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

Jonathan Goldstein, with Ahmed Abdelhamid[ɪ], Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck[ғ], Christopher Meiklejohn[ʳ], Umar Farooq Minhas, Ryan Newton[ɣ], Rahee Ghosh Peshawaria, Tal Zaccai, Irene Zhang

Microsoft, Purdue University[ɪ], University of Washington[ғ], Carnegie Mellon University[ʳ], Indiana University[ɣ]

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. For instance, consider the simple case of two objects, one called Client, and the other called 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 in a single process, 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, method calls are performed through an RPC mechanism. If Client fails (and is restarted) 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 issuing the RPC request, but before receiving the return value, Server will initially try to provide to Client an unexpected return value. 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 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.

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 50 years of work from the database community, which has shown how declarative database systems, which produce deterministically replayable behavior through logging, can completely isolate the developer from the possibility of failure in a performant manner. Most recently, map-reduce and its progeny, 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, including VM/container checkpointing approaches, the failure to address all important issues has prevented their ubiquity. 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. A compelling general-purpose solution to this problem must address the following:

- **Virtual Resiliency** - 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.
- 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.
- Performance/Cost - In order to offer virtual resiliency as a general purpose capability, performance must be comparable to failure-sensitive code with a good application specific strategy (e.g. within a factor of 2).
- Machine Heterogeneity – While machines inside a datacenter can be homogenous, complete distributed apps typically span devices and datacenters. The end-to-end semantics must be easy to understand, reason about, and code against.
- Language Heterogeneity – Because distributed applications span across a wide variety of machines and settings, distributed components written in different languages must be able to work together.

We address this problem with 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 (https://github.com/Microsoft/AMBROSIA), and is used in a cloud service which manages the machine images of hundreds of thousands of machines running a cloud application.

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. Ambrosia achieves throughput results which exceed Google's gRPC, by up to a factor of 12.7X, despite gRPC lacking any kind of failure protection. 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.

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, retroactive code testing, and inflight application upgrades. Ambrosia's machine heterogeneity goes even 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.

Jonathan Goldstein short BIO:

Over the last 20 years, I have worked at Microsoft in a combination of research and product roles. In particular, I've spent about 15 years as a researcher at MSR, doing fundamental research in streaming, big data processing, databases, and distributed computing. My style of working is to attack difficult problems, and through fundamental understanding and insight, create new artifacts that enable important problems to be solved in vastly better ways. For instance, my work on streaming data processing enabled people with real time data processing problems to specify their processing logic in new, powerful ways, and also resulted in an artifact called Trill, which was orders of magnitude more performant than anything which preceded it. Within the academic community, I have published many papers, some with best paper awards (e.g. Best Paper Award at ICDE 2012), and two with test of time awards (e.g. SIGMOD 2011 Test of Time award and ICDT 2018 Test of Time award), and have also taken many organizational roles in database conferences. My research has also had significant impact on many Microsoft products, including SQL Server, Office, Windows, Bing, and Halo, as well as leading to the creation of entirely new products like Microsoft StreamInsight, Azure Stream Analytics, Trill, and Ambrosia. I spent 5 years building Microsoft StreamInsight, serving as a founder and architect for the product. Trill has become the de-facto standard for temporal and stream data processing within Microsoft, and years after creation, is still the most expressive and performant general purpose stream data processor in the world. I am also an inventor of 30+ patents.