isolate the developer from the possibility of failure in a performant manner. Most recently, map-reduce and its progeny ([2], [3], [42]), by pursuing similar strategies, have achieved similar results.

Unfortunately, while there have been attempts at bringing some of these capabilities to general purpose distributed programming, frequently called "exactly once execution" ([1], [30], [4]), the failure to address a number of important issues (details below) has prevented their widespread use. As a result, developers either give up entirely on fully reliable applications, or implement solutions that involve complex, error-prone, and difficult to administer strategies to make applications reliable in today's cloud environments (Section 2). A compelling general-purpose solution to this problem must address the following:

- **Virtual Resiliency** - In this paper, we coin the term virtual resiliency, which provides developers the illusion that machines never fail, by automatically fully healing the system after physical failure, analogous to how virtual memory provides developers the illusion that physical memory never runs out by automatically paging memory to disk. While most data processing platforms already provide efficient programming and execution environments

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377370

with virtual resiliency, there are no commonly used analogous systems for general purpose distributed programming.

- **Non-determinism** – Most distributed applications contain non-determinism, like generating timestamps, or collecting user input. Reliable systems must handle sources of non-determinism gracefully, providing virtual resiliency in the face of such challenges. In this paper, we introduce *impulses*, a novel platform feature for handling non-determinism. Database logging provides some hints for handling these situations, capturing non-deterministic choices in the replay log before committing.
- **Performance/Cost** - Any general-purpose implementation of virtual resiliency, must have performance comparable to failure-sensitive code with a good application specific strategy. Only data processing systems have achieved this today.
- **Machine Heterogeneity** - While machines inside a datacenter can be homogenous, distributed apps typically span devices and datacenters. Additionally, some devices may be heavy and able to persist information necessary to hide failure while others may be best effort. The end-to-end semantics must be easy to understand, reason about, and code against. Today, [4] is the closest to achieving this goal.
- **Language Heterogeneity** - Because distributed applications span across a variety of machines and settings, distributed components written in different languages must be able to work together. Architecture (e.g. .NET DataContract [25]) and language independent serialization formats (e.g. Protobuf [26], Avro [27], JSON [28]) effectively solve this problem.

In this paper, we present Ambrosia (Actor Model Based Reliable Object System for Internet Applications), the first general purpose distributed programming platform for non-deterministic applications, with virtual resiliency, high performance, and machine and language heterogeneity. Ambrosia is a real system, available on GitHub [29], and is used in a cloud service which manages the machine images of hundreds of thousands of machines running a cloud application [40].

Ambrosia's high performance was achieved by incorporating the decades' old wisdom used to build performant, reliable, and available database systems. For instance, we make extensive use of batching, high-performance log writing, high-performance serialization concepts, and group commit strategies.

Using the technology mentioned above, we implement virtual resiliency with only a 25% reduction in throughput for the worst case. We also achieve throughput comparable to gRPC, a popular RPC framework which lacks any kind of failure protection. Compared to gRPC, ping latency increases by only 5.5ms. We vastly outperform today's typical cloud-based, fully resilient designs, in some cases achieving about a 1000x improvement in cost per unit of work served, and with 1 to 3 orders of magnitude lower latency. In active/active configurations, typical Ambrosia failover times are less than 2 seconds, and, excluding service logic, recovery costs are roughly half the primary running costs, leading to generally low recovery times.

Because Ambrosia's virtual resiliency implementation is based on database style logging technology, we also offer familiar related features, like transparent high availability through active standbys. In addition, we also provide application centric features less familiar to databases, such as time-travel debugging [23], retroactive code testing, and inflight application upgrades.

Ambrosia's machine heterogeneity goes beyond allowing applications on different types of machines to communicate. For instance, .NET core applications written in Ambrosia can seamlessly recover from a Windows PC to a Raspberry Pi running Linux, without requiring help from developers.

While constructing Ambrosia, we learned a number of important lessons about modern performant system design. Most of these are based on the following observation: the cost of transferring and storing bytes is going down rapidly, while the cost of processing bytes is improving very slowly. Looking through historical trade magazines, we determined that until 7 years ago, network, storage, and CPU price/performance (i.e. p/p) were all improving comparably. Over the last 7 years, however, network and storage throughput p/p have both improved by about 10x and 7x respectively, while CPU p/p has only improved by about 1.5x. The future, with terabit networking and NVRAM persistent memory looks to hold more of the same. This encourages approaches and system designs, like Ambrosia, which exploit cheap network and storage bandwidth. By carefully optimizing CPU costs through adaptive batching and minimal byte interpretation, Ambrosia's design results in very low costs for the protection it affords.

Paper organization: Section 1 introduces a running example used throughout the paper, and a naive implementation, where we simply assume failure never happens. Section 2 describes how to implement our running example resiliently using standard cloud application building blocks. Section 3 describes the basic Ambrosia design and presents the Ambrosia implementation of our running example. Section 4 describes how externally originating non-determinism (e.g. user input) is handled by Ambrosia, and extends our running example to demonstrate. Section 5 describes important Ambrosia features enabled by its logging oriented approach towards resiliency. Section 6 contains an experimental evaluation which compares Ambrosia against the strategy described in Section 2, as well as a comparison to gRPC. Additionally, we test failover and recovery times. Sections 7, and 8 present related work, and conclusions and future work, respectively.

1. Running Example: Message Forwarding

Consider a message forwarding service which, every thousand messages received, reports the current and time elapsed since the last report. Further, assume that the forwarded and reporting messages go to different destinations, with eventual, but not consistent, freshness guarantees. Below is a naive C# implementation, where we assume failure never happens, and where services communicate with one another through RPC calls on proxies to other services, using a single threaded (i.e. one request at a time per actor) actor style like Orleans [16]:

```
public interface IForwarder {
  void Process(string userMessage);
}

class Forwarder : IForwarder, Actor {
  DateTime startTime;
  int count=0;
  IForwardToService forwardTo;
  IReportToService reportTo;

  public Forwarder(ServiceSet services) {
    forwardTo = GetProxy<IForwardTo>(“forwardTo”);
    reportTo = GetProxy<IReportTo>(“reportTo”);
  }

  void public Process(string userMessage) {
    if (count == 0) {
      startTime = DateTime.Now;
    }
    count++;
    if (count%1000 == 0) {
      long reportNum = count/1000;
      DateTime now = DateTime.Now;
      reportTo.Send(reportNum-1, now,
                    now-state.startTime);
      state.startTime = now;
    }
    forwardTo.Send(userMessage);
  } }
```

In the above example, we assume that an instance of the Forwarder class is instantiated in some program which hosts the code, and directs incoming requests to the Process method. This could be mostly generated given IForwarder, which specifies the signature of the Process method. The service could then be started by running the code, and providing a string which other services can use to connect to the running forwarder instance.

Connections are made to other services by creating proxies in the constructor. These proxies' methods exactly match the request types supported by the destination services. In this case, we can deliver messages to the forwardTo and reportTo services by calling Send methods on the created proxies. The interface types IForwardTo and IReportTo, like IForwarder, exactly define these request types as method calls. Names for the actual running services to connect to are given when creating the proxies (e.g. "forwardTo", and "reportTo"). Note that GetProxy is a service lookup method defined in the actor base class, and the passed strings are the names used when the corresponding running instances were created.

When a Send method is called, the arguments are automatically serialized and sent to the destination actor, where they are then deserialized and executed.

In the absence of failure, this code would sufficiently define the logic of our service: Forwarder contains all necessary state, the constructor establishes all necessary outgoing connections, and the Process method performs all necessary computation, state changes, and sends all required output. Observe that where the Forwarder runs is immaterial. It could be running at the edge, in the cloud, or even on the same machine as one of the other services. Ideally, an implementation of our forwarder in a system with virtual resiliency would look very similar to this code.

Note that we have neglected to address some important issues: do the Send method calls execute synchronously with execution on the called actors, or do they merely initiate the request and continue. Minimally, the send ordering must be respected w.r.t. each individual destination actor, but concurrent execution could be allowed across destinations. For performance reasons, we should, in this example, allow concurrency across destinations, since our problem clearly states that eventual consistency is sufficient. This may, however, not always be the case, and synchronous execution should be possible in a general-purpose framework. Also, aggressive batching of both request processing and communication is needed for high throughput [8]. How can this be accomplished?

2. DISTRIBUTED RESILIENCY TODAY

It is possible, without virtual resiliency, to build fully resilient distributed applications with today's cloud development tools, and today's practitioners do so when necessary. In this section, we explore what such implementations, without virtual resiliency, look like in the context of our running example. At times, developers choose less resilient strategies due to the implementation and deployment complexity, as well as performance challenges, most of which will be made apparent in this section.

Figure 1 shows a typical configuration for a cloud application today. In this particular example, a client, which may or may not be in the datacenter, first durably records its service request in some record sink, like Event Hub, Kafka, or Kinesis, ensuring that the request is preserved in replicated storage. The application logic is then expressed as an Azure/Lambda function and is called on batches of requests. Any output (e.g. to other services), is then sent to other durable queues, and the pattern is potentially repeated. The infrastructure guarantees that every function will run to completion exactly once on every input, although multiple failed attempts may be made before successful completion.

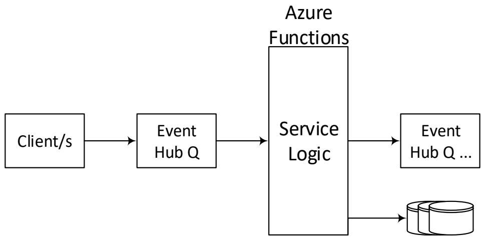

Figure 1: Resiliency using stateless compute

Since Azure and Lambda functions are stateless, to make the application resilient, every call must begin by retrieving all application state necessary to process the request, and write the state back after processing. But since the function may fail at any time during execution, and be retried many times before succeeding, we will need to add code to recover from partial executions when side-effects occur (e.g. communicating), or non-deterministic code is run (e.g. getting a timestamp).

While only one Azure function can be run at a time, in order, and still guarantee correct behavior, most applications naturally partition into independent identical pipelines, which may be run in parallel to achieve higher application throughput. Consequently, they store application state in key/value stores, keyed on the partition id, and present the requests in batches for each partition.

Consider our message forwarding service: to process a batch of messages (per source), the state for that partition is first loaded from storage. Then, the messages are processed, forwarding messages and sending a time reporting message every thousand user messages. Finally, the state is written back to storage. The application state type, message types, and initial values for relevant state fields follow:

```
class State {                 class ReportMessage {
  Id source;                    Id source;
  long count = 0;               long reportNum;
  long lastSeqNo = -1;          DateTime reportTime;
  DateTime startTime;           TimeSpan elapsedTime;
}                             }

class UserMessage {
  Id source;
  long seqNo;
  string message;
}
```

The following code assumes that when a message sink is asked to give the sequence number of the last event received, prior to receiving any events, -1 is returned. This code also assumes that sequence numbers start at 0 and are consecutive. Finally, when a message send is complete, it is assumed that message loss is no longer possible, and that message order is determined by the order in which they are sent.

```
void Process(Id source, List<UserMessage> batch,
    MessageSink ForwardTo, MessageSink ReportTo)
{
  State state = LoadState(source);
  long lastSent = ForwardTo.Last().seqNo;
  foreach(var m in batch) {
    if (m.seqNo > state.lastSeqNo) {
```

```
8        state.count++;
9        state.lastSeqNo++;
10       if (state.count%1000 == 1) {
11         if (count == 1) {
12           state.startTime = DateTime.Now;
13           SaveState(source);
14         }
15         else {
16           state.startTime =
17               ReportTo.Last().reportTime();
18         }
19       } else if (state.count%1000 == 0) {
20         long reportNum = count/1000;
21         if (reportNum > (ReportTo.Last().seqNo+1)) {
22           DateTime now = DateTime.Now;
23           ReportTo.Send(new UserMesage(source,
24               reportNum-1, now, now-state.startTime));
25         }
26       }
27     }
28     if (m.seqNo > lastSent)
29       ForwardTo.Send(m);
30   }
31   SaveState(source);
32 }
```

Note that great care has been taken in the above code to ensure correct behavior in the presence of partial executions:

- We increment the counter only if the message wasn't counted in the last SaveState, by checking lastSeqNo (lines 7-9).
- We save state when the first event ever is processed, since we need to set startTime for later computation of duration, and failure could occur on the first Process call (line 13).
- We check, before sending a ReportMessage, whether we've already sent that message in a previous execution (line 21).
- We put reportTime into ReportMessage, and then retrieve it for duration calculations, in order to ensure that durations are consistent with the wall clock passage of time (e.g. the sum of durations is equal to actual time elapsed), even in the presence of partial executions. The timestamp is, in fact, a source of non-determinism that if improperly handled, would produce nonsensical ReportMessages (lines 24, 10-19).
- Before forwarding a message, we check to see if a previous execution already forwarded the message by checking the sequence number. We are careful to forward at the end of the loop iteration since we don't want message forwarding to interfere with timestamp generation (line 28).

The subtle design issues discussed above illustrate some of the challenges associated with writing resilient code today without virtual resiliency. As we'll see in Section 6, there are also serious performance problems with these deployments in practice.

3. AMBROSIA DESIGN

3.1 Ambrosia's Approach and Architecture

Ambrosia's approach for implementing virtual resiliency is an evolution of past approaches for creating deterministic robust distributed systems ([1], [6], [30], and [33]).

In particular, these other systems advocate logging incoming requests, and using replay to recover the system to an equivalent state prior to failure. Some are even able to transparently reconnect after failure. For instance, Pheonix [1] can fully recover deterministic components and their connections, ensuring that failure does not change overall application state or behavior, even though the application writer's code is oblivious to the possibility of failure. We are now ready for a more precise definition of virtual resiliency:

Virtual Resiliency *– A distributed platform capability which enables developers to produce applications whose behavior, other than performance, are unaffected by failure, but where developers write failure oblivious code. This capability is supported through a combination of logging and language idioms which make the application deterministically replayable, as well as automatic reconnection protocols which ensure that disconnected/recovered components may reconnect and continue as if failure didn't occur.*

Note the use of the phrase "deterministically replayable". Transactional databases provide deterministic replayability, even though they have many sources of non-determinism, like thread scheduling. They are, however, deterministically replayable, which means that with the aid of the recovery log, they can recover to a state consistent with previous interactions.

Map-reduce systems, and their progeny, like Spark (excluding Spark Streaming), had virtual resiliency from their inception. Always, the capability relied on deterministic replay.

Unfortunately, none of the general purpose distributed platforms which provide virtual resiliency, like Pheonix, handle non-determinism, have designs which make Ambrosia's level of performance possible, or provide machine and language heterogeneity.

The following diagram illustrates the architectural components of two communicating Ambrosia services/objects/actors, called "immortals":

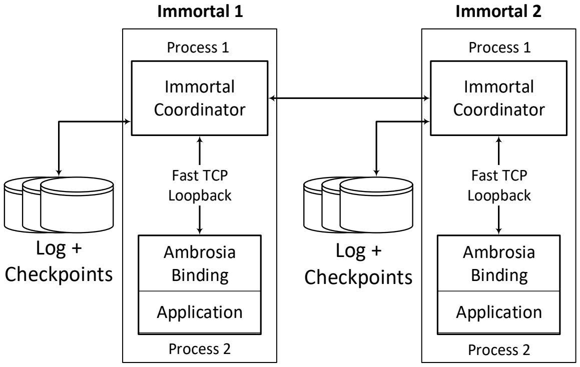

Figure 2: Resiliency using Ambrosia

Ambrosia is a peer to peer system. Any Ambrosia immortal can make RPC style requests of any immortal it's connected to, including itself, given a published API.

For recovering the state of a failed immortal, Ambrosia's approach is similar to previous work. In particular, all input requests are logged to replicated storage prior to execution, guaranteeing correct state reconstruction during replay-based recovery. Additionally, upon reconnection, like previous work, Ambrosia employs a protocol using internal sequence numbers to ensure deterministically ordered exactly once delivery of requests. Through an open-source process and communication virtualization layer called the Common Runtime for Applications (CRA) [20], successful reconnection happens even if an immortal comes up on a different machine. The end result is **virtually resilient**, an ecosystem which can fully self-heal without assistance from the immortal developer, where all failure is turned into waiting.

While the basic viability of Ambrosia's approach should be clear for deterministic immortals, there are important challenges which need to be overcome for this approach to be practical:

- How should Ambrosia handle non-deterministic immortals? Non-determinism can come from a variety of sources, including non-deterministic results like getting the current time, and accepting input from non-replayable sources, like user input.
- How do we make Ambrosia's approach performant?
- How do we achieve language and machine heterogeneity?

We begin with a deeper discussion of Ambrosia's architecture, shown in Figure 2, which addresses the issues of language and machine heterogeneity, and will frame our explanation of how we support C#.

First, note that each immortal is composed of two running processes, which are expected to run on the same container/VM /machine. The choice to run each immortal as two separate processes is an implementation convenience which allows us to more easily add support for multiple languages, at additional latency cost, but is not fundamental to Ambrosia's design.

We assume that the two processes that comprise each immortal share a failure domain. That is, if one process crashes, the other will also crash or will stop prior to any attempt at restarting. In clusters, a recommended deployment model uses Kubernetes pods, which correctly establish failure domains. Ambrosia relies on TCP for network connections; as a result, we assume messages are always delivered reliably and in order. Any failures at the network level are handled by the TCP protocol (e.g., missing packets). On Windows, Ambrosia uses fast TCP loopback, reducing the penalty for a two-process design.

Note that Ambrosia is intentionally careful to NOT make deployment decisions. For instance, Ambrosia's only responsibility is to run correctly in a properly deployed environment, and not pollute the log when infrastructure failure occurs. Being in the same failure domain, deployers are expected to ensure that both processes are brought down before restarting them on the same, or a different node. Whether to try again on the same node, or move to another node, is in the hands of the deployer, and is intentionally not Ambrosia's concern. In this manner, Ambrosia may be used in a maximally flexible way. Note that there are certain situations which will cause one or both of the processes to crash, like a primary losing the file write lock on the log, losing access to the log or metadata, or running out of memory. Deployers are expected to monitor the health of deployed immortals, and fully fail (and possibly restart) the instance when these error situations are encountered.

The first process is the immortal coordinator (IC), which handles all log interactions, and communication with other immortals. This process is also responsible for orchestrating immortal recovery, including both broken connections to other immortal coordinators. and handling failover when active secondaries are present.

The IC relies on CRA [20] as an application hosting layer that virtualizes the TCP connections between a graph of vertices, which are, in our case, Immortals. For instance, when an immortal fails and is restarted on another machine, after the state has been recovered through replay, all previously connected immortals are automatically reconnected by CRA to the restarted instance, going through an Ambrosia specific sequence number based reconnection protocol guaranteeing logically exactly once delivery.

The IC is blissfully unaware of types, or even the nature of requests passed between immortals. From the IC's point of view, messages, in the form of byte arrays, are passed along the connections that they share, are logged, and sent to a language binding. Any information about types, endianness, and even whether the message is a new request or return value, is immaterial to the IC. Furthermore, we've implemented the IC in both .NET Framework and .NET Core, enabling it to run on a wide variety of architectures and operating systems. There will be a more detailed discussion of the IC in Section 3.3, which will be revisited in Section 4.2 to describe how impulses are implemented.

The second process is divided into 2 parts: the first part is a language-specific Ambrosia binding, responsible for interacting with the IC. The IC sends messages to the language binding, which interprets those messages as new requests or return values associated with previous outgoing requests. The language binding then executes service logic in response to these messages, and sends to the IC any outgoing requests or return values in the form of new messages. We now state an overly strong language binding contract which guarantees virtual resiliency:

Strong Language Binding Contract: From some initial state, any execution of the same incoming requests in the same order must result in both an equivalent final state, as well as the same outgoing requests in the same order. In addition, the binding must also provide a state serializer.

Note that the above definition does not specify anything about threading, language idioms or style, or even if the language needs to be Turing complete. Rather, it only requires determinism w.r.t. the log. Also, to avoid replaying from the start of the service during recovery, the IC must occasionally checkpoint the state of the immortal, which includes the application state. The specific method of, and format for, serialization can vary from language to language, or even amongst bindings for the same language.

Machine heterogeneity in Ambrosia is achieved by the architecture described above. Since ICs pass untyped byte arrays amongst themselves, leaving it to the language binding layer to interpret these messages, machines with varying operating systems and architectures need only to agree on the serialization format of these messages to successfully communicate. Furthermore, our choice to implement .NET core versions of both our IC and C# language binding, combined with support for C#'s architecture independent binary data contract serialization [25], makes this capability available in Ambrosia today for a wide variety of architectures and operating systems.

Ambrosia today goes even further: a .NET core immortal running on a Windows PC is recoverable on a Raspberry Pi running Linux, including all the connections to other immortals. This is a consequence of .NET core applications running on a wide variety of platforms, and state serialization for both the C# language binding and IC depending exclusively on architecture independent serialization strategies.

Since the IC is serialization format oblivious, as long as two language bindings agree on an argument serialization format, like Avro, or Protobuf, they may successfully invoke each other's RPCs, achieving **language heterogeneity**.

At this point, it is interesting to point to a few important differences between Ambrosia's architecture, and the architecture described in Section 2:

- Because the log of requests is hidden in the IC, there is great flexibility in storing the log. For instance, the IC could store the log in a local file, or some form of cloud storage, depending on an application's needs. This decision may even be delayed until deployment time, with different decisions made for different deployments.

- Application developers no longer write logic to recover from partial executions, since this is all handled by the IC. For the same reason, they also no longer write code to retrieve and store state, since all state is made implicitly durable through logging.

- Since the log implicitly contains all state changes for the application, debugging is greatly facilitated. To perform “time travel debugging” [23], we simply execute from a checkpoint before a bug occurred, and roll forward with the debugger attached, without involving any distributed components outside the immortal. This kind of debugging convenience is very difficult to replicate when applications explicitly write recovery code and durable state.

3.2 C# Language Binding

We begin by observing that the Ambrosia C# code for our running example is very similar to the naive code presented in Section 1. This is facilitated by a few similar assumptions about how code is run by our C# language binding:

- The public API for an Immortal is given using a C# interface.
- We assume that all incoming requests and interleaved return values of previous outgoing calls are processed sequentially in a single threaded manner
- Asynchrony (but not parallelism) is achieved using the standard C# async framework, where outgoing asynchronous calls may be awaited, resulting in a suspension of the request execution until the return value arrives in the incoming request and return value stream.

The biggest difference between our naive C# code and actual C# code derives from the use of DateTime.Now, which is non-deterministic upon replay. Specifically, upon recovery, a different timestamp is generated from the original, violating the language binding contract by producing different outgoing message argument values for the Send calls.

We overcome this problem by relaxing the language binding contract:

Weak Language Binding Contract: From some initial state, any execution of the same incoming requests in the same order must result in an equivalent final state. In addition, the outgoing requests must be deterministic in their number and destination ordering, but the contents may vary. Finally, the binding must also provide a state serializer.

The IC then guarantees that for each Immortal, only the first successfully logged incoming message is actually used for execution, regardless of content differences if the source recovers. This weak language binding contract, combined with the first logged replay guarantee together provide deterministic replayability across the whole distributed system by guaranteeing the integrity of message position, and ensuring that only one version of each message is ever acted upon.

We exploit this relaxed language binding contract to harden values from polled non-deterministic sources like DateTime.Now. Specifically, we perform an awaited Ambrosia self-call, passing the current time. Like all other incoming calls, the IC logs the self-call before processing. Upon execution, the self-call assigns the passed timestamp to a member, which is used when the awaiting original request continues.

The interface for the Ambrosia message forwarding immortal in our running example is shown below:

```
public interface IForwarder {
  void Process(string userMessage);
  void setStart(DateTime newSTime);
}
```

First, note the similarity to our naive code from Section 1. Also observe the existence of setStart, which is the self-call used to harden the polled timestamps.

Additionally, some important differences with the example in Section 2 are immediately apparent. First, note that there is no need for a batch interface. As we will see in Section 3.3, Ambrosia automatically batches requests when needed. Also, note the lack of sequence numbers. Since Ambrosia handles correct reconnection upon failure, there is no need to surface sequence numbers in the application code. Finally, Ambrosia today does not have automatic parallelization, which remains an item for future work, so there is no need to pass source.

The implementation of the forwarder contains the application logic, some attributes to support state serialization, and initialization to set up proxies for sending messages and reports:

```
[DataContract] class Forwarder:
             Immortal<IForwarderProxy>, IForwarder {
  [DataMember] DateTime startTime;
  [DataMember] int count=0;
  [DataMember] IForwardToProxy forwardTo;
  [DataMember] IReportToProxy reportTo;

  protected override async Task<bool> OnFirstStart() {
    forwardTo = GetProxy<IForwardTo>(“forwardTo”);
    reportTo = GetProxy<IReportTo>(“reportTo”);
  }

  void override async Task Process(string userMessage)
  {
    if (count == 0) {
      await thisProxy.setStartAsync (DateTime.Now);
    }
    count++;
    if (count%1000 == 0) {
      long reportNum = count/1000;
      DateTime lastTime = startTime;
      await thisProxy.setStartAsync (DateTime.Now);
      reportTo.Send(reportNum-1, startTime,
                    startTime-lastTime);
    }
    forwardTo.SendFork(userMessage);
  }

void override async Task setStart(DateTime newSTime)
  {
    startTime = newSTime;
} }
```

From the IForwarder interface, we generate C# libraries which contain abstract base classes with associated abstract method calls, which are implemented by the application writer. For instance, the Forwarder class in the above example implements IForwarder, which is in the associated generated C# library.

These generated libraries also contain proxies for making method calls on immortal instances of this type from other Ambrosia applications. For instance, in the above example, forwardToProxy is of type IForwardToProxy, which is a generated type for interacting with immortals which implement IForwardTo, which is not shown here. Like the naive version of our code, GetProxy is used to get a handle to an immortal registered in a catalog of immortals stored in a table. (Ambrosia uses Azure tables.). The two GetProxy calls reside in OnFirstStart, which is a logical constructor and is called once at the logical start of an application.

Observe that Forwarder is data contract serializable, and relevant fields, including references to other immortals (proxies), are labeled as data members. This ensures that when a checkpoint is taken, the forwarder's state is serialized. This state is then automatically deserialized during recovery.

In C#, Ambrosia calls to other immortals can be executed in either an awaitable (called async), or non-awaitable (called fork) fashion. For instance, the Send call on forwardToProxy is a forked call, which means it is not awaitable. This corresponds to the call version assumed for best performance in the naive code. Both RPC versions are automatically generated in the proxy for using an Ambrosia immortal. If an RPC is executed in a non-awaitable fashion, no return value is expected or sent, similar to sending an event. If an awaitable Ambrosia call is awaited, as in the call to setStart, the executing call is suspended until the return value arrives through the message queue from the coordinator. When the return value is handled, the suspended RPC is woken up, and continues execution.

Again, note the use of the setStart method. By making an Ambrosia call to itself with the newSTime argument, the IC ensures that each time the setStart call is successfully logged, all subsequent replays will use the logged timestamp argument rather than the one from the call generated by replay.

Note, in Forwarder, the lack of sequence number logic, LoadState and SaveState, and the disregard for the possibility of partial execution followed by failure. The code will nevertheless execute in Ambrosia in a fully fault tolerant manner, and is deterministically replayable as a result of Ambrosia being virtually resilient.

3.2.1 Consistency and the C# Binding Model

Consider the overall programming model presented in this section: Aside from the influence of non-replayable sources, like getting the current time, Ambrosia reduces the standard fully distributed cloud programming model with exposed failure and migration, to failure free independently executing actors running on a single node, where each actor is single threaded. Concurrency is allowed within an actor, specified using C#'s async framework. Specifically, each Immortal maps to an object, and Ambrosia method calls, which can be made as either async or fork calls, correspond to traditional C# method calls.

An interesting corollary of Ambrosia's single threaded Immortal execution model is that if all Immortal calls across the application are made using the async version of the call, and immediately awaited, and only one Immortal has an OnFirstStart, the application becomes a globally single-threaded job.

This does NOT mean that Ambrosia eliminates the need for further mechanisms required for consistency. Consider that locks and transactions are useful mechanisms for maintaining consistency even within a single node, when concurrent execution is present. In fact, these mechanisms may now be used in a manner consistent with single node execution without failure, eliminating failure induced mechanism inconsistency (e.g., losing lock state). Clearly, such consistency mechanisms are still potentially useful in Ambrosia applications which require consistency in the presence of concurrent calls across Immortals.

3.3 Performant IC Design

In Ambrosia, most of the heavy lifting for virtual resiliency is done in the IC. In particular, it is responsible for maintaining connections to other immortals, including reconnecting after disconnection or after recovery, as well as coordinating logging, checkpointing, and recovery. In addition, Ambrosia's IC design is very performance oriented, to great benefit (see Section 6).

We discuss the IC design in the context of an example shown in Figure 3. In this example, we follow an immortal method call, m, through the caller and callee's immortal coordinators, and all the logging and other activity caused by the call. We discuss our various performance optimizations in this context.

We begin with a method call on Immortal 2, labeled "1)", for step 1, made from Immortal 1's application. Once the method call is made in the application code, it is passed to the language binding, which serializes all the arguments, including the destination, and adds the result to a queue of page buffers for later sending (step 2). After serialization, the entire message associated with m, except the destination, which comes first, is considered one big byte array, and is not interpreted until the language binding in Immortal 2. This greatly facilitates high performance. Also, the strategy of queueing serialized requests for sending, as a result **creating batches of requests**, similar to the strategy used in Trill [8], is a strategy used throughout the system. In particular, while a batch is being sent, all arriving messages are added to the next batch, which is sent after the previous one is sent. These batches have a maximum size to control memory footprint, which can result in blocking the enqueueing source. Visually, buffers in the diagram imply that batching is happening.

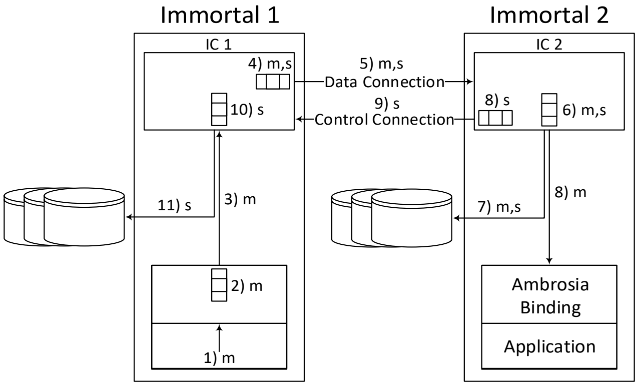

Figure 3: Method call protection example
(m = a message, s = m's sequence number)

In the IC, each outgoing connection to another immortal has an associated set of output buffers for batching, so when the method call is passed as part of a batch to the IC (step 3), the IC looks at the destination of each message in the batch, and adds it to the appropriate output buffer (step 4). It is worth pointing out that the batches (in addition to the individual messages!) produced in this output buffer aren't unpacked or interpreted until they reach the language binding which dispatches the individual method calls. This greatly facilitates performance. Also, notice the introduction of s, which is the sequence number associated with the message. This sequence number is associated with the outgoing connection, and monotonically increases with each message (not batch) sent through that connection. As a consequence, the association of sequence number to messages is the same for both Immortal 1 and Immortal 2, and is independent of batching decisions. While it's not actually transmitted, we include it here to note the importance of associating sequence numbers with messages. The batch is then sent to Immortal 2's IC (step 5), where it is added to a buffer, which serves as a log page (step 6).

When the buffer page is flushed to disk (step 7), the new high sequence number watermarks for all inputs which contributed to the flushed page are recorded as part of the log record. This enables a recovering immortal to know how much input has been consumed, which is important in establishing correct reconnection. Once the log page has been flushed, it is sent to the language binding for Immortal 2 (step 8), which dispatches the method. By waiting to send the page to the language binding until after it has been flushed, we are preventing the creation of side effects, in the form of outgoing calls, until the input has been "committed". This is similar, in spirit, to **batch commit**.

If this concluded our activity, the output buffer in IC 1 for the connection to IC 2 would grow infinitely, because, it couldn't release buffer pages until it knew that their contents had been flushed to disk by IC 2. For this reason, after IC 2 flushes the page to disk, it sends a batched message (step 8) back to IC 1 (step 9) containing the high sequence number watermark for the messages just flushed to disk originating from IC 1. Since these messages have been successfully flushed, there is no longer a need for IC 1 to remember them, and IC 1 may release the memory used to store them. These cleanup messages arrive along a different TCP connection than data to avoid possible deadlocks, when limitations on buffer sizes could prevent cleanup messages from getting through. As a result, for each unidirectional logical connection between immortals, there are 2 TCP connections, one for data and one for "control" messages.

Finally, IC 1 flushes changed per output high watermarks for received cleanup messages, each time log pages are flushed to disk (steps 10 and 11). This enables the IC, during recovery, to discard output produced during replay which consumers have already durably consumed.

In additional to the adaptive batching and batch committing, in order to improve performance, we also employ strategies familiar to DBMS architects for **concurrently writing to in-memory log pages efficiently**. While a log write is taking place, input arriving from multiple immortals, each with its own thread, contend for space in the log buffer page, effectively creating a serial order for arriving input from different sources.

We therefore take the usual approach, described in [18], where each thread grabs the position in the current log page in which it will write its bytes. Threads then concurrently write their bytes to the log record, where the last writer, which closes the page to further writes, waits for the concurrent writers to finish before writing the page to storage. After the page is closed to writing, new writers write to the next log page etc. Our implementation uses compare and swap to execute this strategy in a highly efficient manner, as is described in [18]. Like other page buffers, these buffers are size limited, and may cause blocking which cascades to senders.

It is also worth pointing out that checkpoints are periodically taken, when the log file exceeds a particular file size, at which point a new log file is started with all records which follow the checkpoint. Checkpoints contain both serialized application state, as well as immortal coordinator state, which includes the state of all send and log buffers. While checkpointing causes loss of availability in non active-active configurations, for active-active configurations, the primary simply starts a new log file without taking a checkpoint, and one of the secondaries is used to create the actual checkpoint. This allows checkpointing without associated loss of availability.

4. IMPULSES

4.1 Non-replayable sources and impulses

While polled non-replayable sources can be handled in a manner similar to the example in Section 3.2, which makes calls to DateTime.Now, what should we do about non-replayable sources that push data into an immortal? This could include data sources like live Twitter feeds, where best effort is all that's available, or even UI input, where a user can't be expected to reenter input during recovery. For applications with UI input, the immortal represents the state of a running application, including all information needed to render, and the UI makes calls into the immortal to modify that state from the same process.

When faced with such sources of non-determinism, Ambrosia developers use a novel feature called impulses, which are special RPCs that can only be called on a fully recovered and operating immortal instance. Specifically, this means that impulse calls cannot be made during recovery. When receiving an impulse call, the arguments are recorded in the log before execution, and will be replayed during recovery.

Impulses are identified in the immortal interface, like other RPCs, but are tagged with the property "ImpulseHandler". For instance, consider the following extension to our running example, where new messages may also originate from user input entered through the keyboard. We use "…" to represent previously presented code. The interface and immortal follow:

```
public interface IForwarder {
  …
  [ImpulseHandler]
  void AcceptInput(string newInp);
}

[DataContract] class Forwarder:
             Immortal<IForwarderProxy>, IForwarder {
…

  void override async Task AcceptInput(string newInp){
    thisProxy.Process(newInp);
  }

  protected override void BecomingPrimary() {
    new Thread(() => {
    while (true) {
      var line = Console.ReadLine();
      thisProxy.AcceptInputFork(line);
} } } }
```

Note that the background thread which accepts and submits user input, through our impulse, is created in BecomingPrimary. BecomingPrimary is an overrideable immortal method which is called after recovery is over, when the instance takes a primary role (see Section 5.1 for a discussion of Ambrosia's active-active capabilities). By starting the input thread in this method, we ensure that new input isn't being submitted through our impulse during recovery, and that none of the secondaries, if we are running in an active-active deployment, are requesting input.

From the above, it should be clear that all incoming external interactions can be easily handled in an at most once manner with impulses. Traditional approaches for handling external incoming and outgoing interactions may still be used, similar to what's presented in Section 2, given that code is guaranteed to run at least once. Exploiting this guarantee, in conjunction with application level sequence numbers and durable queues, one can still achieve exactly once semantics, but without the benefits of Ambrosia.

4.2 Implementing impulses

Unlike conventional Ambrosia methods, impulses are logically executed at most once. If an immortal, which collects and sends an impulse, fails prior to transmitting the impulse to the receiving immortal, or if both the sender and receiver fail before the impulse is made durable, the impulse will be lost. In particular, we cannot rely on replay to reproduce the outgoing impulse on the sender.

As a result, if we tried to treat impulses as ordinary method calls in the protocol described in Section 3.3, the sequence numbers in senders and receivers could become inconsistent. For instance, suppose a sender A takes a checkpoint, and at the time of the checkpoint, has sent a total of 200 messages to receiver B. After checkpointing, assume 100 impulses are sent to B, after which A fails. After A's recovery, during which outgoing impulses are not recreated, A will believe it has sent 200 messages to B, while B believes it has received 300 messages from A. A will subsequently eat the next 100 messages to B, though they've never been sent.

We therefore keep track of two sequence numbers instead of one: total, and replayable (i.e. non impulses). The protocol described below has the following behavior:

1) Non-impulses are executed exactly once in the proper order

2) Logged impulses are executed exactly once in their proper order

3) Impulses which are not logged are lost

4) Impulses from a recovering sender not already sent to a receiver are lost, including checkpointed impulses in send buffers

It is easy to see that in the absence of system failure, running the protocol described in Section 3.3, but with sequence number pairs, will result in correct behavior with no loss of impulses. Similarly, broken TCP connections (without system failure), can be similarly healed without loss.

Complications, however, ensue, with system failure and recovery. Recovery begins by restoring the last checkpoint, including both application and immortal coordinator state. Recovery then cleans all impulses out of all restored send buffers. This enforces behavior 4 above. Note that as recovery processes log records, it retrieves both total and replayable sequence numbers for cleanup messages (written to the log in step 11 in Figure 3). Since, at this point, the recovered output buffers only contain replayable messages, the replayable sequence numbers are used to clean output buffers during recovery.

After replay, during reconnection, the receiver sends both the replayable and total sequence numbers to the sender. The sender then begins replay from the call following the last received replayable message, and sets the total sequence number for that message to 1 higher than the total sequence number from the receiver. In this manner, the receiver receives the first call following the last received, and sequence numbers between the sender and receiver are made consistent.

Note that the sequence number consistency enforcing protocol described above is only for reconnecting for the first time after recovery. When reconnecting in other situations (e.g. after TCP connection failure), sequence numbers are already consistent, and the sender simply starts from the message after the last received.

5. LOG BASED AMBROSIA FEATURES

There are four additional major capabilities enabled by Ambrosia's logging-based approach to virtual resiliency.

5.1 High Availability

The first of these features is high availability through active standbys. In Ambrosia, the log, and associated checkpoints, are written to a directory specified by the immortal deployer. In both Windows and Linux, that directory can be backed by either local storage, or cloud-replicated storage. For instance, Azure Files [24] may be mounted on all internet connected Linux and Windows machines. Alternatively, Azure Managed Disks [24] offer a performant and very cost-effective alternative for immortals running inside Azure datacenters.

At any given moment, there is one primary, which produces the log and is connected to other Ambrosia immortals, and secondaries, which consume the log in recovery mode, until they become primary. Leader election is simply the result of all instances continuously (e.g. every half second) trying to acquire the exclusive write lock on the log file. When an instance acquires the lock, it becomes primary, and CRA establishes all connections to other immortals. If a primary ever loses the file lock, it commits suicide.

The log is broken into deployer specified chunks, such that whenever a threshold is reached, a new log file is created with an incremented chunk number as part of the filename. When a secondary becomes primary it immediately starts a new log file.

In Ambrosia's implementation of high availability, checkpoints are generated by a secondary, such that each time a new log file is started, there is an associated generated checkpoint which contains the state of the immortal instance at the *start* of the log file. The secondary-based checkpointing prevents loss of primary availability while checkpointing and turns out to be the optimal strategy in a resource-reservation based environment like the cloud [21]. A new secondary then starts from the latest checkpoint and rolls forward until it is caught up.

5.2 Time Travel Debugging

Using checkpoints and log files, Ambrosia exploits application deterministic replay to implement time travel debugging: the developer starts the application process and attaches the debugger, and then starts the immortal coordinator in time travel mode. In this mode, the developer points the coordinator to the log and checkpoint files (which may still be live) and specifies the checkpoint number to begin recovering from. The immortal coordinator then runs recovery, never becoming primary.

Since the debugger is attached to the application process, all the usual debugger features may be used, like setting breakpoints, and stepping through code. Because replaying the log is deterministic, the same application behavior may be replayed and debugged as many times as desired, even against a live log.

5.3 Retroactive Code Testing

Related to time travel debugging, if the application writer wants to test an alternate version of the application which has the same interface and state (as is frequently the case when fixing bugs), they can perform time travel debugging with the updated version of the application, using the debugger, to find a bug or test a fix. A developer may even use this feature to create new application generated logs against the replay.

Observe that new versions of services may be rolled out this way, where the new version starts as an active secondary and becomes primary when all instances associated with old versions are killed.

5.4 Live Service Upgrades

Occasionally, services go through significant upgrades, where the API to the service broadens, and/or where the type of the application state changes (e.g. the addition of new counters). For such situations, Ambrosia allows developers to define an "upgraded Immortal", where both old and new versions of the application code are present in the process.

When such an immortal is deployed, it recovers using the old version of the service. When it becomes primary, it calls a constructor for the new version of the service, which takes as an argument the state of the old version at the time it becomes primary. A new checkpoint is then taken of the new version of the service, and the upgrade is complete.

To deploy such an upgrade, it is initially added as an active secondary. While killing all the instances of the old service, the new version becomes primary and the service continues. Note that any old versions of the service still running simply die once the new version becomes primary.

6. EXPERIMENTAL EVALUATION

In this section, we present performance results comparing Ambrosia with alternative baselines. We measure throughput, latency, overhead of logging, fail-over and recovery times.

6.1 Implementations under Test

We explore the performance of several different implementations of a client-server application, where the client sends requests to the server, each of which contains a byte array. The server counts the total number of requests and bytes sent. We focus on three different implementations, which we describe next.

6.1.1 gRPC

gRPC is a performance-oriented cloud RPC framework that does not do any logging or recovery. It is just straight-up RPC. As such, we would expect it to soundly beat Ambrosia, and it should represent an upper bound of what's possible.

Note that we used the gRPC streaming implementation in C++, which according to [22], is the most performant option. In our streaming setup, the server has a streaming RPC, called Receive, which takes a byte array of the appropriate size and keeps a running total of all bytes received. The choice of a byte array is designed to minimize serialization and deserialization overhead, which is orthogonal to the issues tested here.

For the latency test, we use a single RPC call, which performs the fastest round trip available in gRPC, performing one at a time to ensure minimum interference.

6.1.2 Ambrosia – C#

The focus of this paper, we provide both .NET framework and core implementations, and run on both Windows and Linux. In these experiments, we use the .NET framework implementation on Windows. We wrote two Immortals: a client and a server. Both are fully recoverable and generate their own logs and checkpoints. Each write their logs to Azure storage. In particular, we wrote our logs to 6x P10 Azure Premium Managed Disks, which were pooled together in a software RAID configuration with aggregate bandwidth of 600 MB/s. Note that this RAID configuration represents the cheapest way to allocate replicated storage with the bandwidth we anticipated we'd need for our tests.

Like the gRPC implementation, the server computes the total bytes sent, which is part of the server's serializable state, and is marked as a data member. Except for our resiliency tests, checkpointing (but not logging) was turned off for these experiments.

Like gRPC, we use our streaming RPC calls (Fork). Conceptually, there is very little difference between the code written to implement this microbenchmark in gRPC and Ambrosia, although differences in C# and C++ make the Ambrosia-C# version more readable.

6.1.3 Serverless and Stateless Compute

A popular design (see Figure 1) for microservices is to ingress data into a fully managed, real-time data ingestion and messaging layer such as Azure Event Hubs or AWS Kinesis. The messaging layer feeds data to a serverless execution fabric such as Azure Functions or AWS Lambda, which pulls data batches from the messaging layer and executes the user code. The user code is stateless; it loads state from a persistent backend such as Azure Tables or AWS DynamoDB, runs application logic, and writes back the state at the end of execution. We can compute the total cost to run a microservice using this architecture, in terms of dollar amount per month, per MB/sec of ingress. We assume that both the messaging layer and the serverless functions layer can parallelize as much as needed. For Azure, the cost components of a deployment are:

(1)EventHub ingress cost: It currently costs $0.028 per million messages, plus $0.015 per hour, per throughput unit (1 MB/sec ingress, 2 MB/sec egress). Event Hubs also offers a dedicated option that costs $4999.77 per month; we choose the lower cost between these options for our computations.

(2)Azure Function costs have two components. First, there is a cost of $0.20 per million executions, we assume that a function is invoked with batches containing up to 256KB of data from Event Hubs. Second, there is an execution time cost of $0.000016 per GB/sec, a unit of resource consumption. Resource consumption is calculated by multiplying average memory size in gigabytes by the time in milliseconds it takes to execute the function. We assume 128MB of average memory (the lowest allowed) and that it takes 0.1ms per event in the batch fed to Azure functions.

(3)We perform one read and write to storage per function invocation. Azure Tables cost $0.00036 per 10,000 transactions, with a $0.07 per GB cost for the actual first terabyte of storage. We assume that 1GB is enough to hold the state for our example.

The costs for AWS were computed similarly and verified using the AWS cost calculator [31]. Briefly, AWS Kinesis is priced at $0.015 per shard-hour (1MB/second ingress, 2MB/second egress), plus $0.014 per million PUT payload units, AWS DynamoDB is priced at $1.25 per million reads and $0.25 per million writes, and AWS Lambda is priced similarly to Azure Functions.

We vary the per-message size from 16 bytes and up, and compute costs over a month if the service ingests 100MB/sec over the entire month. We then scale the result down to report the cost per month, per MB/sec of ingested data.

6.2 Experimental Setup

In all cases except serverless, we perform two types of throughput experiments, run on 2xD14v2 (16 cores, unlimited storage bandwidth) Azure instances to optimize performance and the most efficient of 2xF2S (2 cores, 96 MB/s to storage) and 2xF2SV2 (2 cores, 47 MB/s to storage) to optimize price/performance. We also measure ping latency under minimal load, where one instance acts as the client, and the other as the server. The same actual instances were used in all experiments to eliminate hardware variation.

The resilient setup based on Section 2 uses only serverless components, and simply calculates costs for comparable work done, and measures end-to-end ping latency on Azure, starting from a VM, going to Event Hubs, serviced by an Azure Function, outputting to Event Hubs, and retrieved by the original VM.

6.3 Results

6.3.1 Throughput

The results of the performance optimized throughput experiments are shown in Figure 4. First, observe that even though gRPC is a bare RPC framework, without any notion of failure resiliency, it is nevertheless significantly less performant for small message sizes than Ambrosia. For 16-byte arguments, it is actually more than 10x slower than Ambrosia-C#! Also, note that gRPC slightly outperforms Ambrosia-C# near the throughput limit, but pulls back, for some reason, to efficiency levels indistinguishable from Ambrosia-C#. We saw this trend continue for even larger message sizes.

For the throughput -- price/performance optimized experiment, we compare the costs of performing our throughput experiment in terms of cost per month per MB if we ran the experiment continuously for a month.

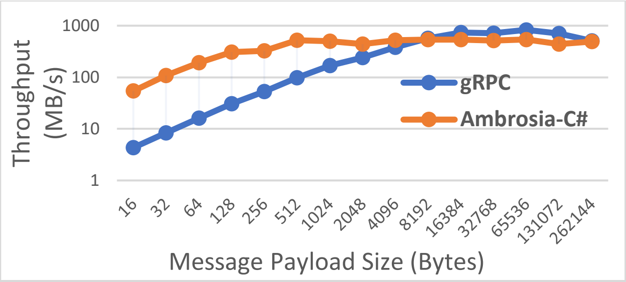

Figure 4: Performance Optimized

Performing any comparison of this sort is fraught with difficult decisions which can make one strategy fare better or worse. For instance, EventHub can be run in either basic or standard mode. There are several differences, one of which is the ability to write queue history to cold storage for later processing. The difference in price is a factor of two. Of course, Ambrosia provides this capability, making the log directly available. Nevertheless, in our calculations, we chose the basic level of support, as some users may not care about this feature.

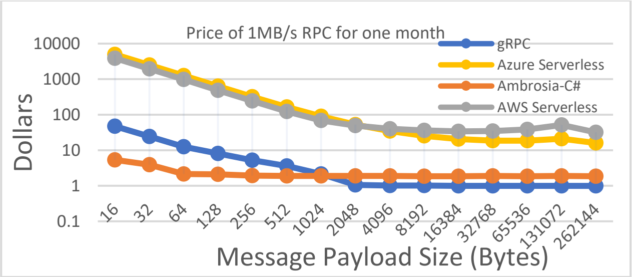

Figure 5: Price/Performance Optimized

Also, unlike earlier throughput experiments, for gRPC and Ambrosia, we chose instances which optimize price/performance, even though overall performance is lower, in some cases choosing different instances for different message sizes. Also, we use the on-demand price of these VMs, which can be reduced by about 30% with long term reservations.

The results are shown in Figure 5. Both gRPC and Ambrosia are much cheaper than stateless compute, in some cases, by about 1000x! For gRPC, this isn't a surprise, as our stateless compute pipeline is resilient to failure, and gRPC isn't, but Ambrosia provides equivalent resiliency with a much easier programming model. Another surprise, Ambrosia is significantly cheaper than gRPC for message sizes below 1K. One might expect the cost of storage bandwidth to be a large component of Ambrosia's cost to run, but this is not the case. The monthly cost of an FS2 instance, one of the cheapest VM instances on Azure, is $169.36, while the monthly cost of 100MB of continuous storage bandwidth is only $19.71. There is a throttle on storage bandwidth, though, for such small VM sizes, which is responsible for gRPC pulling ahead of Ambrosia for larger message sizes by about a factor of 1.9.

We also ran the throughput experiment with logging turned off on our largest instances. Turning off logging increases throughput by 33% for message sizes >= 256, and by 0% for 16 byte messages. 33% reflects network bound scenarios, where per connection TCP bandwidth is surpassed, since storage is written to using a different connection. Eventually, though, NIC capacity is reached, causing 1/4 overall loss of throughput. For small message sizes, the CPU is the bottleneck and the logging overhead becomes negligible.

6.3.2 Latency

This experiment is designed to test the latency of the various implementations under light load, which reflects the best latency achievable by these systems. For this we perform pings, where only one outstanding ping is allowed. The results of the ping experiment are shown in Table 1:

Table 1: Latency in milliseconds

	0.5	**0.9**	**0.99**	**0.999**	**Mean**
Ambrosia	6.57	7.1	8.71	11.34	6.63
gRPC	0.5	0.59	0.8	61.85	0.58
Azure Serverless	31.62	130.5	324.7	6708	80.51
Ambrosia-NoLog	2.15	2.49	3.05	7.59	2.32

The first four columns show the latencies for various percentiles. For instance, 0.5 is the median, 0.9 is the value for which 90% of the latencies are lower, etc. Unsurprisingly, gRPC, which simply sends a message across the wire from one machine to the other, is the clear latency champ. Ambrosia, on the other hand, must make two sequential round trips to our P10 disks. What we see here is that adaptive batching and asynchrony completely closes the gap (and then some) on throughput, but not latency. Oddly, gRPC has higher tail latency around 60ms. These are not one-time outliers; they occur regularly throughout the workload. It likely reflects global locking associated with gRPC periodically cleaning up resources. For stateless compute, latency is about 5x higher than Ambrosia at the median, but steadily becomes higher and higher as the percentile increases, resulting in 600x higher latency than Ambrosia at three nines.

The last row in Table 1 shows the result of the latency experiment with logging turned off. As expected, with logging turned off Ambrosia's latencies are roughly 3x lower than with logging. But they are still higher than those of gRPC's because Ambrosia requires strictly more network hops vs. gRPC due to its multi-process design (Figure 2).

6.3.3 Testing Ambrosia's Resiliency

This last set of experiments measures (a) the fail-over time, and (b) the recovery time. In doing these measurements our goal is to provide evidence that in the presence of failures, Ambrosia can fail-over quickly and also that Ambrosia's recovery overhead is low.

To test Ambrosia's failover performance, we performed our latency test continuously with 3 active instances for the server, where the log is backed by Azure Files. We induced periodic failure of the primary server, resulting in failover to an active secondary, and restarted the failed server to setup for the next failure. We then found the corresponding spikes in ping latency. Since failover time was orders of magnitude higher than ping latency, we simply used these measured latency spikes as our failover time measurements.

The result of this experiment shows that with this setup, Ambrosia fails-over in 1.8 seconds, on average, with little variation.

To measure recovery time, we conduct an experiment where we perform our throughput test on 20GB of 64K messages, with only the initial checkpoint generated. We measure the total execution time of this experiment, without failures, to be 77 seconds. Ambrosia is able to recover from the initial checkpoint and log from start to finish in 40 seconds. This shows that Ambrosia recovery costs are less than the Ambrosia costs of running the service with logging in the first place, which puts a bound on recovery time.

Shrink [21] combines this sort of per service information with other information, like the availability target, and rate of failure, and additionally tunes parameters like checkpointing frequency and number of active instances to achieve optimal cost for an availability target. For instance, consider the following tradeoffs: adding more active secondaries to mask recovery time increases availability, but at additional cost. More frequent checkpointing decreases recovery time, but increases the cost of overall checkpoint generation, and is limited by the size of the checkpoint compared to the size of the log between checkpoints. Connecting Ambrosia and Shrink to create a highly efficient, adaptive deployment framework is a subject of future work.

7. FURTHER RELATED WORK

Ambrosia builds on ideas first proposed in Phoenix ([1], [30]). Similar to Ambrosia, Phoenix provides virtual resiliency focusing on database applications but lacks support for non-determinism, design elements that would provide performance comparable to Ambrosia, and language and machine heterogeneity. In [33], authors propose a light-weight logging and replay technique namely, command logging. Their goal is to overcome the overhead of fine-grained logging typically required by the ARIES protocol [32]. Command logging records all the transactions which were executed on the database, taking transactionally consistent checkpoints of the log periodically. During recovery, starting from a recent checkpoint, it replays all the commands in the log to bring the database to a consistent state. At a high-level, Ambrosia also uses a light-weight logging and recovery technique and thus benefits from this research. However, Ambrosia's use-cases go beyond database systems, and thus certain assumptions do not hold. For example, as opposed to [33], in Ambrosia all the commands are logged before they are executed and it assumes no transactions support (and hence no aborts).

Support for deterministically replayable computation is relevant for language bindings, or maybe even the construction of languages and runtimes specifically for use with Ambrosia. In C#, deterministic replayability is accomplished via a combination of programmer obligations and language-specific mechanisms. Ensuring deterministic execution has been the subject of a substantial body of research in operating systems ([10], [11]), threading libraries ([12], [13]), and programming languages ([14], [15]). When developing a service to run on top of Ambrosia, any combination of these approaches may be used, as deterministic replayability is a local property of each communication endpoint.

One of the important design goals for Ambrosia is to support machine heterogeneity. Sapphire [4] is the system that comes closest to our work in this respect. Sapphire provides a distributed runtime, which can be extended to run code across a variety of devices ranging from cloud data center machines to mobile devices. Unlike Ambrosia, the main focus in Sapphire is on flexibility and extensibility, which is achieved by separating the application logic from the deployment logic. Sapphire could benefit from and build on top of the virtual resiliency guarantees provided by Ambrosia.

Ambrosia also takes inspiration from actor-based systems, such as Orleans [16] and Erlang [34], which provide simple abstractions to build scalable distributed applications. In contrast, Ambrosia provides virtual resiliency guarantees with high performance. Reactor [41] extends actor-based frameworks with support for transactions. As discussed earlier, transactions are orthogonal to the virtual resiliency guarantees provided by Ambrosia.

There is also work on VM/container level replication for resiliency based on checkpointing ([37], [38], [39]). These are all relatively high cost physical approaches to logging that require infrastructure support, as opposed to our lightweight logical approach which doesn’t rely on special infrastructure support. While some of these approaches ([36], [35]) enable virtual resiliency on servers, even with arbitrary multithreaded code, clients are out of scope, and must deal with broken TCP connections. In addition, since they don’t have a logical understanding of the workload, they are unable to support retroactive code testing, live service upgrades, and efficient migration across architectures and operating systems. Finally, the most efficient of these is based on an epoch mechanism which loses state changes, meaning that users have to choose between lower overhead and time travel debugging.

8. CONCLUSIONS AND FUTURE WORK

This paper introduces Ambrosia, the first general purpose platform for distributed nondeterministic applications that provides its developers virtual resiliency with unprecedented performance, and the flexibility of working across a variety of machines, operating systems, and languages. Furthermore, Ambrosia supports high availability, time-travel debugging, retroactive debugging, and live service upgrades. Ambrosia is a real system, used to build a service which manages hundreds of thousands of machines. The service development team, when asked for feedback, indicated that Ambrosia, in practice, significantly improved the time it took to get their service to an acceptable level of quality, due to the elimination of failure associated bugs, and always-on time travel debugging.

Ambrosia’s performance depends upon technology, developed by the database community, used to develop performant data processing systems. For instance, we make extensive use of adaptive batching from the streaming community, efficient log writing, and batch commit concepts.

Therefore, Ambrosia achieves competitive throughput with gRPC, a widely used non-resilient RPC framework, but with higher latency costs due to cloud storage latency. Furthermore, Ambrosia is both simpler to program, and cheaper to run, than a typical stateless compute cloud configuration designed to be resilient to failure, outperforming this configuration by about 1000x for small message sizes on cost, and 1-3 orders of magnitude on latency.

These results also indicate that the stateless compute approach embraced by most cloud developers is likely a temporary workaround until systems like Ambrosia mature.

While Ambrosia is immediately useful, there are many related research problems worth thinking about. The most obvious next step is elastic scale out. While databases have certainly solved the problem for transactional systems, they rely on the ability to abort in flight transactions. In an exactly once system, this is not an option, and performant solutions to this problem must be found as part of a desirable implementation.

While this work hasn’t emphasized the ability to relocate immortals on other machines, this is potentially very exciting in the world of devices, where Ambrosia facilitates the construction of easily migratable apps from one device to another, without loss of state.

Figuring out how to support other languages is both useful and interesting. For instance, the language binding choices made for Javascript, a single-threaded language, may be quite different from a language like C#, where thread non-determinism can complicate achieving deterministically replayable behavior.

Finally, as the number of CPUs and distributed state proliferates with IOT, the problem of distributed state management in distributed applications will become excruciating. Ambrosia provides a crucial building block to tame this complexity. Understanding Ambrosia’s role, and potential gaps, for these scenarios is very important.

9. REFERENCES

[1] Roger S. Barga, David B. Lomet. Phoenix: Making Applications Robust. SIGMOD Conference 1999: 562-564

[2] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI Conference 2004: 137-150

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica: Spark: Cluster Computing with Working Sets. HotCloud 2010

[4] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, Henry M. Levy: Customizable and Extensible Deployment for Mobile/Cloud Applications. OSDI 2014: 97-112

[5] Laura M. Haas, Patricia G. Selinger, Elisa Bertino, Dean Daniels, Bruce G. Lindsay, Guy M. Lohman, Yoshifumi Masunaga, C. Mohan, Pui Ng, Paul F. Wilms, Robert A. Yost: R*: A Research Project on Distributed Relational DBMS. IEEE Database Eng. Bull. 5(4): 28-32 (1982)

[6] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3): 375-408 (2002)

[7] Wilschut, A., and Apers, P.: Dataflow Query Execution in a Parallel Main-memory Environment. In Distributed and Parallel Databases 1(1), 1993.

[8] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James F. Terwilliger: Trill: Engineering a Library for Diverse Analytics. IEEE Data Eng. Bull. 38(4): 51-60 (2015)

[9] Manish Mehta, David J. DeWitt: Data Placement in Shared-Nothing Parallel Database Systems. VLDB J. 6(1): 53-72 (1997)

[10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, 2010.

[11] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven Gribble. Deterministic process groups in dOS. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, 2010.

[12] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pages 97–108, New York, NY, USA, 2009. ACM.

[13] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In Symposium on Operating Systems Principles. ACM, 2011.

[14] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In POPL, pages 257–270, 2014.

[15] Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A type and effect system for deterministic parallel Java. In Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09, page 97, Orlando, Florida, USA, 2009.

[16] Philip A. Bernstein, Sergey Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Computing 20(5): 71-75 (2016)

[17] Badrish Chandramouli, Raul Castro Fernandez, Jonathan Goldstein, Ahmed Eldawy, Abdul Quamar: Quill: Efficient, Transferable, and Rich Analytics at Scale. PVLDB 9(14): 1623-1634 (2016)

[18] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta: LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888 (2013)

[19] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84). ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/602259.602261

[20] Ibrahim Sabek, Badrish Chandramouli, Umar F. Minhas. CRA: Enabling Data-Intensive Applications in Containerized Environments. In ICDE Conference, 2019. https://github.com/microsoft/CRA.

[21] Badrish Chandramouli, Jonathan Goldstein: Shrink - Prescribing Resiliency Solutions for Streaming. PVLDB 10(5): 505-516 (2017).

[22] gRPC Benchmarking. http://grpc.io/docs/guides/benchmarking.html

[23] ET Barr, M Marron: Tardis: Affordable time-travel debugging in managed runtimes. ACM SIGPLAN Notices 49 (10), 67-82

[24] Azure Storage. https://azure.microsoft.com/en-us/product-categories/storage/

[25] Data Contracts. https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/using-data-contracts

[26] Protocol Buffers. https://developers.google.com/protocol-buffers/

[27] Avro. https://avro.apache.org/

[28] JSON. http://json.org/

[29] A.M.B.R.O.S.I.A. https://github.com/microsoft/AMBROSIA

[30] Roger S. Barga, David B. Lomet, German Shegalov, Gerhard Weikum. Recovery guarantees for Internet applications. ACM Trans. Internet Techn. 4(3): 289-328 (2004)

[31] AWS Cost Calculator. https://calculator.s3.amazonaws.com/index.html.

[32] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17, 1 (March 1992), 94-162

[33] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, Michael Stonebraker: Rethinking main memory OLTP recovery. ICDE 2014: 604-615

[34] Erlang Programming Language. http://www.erlang.org/

[35] Manos Kapritsos and Yang Wang, University of Texas at Austin; Vivien Quema, Grenoble INP; Allen Clement, MPI-SWS; Lorenzo Alvisi and Mike Dahlin, University of Texas at Austin. All about Eve: Execute-Verify Replication for Multi-Core Servers. OSDI, 2012

[36] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chenx, and Junfeng Yang. PAXOS Made Transparent. SOSP 2015

[37] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. NSDI 2008

[38] George W. Dunlap, Dominic G. Lucchetti, Peter M. Chen, Michael Fetterman. Execution Replay for Multiprocessor Virtual Machines. ACM VEE 2008

[39] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, Chen Yu. Live Migration of Virtual Machine Based on Full System Trace and Replay. HPDC 2009

[40] Ambrosia Technical Report. https://www.microsoft.com/en-us/research/publication/a-m-b-r-o-s-i-a-providing-performant-virtual-resiliency-for-distributed-applications/

[41] Shah, Vivek and Marcos Antonio Vaz Salles. "Reactors: A case for predictable, virtualized actor database systems." Proceedings of the 2018 International Conference on Management of Data. ACM, 2018.

[42] Alvaro, Peter, et al. "BOOM: Data-centric programming in the datacenter." EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-113 (2009).