Does The Cloud Need Stabilizing?

Murat Demirbas, Aleksey Charapko, Ailidani Ailijiang
{demirbas, acharapk, ailidani}@buffalo.edu

University at Buffalo, SUNY

Abstract. The last decade has witnessed rapid proliferation of cloud computing. While even the smallest distributed programs (with 3-5 actions) produce many unanticipated error cases due to concurrency involved, it seems short of a miracle these web-services are able to operate at those vast scales. In this paper, we explore the factors that contribute most to the high-availability of cloud computing services and examine where self-stabilization could fit in that picture.

1 Introduction

The last decade has witnessed rapid proliferation of cloud computing. Internet-scale webservices have been developed providing search services over billions of webpages (such as Google and Bing), and providing social network applications to billions of users (such as Facebook and Twitter). While even the smallest distributed programs (with 3-5 actions) can produce many unanticipated error cases due to concurrency involved, it seems short of a miracle that these web-services are able to operate at those vast scales. These services have their share of occasional mishap and downtimes, but overall they hold up really well.

In this paper, we try to answer what factors contribute most to the high-availability of cloud computing services, what type of fault-tolerance and recovery mechanisms are employed by the cloud computing systems, and whether self-stabilization[1] fits anywhere in that picture.

Self-stabilization had shown a lot of promise early on for being applicable in the cloud computing domain. The CAP theorem[2] seemed to motivate the need for designing eventually-consistent systems for the cloud and self-stabilization has been pointed out by experts as a promising direction towards developing a cloud computing research agenda [9]. On the other hand, there has not been many examples of stabilization in the clouds. For the last 6 years, the first author has been thinking about writing a "Stabilization in the Clouds" position paper, or even a survey when he thought there would surely be plenty of stabilizing

---

[1] Stabilization [18, 6] is a type of fault tolerance that advocates dealing with faults in a principled unified manner instead of on a case by case basis: Instead of trying to figure out how much faults can disrupt the system's operation, stabilization assumes arbitrary state corruption, which covers all possible worst-case collusions of faults and program actions. Stabilization then advocates designing recovery actions that takes the program back to invariant states starting from any arbitrary state.

design examples in the cloud. However, this proved to be a tricky undertaking. Discounting the design of eventually-consistent key-value stores [17] and application of Conflict-free Replicated Data Types (CRDTs) [34] for replication within key-value stores, the examples of self-stabilization in the cloud computing domain have been overwhelmingly trivial.

We ascribe the reason self-stabilization has not been prominent in the cloud to our observation that cloud computing systems use infrastructure support to keep things simple and reduce the need for sophisticated design of fault-tolerance mechanisms. In particular, we identify the following cloud design principles to be the most important factors contributing to the high-availability of cloud services.

1. **Keep the services "stateless" to avoid state corruption.** By leveraging on distributed stores for maintaining application data and on ZooKeeper [26] for distributed coordination, the cloud computing systems keep the computing nodes almost stateless. Due to abundance of storage nodes, the key-value stores [1] and databases [15, 11] replicate the data multiple times and achieves high-availability and fault-tolerance. In our earlier work [4], we explored how distributed coordination is achieved in cloud computing, and found that an overwhelming majority of cloud computing systems adopt coordination services, such as ZooKeeper, for maintaining concurrency-critical state. [3]

2. **Design loosely coupled distributed services where nodes are dispensable/substitutable.** The service-oriented architecture [37], and the RESTful APIs for composing microservices are very prevalent design patterns for cloud computing systems, and they help facilitate the design of loosely-coupled distributed services. This minimizes the footprint and complexity of the global invariants maintained across nodes in the cloud computing systems. Finally, the virtual computing abstractions, such as virtual machines [36], containers [13], and lambda computing servers [7] help make computing nodes easily restartable and substitutable for each other.

3. **Leverage on low level infrastructure and sharding when building applications.** The low-level cloud computing infrastructure often contain more interesting/critical invariants and thus they are designed by experienced engineers, tested rigorously, and sometimes even formally verified [32]. Higher-level applications leverage on the low-level infrastructure, and avoid complicated invariants as they resort to **sharding** at the object-level and user-level [35]. Sharding reduces the atomicity of updates, but this level of atomicity has been adequate for most webservices, such as social networks.

A common theme among these principles is that they keep the services simple, and trivially "stabilizing", in the informal sense of the term. Does this mean

---

[2] CAP theorem says that in the presence of a partition, it is impossible to have strong consistency and high availability [10, 20, 21]. Since cloud computing systems value high-availability (as they make money by being highly-available), weaker-consistency models such as eventual-consistency became popular.

[3] ZooKeeper is a Paxos replicated state machine implementation with a filesystem API as the interface. It is used for keeping "metadata" safely and consistently to the face of node failures, concurrency, and asynchrony.

that self-stabilization research is unwarranted for cloud computing systems? To answer this, we point to some silver lining in the clouds for stabilization research. We notice a trend that even at the application-level, the distributed systems software starts to get more complicated/convoluted as services with more ambitious coordination needs are being build.

In particular, we explore the opportunity of applying self-stabilization to tame the complications that arise when composing multiple microservices to provide higher-level services. This is getting more common with the increased consumer demand for higher-level and more sophisticated web services. The higher-level services are in effect implementing distributed transactions over the federated microservices from multiple geodistributed vendors/parties, and that makes them prone to state unsynchronization and corruption due to incomplete/failed requests at some microservices. At the data processing systems level, we also highlight a need for self-regulating and self-stabilizing design for realtime stream processing systems, as these systems get more ambitious and complicated as well.

Finally, we point out to a rift in the cloud computing fault model and recovery techniques, which motivates the need for more sophisticated recovery techniques. Traditionally the cloud computing model adopted the crash failure model, and managed to confine the faults within this model. In the cloud, it was feasible to use multiple nodes to redundantly store state, and easily substitute a stateless worker with another one as nodes are abundant and dispensable. However, recent surveys on the topic [29, 23] remark that more complex faults are starting to prevail in the clouds, and recovery techniques of restart, checkpoint-reset, and devops involved rollback and recovery are becoming inadequate.

1.1 Outline of the rest of the paper

To illustrate how the three design principles above are embraced in cloud computing systems, in Section 2 we present examples of services that power global-scale operations of Google and Facebook, and then delve in to the architectures of these services and their interactions with each other. We introduce the service-oriented-architecture (SOA) design pattern popular in the cloud computing software, and the microservices design pattern.

In Section 3, we review the literature on what types of faults occur in the cloud computing systems and what type of fault-tolerance and recovery mechanisms are employed to deal with these.

In Section 4, we point out to opportunities for applicability/adoption of self-stabilization techniques in new emerging topics in cloud computing areas, including in managing distributed coordination when composing multiple microservices to form higher level services (in Section 4.1), in dealing with complicated dataflow dependencies in realtime stream processing systems (in Section 4.2).
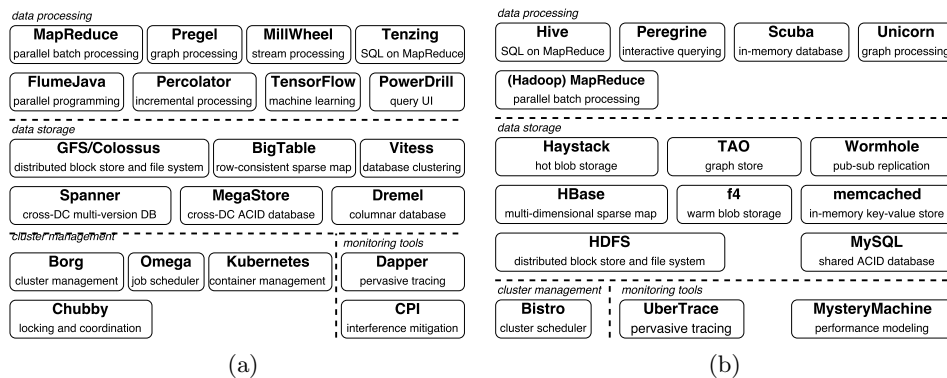
data processing

MapReduce	Pregel	MillWheel	Tenzing
parallel batch processing	graph processing	stream processing	SQL on MapReduce

FlumeJava	Percolator	TensorFlow	PowerDrill
parallel programming	incremental processing	machine learning	query UI

data storage

GFS/Colossus	BigTable	Vitess
distributed block store and file system	row-consistent sparse map	database clustering

Spanner	MegaStore	Dremel
cross-DC multi-version DB	cross-DC ACID database	columnar database

cluster management   **monitoring tools**

Borg	Omega	Kubernetes	Dapper
cluster management	job scheduler	container management	pervasive tracing

Chubby	CPI
locking and coordination	interference mitigation

(a)

data processing

Hive	Peregrine	Scuba	Unicorn
SQL on MapReduce	interactive querying	in-memory database	graph processing

(Hadoop) MapReduce
parallel batch processing

data storage

Haystack	TAO	Wormhole
hot blob storage	graph store	pub-sub replication

HBase	f4	memcached
multi-dimensional sparse map	warm blob storage	in-memory key-value store

HDFS	MySQL
distributed block store and file system	shared ACID database

cluster management   **monitoring tools**

Bistro	UberTrace	MysteryMachine
cluster scheduler	pervasive tracing	performance modeling

(b)

Fig. 1: (a) Google and (b) Facebook software infrastructures (redrawn from [33])

2 Cloud Computing Design Principles

2.1 Cloud Software Infrastructure

Internet scale applications such as search engine and social network demand a complicated software infrastructure with cluster management, data storage, data processing, and application layers. Each layer usually contains multiple systems to serve different use scenarios. Figures 1a and 1b illustrate the known systems at Google and Facebook. The cluster management layer allocates hardware resources and schedules jobs as a service provided to upper layers. Distributed monitoring systems help system admins to keep cluster healthy. There are several data storage systems that persist system state at scale and provide a rich interface with various consistency guarantees. Data processing layer implements required data analysis and representation for the applications.

Upon a closer examination, the distributed system within each layer often adopts a simple architecture pattern where the master node maintains the critical states and workers accepts tasks from the master and compute on data in a stateless manner. The master node relies on state-machine replication for fault-tolerance. Finally, these systems often employ ZooKeeper for coordination needs: including metadata management, group membership, leader election, and resource and configuration management. Cloud applications typically leverage such robust infrastructure for fault tolerance and recovery among other things.

2.2 Service Oriented Architecture (SOA) and Microservices

SOA is the most popular software design style in large-scale applications. In SOA, the functionality is divided into isolated services and each service can use other services through a communication protocol (e.g. RESTful) over a network. Compared to the more traditional monolithic architecture pattern, the resulting
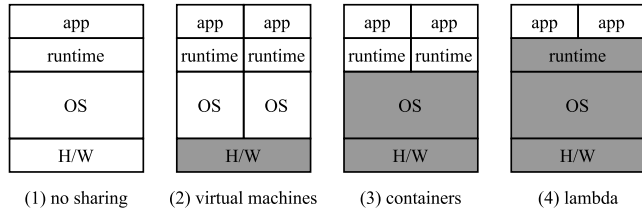
app		app	app		app	app		app	app
runtime		runtime	runtime		runtime	runtime		runtime	
OS		OS	OS		OS			OS	
H/W		H/W			H/W			H/W	

(1) no sharing   (2) virtual machines   (3) containers   (4) lambda

Fig. 2: Evolution of sharing (redrawn from [25])

microservices architecture pattern is more scalable through functional decomposition.

Every microservice in the system is a simple, independent, distinct feature or functionality of the application. Each service instance is typically a process running on its own VM or container. Each microservice instance maintains its own state by using the data storage systems abundant in cloud computing infrastructure. The interconnected microservices exposes a well defined REST API that is consumed by other services or clients. Representational state transfer (REST) is the most common choice for inter-service communication because it makes use of stateless protocol and standard operations (HTTP) for fast performance, reliability and the ability to grow/update without affecting the running system as a whole. Service API, therefore, is identified by its URL. Between any service requests, the client context is not stored on the server providing the service. As a result, each request from any client must contain all the information necessary to serve the request.

2.3 Towards Stateless Computing

The infrastructure to support the cloud application has been rapidly evolving, mostly in the opensource domain. The evolutionary trend is to increase the flexibility of cloud infrastructure while reducing the overheads via virtualization and resource sharing. Figure 2 compares these infrastructure approaches and their shared layers in gray.

The development of virtualization technologies and increased efficiency of Virtual Machines (VMs) gave rise to the proliferation of cloud computing applications. In the cloud, horizontal scalability is attained much easier by no longer requiring physical hardware to support applications. Many systems are able to run in virtual machines on the same hardware, making resource utilization higher and favoring horizontal scalability over vertical. This allowed applications to have smaller state and become more portable for the ease of quick deployment on new virtual machines.

Containerized environments took the resource sharing beyond the hardware. Containers are isolated application environments that share hardware and an underlying operating system [13]. They are very lightweight and can be brought up online or turned off in the matter of seconds. This demanded the applications

to be lightweight as well, and having large state became prohibitively expensive due to the costs of recovering the state after container migration or restart.

Finally, serverless or lambda [7] computing allows multiple applications to share not only the same operating system, but also the user-space libraries and runtimes. With serverless computing, the applications are defined as a set of functions or lambda-handlers that have access to some common data-store for state. The functions themselves, however, do not carry any state from one execution to another. Handlers are typically written in a portable, interpreted languages such as JavaScript or Python. By sharing the runtime environment across functions, the code specific to a particular application will typically be small. The code size, portability and stateless nature of handlers make them inexpensive to send to any worker in a cluster.

2.4 Coordination in the Cloud

Coordination plays an important role in the cloud applications, especially as the complexity of the systems grows. In our recent work [4], we surveyed the use of Paxos-like consensus systems used in various cloud computing systems. We identified 9 distinct usecases for the coordination as illustrated in Figure 3. The most common of these usecases are metadata management with 27% of the usage scenarios surveyed, leader election (20%), group membership (11%), and synchronization (11%).

Metadata management provides a mechanism for maintaining metadata, such as configuration or state of shared objects, across the nodes in the cluster in a consistent and fault-tolerant manner. The leader election usecase allows systems to safely elect a single node acting as a dedicated master for some operations. Synchronization is another important use case that enables distributed locks. Other use cases for consensus systems in the cloud include server and log replication, barrier orchestration, service discovery and distributed queues.

3 Cloud Computing Faults and Recovery

3.1 Cloud Computing Faults

Failures in the cloud can be roughly categorized as small-scale faults and large-scale failures. Small faults typically only impact a few clients or some specific requests, allowing the system to process the rest of the workload. For instance, an example of such small fault would be a rejected hotel booking request due to the insufficient funds on the card. In this case, the fault does not have any impact on other users or requests, yet the system has to take a corrective action to make sure the room is not marked as reserved. The credit card failure left the booking service in a corrupted state as the room was reserved without a payment. However, this corruption is localized to a single service and not likely to spread to other services, making the room unavailability to be the only adversarial effect of the fault if left without a correction.
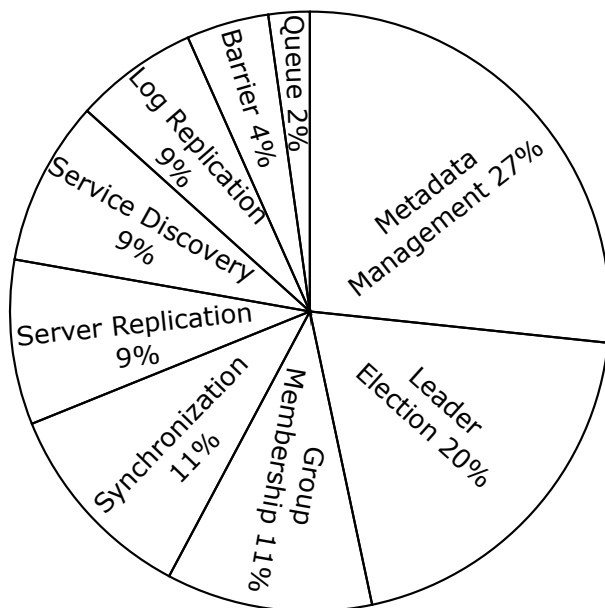
Fig. 3: Consensus systems usage in the cloud (redrawn from [4])

Crash fault approach to fault tolerance is crucial to many of such small and localized failures. If the node failure did not impact any global state of the systems, such as persisting some corrupted data to storage, then a failure can be dealt with a simple restart or provisioning of a new node. In some cases, even if the failure is persisted to storage, it can be masked through replication and redundancy in the cloud.

More intricate small-scale failures may corrupt a state in a way that allows such corruption to propagate from one component to another. In such non-localized corruption scenarios, correcting state is more difficult, since the engineers need to account for the possibility of re-corruption cycles [28]. The spread of corruption across the components of the cloud system can lead to a catastrophic failure.

Large-scale failures are faults that leave the system unavailable for significant number of users or requests. Gunawi et al. conducted a survey of catastrophic failures across various cloud applications and identified the most common reasons cloud systems crash [23] as illustrated in Table 1. According to their study of 597 failure reports from different systems, only 15% of the failures are the result of some bugs in the code, while another 10% are due to the problems in configuration. Upgrade failures, which are bugs and misconfigurations introduced during the system upgrade and maintenance account for another 16% of failures.

Table 1: Causes of Catastrophic Failures in the cloud [23]

Root Cause	# of Services	# of Occurances	% of Occurances
Unknown	29	355	-
Upgrade	18	54	16
Network	21	52	15
Bugs	18	51	15
Misconfiguration	19	34	10
Traffic Load	18	31	9
Cross-Service Dependencies	14	28	8
Power Outages	11	21	6
Security Attacks	9	17	5
Other Human Errors	11	14	4
Storage Failure	4	13	4
Server Failure	6	11	3
Natural Disasters	5	9	3
Other Hardware Failures	4	5	1

Many of the other failures are subject to the external events, such as security attacks, natural disasters or failure of outside dependencies

Cross-service dependencies are a special category of external failures originated at some service outside of the affected cloud system. These failures show that a reliability of a cloud application depends not only on the reliability of internal infrastructure and code, but also on the services external to the application. This is especially important in the context of microservices architecture, since a single unreliable microservice can cause cascading failures on its dependents.

3.2 Recovery Models

Recovery by Waiting. Recovery by waiting is typically used in cases of outside cross-service dependency failures [23]. In these situations, the cloud application and infrastructure is capable of proper operation, but fails due to the unavailability of an external service, and must wait for the outside resources to become available again. As a remedy to the waiting for outside recovery, cloud applications need to design for redundancy on the external dependencies they use.

Recovery by Restart. Since many of the cloud components are either stateless or have small state that can be quickly replicated and loaded to new nodes, restarting the failed or slow nodes became a common practice in the cloud systems. Cloud systems are carefully guarded against server failures, making

the recovery by restart an easy and safe option for handling many faults. This led to the fact that many smaller scale problems, such as memory leaks or re-optimizations done as a results of shifting workloads, are addressed by restarting the nodes instead of fixing the code or introducing additional complexity to it [8]. Some cloud applications constantly induce server crashes in their infrastructure to probe the system and make sure it can handle a larger-scale faults [12].

However, recovery by restart is not able to address all failures happening in the cloud systems. Some failures are not caused by problems in the software, while some more intricate problems may involve distributed state corruption that can persist after individual node restarts.

Recovery by Checkpoint Reset. Checkpoint-reset is also a popular way to bring the system back to the correct state after the fault. These type of recovery causes the system to lose some progress and restart from some older known safe checkpoint. Because of this drawback, this type of recovery cannot be used with systems providing real-time feedback to the users. However, checkpoint-reset is a popular approach for data-processing [16, 31] and distributed machine learning [2, 39] applications.

Recovery by Application Rollback. Failures during the system upgrade are the single largest cause of catastrophic failures in the cloud systems [23]. These failures often happen due to the bugs or misconfigurations introduced with application upgrades. The natural way to remedy such faults is rolling back to the previous stable version of the application and configuration.

Large applications, often perform gradual application upgrade or configuration rollout to the production system to minimize the impact possible bugs may bring to the systems. The idea is to catch the problem before it was deployed full-scale, thus both minimizing the severity of problem and time to roll-back. For instance, Facebook's Configerator tool controls how new configurations roll-out from testing to the production environment and how the new configuration gradually expands to cover more production nodes [38]. Configerator monitors the system behavior with every increase of new configuration's coverage and can automatically rollback to the previous good configuration if the problem is detected during the rollout.

Recovery by State Repair. An application state can be corrupted due to bugs in the code or configuration, however not every cloud application can tolerate being reset to a prior checkpoint or some predetermined safe state. For such application, engineers must design a corrector [5] that fix the application state when a corruption occurs. Many severe bugs are capable of corrupting the state of the application. For instance, Leesatapornwongsa et al. compiled the database of known distributed concurrency (DC) bugs, one of the most complicated bug types in distributed systems from popular cloud-scale applications [29]. The authors also analyzed the fixes that were devised to address these issues in the code.

Many DC bugs in [29] were fixed in an offline manner by patching the code to prevent the faulty execution from occurring, but some more intricate bugs could not be addressed in such manner. Unlike the preventable bugs caused by bad
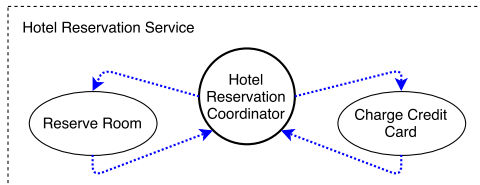
Fig. 4: An example of a monolithic hotel reservation service.

message timing or incorrect timing assumptions, the more difficult DC bugs were caused by unfortunate and uncontrollable timing of the component failures in the cloud applications. Instead, such bugs were fixed by adding a simple corrector to the system that forward corrects the global state to a consistent state while the system is running.

Offline Recovery. Many of the DC bug fixes mentioned earlier operated at the code level, preventing the problem from occurring in the fixed version. These fixes happen offline, as the application needs to go through a development cycle of making a fix, testing and deployment. In other words, the engineers develop the correctors in an ad-hoc manner on a case-by-case basis.

4 Future Directions for Stabilization in the Cloud

4.1 Distributed Coordination at Application Level

Cloud applications are often build on top of frameworks and architectures that hide the complexity of distributed systems. For example, MapReduce provides powerful batch-processing capabilities while keeping a simple API and allowing the application developer to write code as if they were developing a single-threaded application [16]. Distributed transactions accomplish similar goals, as they allow developers to think of distributed and concurrent requests as sequential operations on a single machine.

A transaction updates the state of multiple remote nodes or components in the cloud application in such a way that either all components are successful at changing their state or none. For example, consider our earlier hotel reservation application that perform multiple actions for the booking to be complete. The application consists of a coordinator and two components: room reservation and credit card processing, as shown in the Figure 4. With the help of transactions, the application can tolerate component failures without the global state being affected. For instance, if the room reservation component successfully marks the room as reserved, but the credit card processing fails, then the system will abort the transaction and no room-reservation will be preserved in the state of the application. Traditionally, transactional systems [15, 3] employ atomic commit protocols, such as two-phase commit (2PC), to perform transactions. However, 2PC is regarded as slow protocol with a number of corner cases that affect its performance and availability [14, 11].
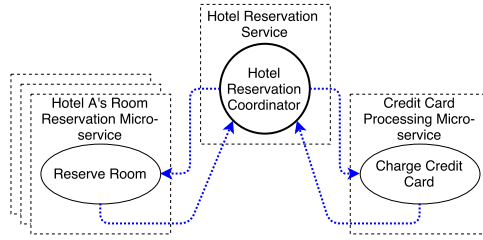
Fig. 5: An example of a micro-service hotel reservation system.

Transactions do not fit well in the geodistributed heterogenous cloud systems due to performance reasons [24]. Moreover, developers cannot always rely on having a common coordination tool in microservice designs, because some of the services may be external to the application and maintained by another party. Returning to our hotel booking example, every component of the reservation service had little knowledge of other components' states, but at least these components were part of the same infrastructure and could coordinate. Now consider an example in which the hotel reservation system needs to interact with different external booking microservices for each of the hotel brands, as illustrated in Figure 5. An external microservice can mutate its own state, such as booking a room or canceling the reservation, but it cannot participate in the internal transactions, since the infrastructure and protocols to coordinate between the microservices are lacking.

Despite the fact that transactional solutions do not work well with microservices architectures especially in the presence of outside actors, we often need to provide some of the transactional guarantees to such cloud systems. In particular, if one of the microservices in the request is not successful, we need to revert the state of the microservices that have already changed their states. We illustrate this process in the Figure 6. These corrective actions are typically written at the coordinator layer of the application in an ad-hoc manner and are not enforced by some specialized protocol. With this corrective approach, engineers need to provide undo-actions for the anticipated failures. For instance, if the room booking micro-service has already reserved the room when the credit card processing micro-service failed to charge the credit card, the coordinator needs to issue a corrective action to the booking microservice and cancel the reservation in order to bring the overall global system in a consistent state.

Self-stabilization can be useful in these scenarios, where distributed microservices need to be coordinated and kept eventually consistent to the face of failures. On the other hand, in order to become applicable for large scale cloud systems, self-stabilization methodology should allow certain compromises. For instance, recovering to an approximate invariant can be supported. Another example would be to design stabilization in a limited/specialized framework written to handle distributed coordination, such as distributed sagas [30].
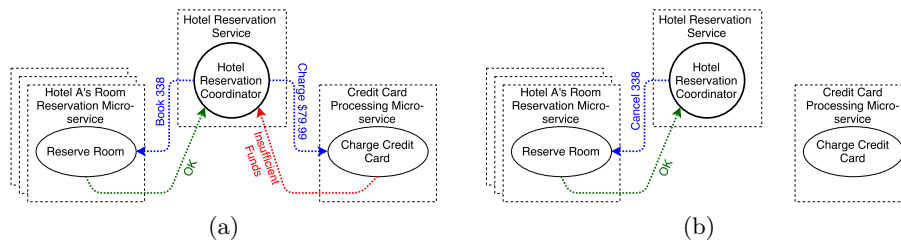
Fig. 6: Hotel reservation example with corrective actions. (a) Credit card micro-service is not successful at charging the card. (b) Correcting the state of room-booking micro-service.

4.2 Stabilization in DataFlow Systems

Modern data-processing [31, 27] and distributed machine learning systems [2] often represent the computation as a graph with data flowing along the edges and vertices representing the computations. These systems tend to utilize checkpoint-reset as their failure recovery model, however this may be inadequate for online stream processing systems, such as Twitters Heron [27]. Under checkpoint reset, an online stream-processing system would lose all computations happening after the checkpoint in case of a recovery. Heron runs on a large number of small-state workers that receive their tasks from a centralized streaming manager and utilizes recovery by restart model for many faults, as the small tasks can be rescheduled to different workers as needed.

Dhalion is a system that aims to automate many of the actions undertaken by Herons engineers for fault recovery and performance tuning [19]. Dhalion achieves this through monitoring of many vital parameters and using the moni-toring data to detect abnormalities in operation and come up with a set of pos-sible causes for abnormal behavior. Dhalion uses the probable causes to identify the best possible fix out of the ones available to it and apply that correction to the systems, bringing it to a healthier state. The systems attempts to resolve the problems at the higher, infrastructure level instead of the application-logic level. For instance, if Dhalion detects the performance is inadequate, it will attempt to scale up by adding more workers. Similarly, if Dhalion sees a faulty worker, it will attempt moving the tasks away from that worker into a healthy one.

5 Concluding remarks

In this paper, we presented the design principles that contribute most to the high-availability of cloud computing services, and reviewed the type of fault-tolerance and recovery mechanisms employed by the cloud computing systems. We argued that since cloud computing systems use infrastructure support to keep things simple and reduce the need for sophisticated design of fault-tolerance mechanisms, self-stabilization has not been prominent in the cloud. Finally, we

identified recent trends in cloud computing that motivate the design of more sophisticated fault-tolerance mechanisms/frameworks, namely the implementation of distributed coordination over microservices, the development of sophisticated realtime dataflow processing systems, and the increased frequency of intricate failures due to complicated interactions of services.

In order to adopt self-stabilization to address availability issues in cloud, we believe that it is important to adhere to the simplicity and "infrastructure-first" principles in the cloud computing systems. We anticipate that the resulting solutions would make compromises to be "practically self-stabilizing"[6] rather than "traditionally/formally self-stabilizing". For example, while "arbitrary state corruption" model is interesting theoretically, it is infeasible to realize at the implementation level, and it is overkill in that the return on investment is unjustified. We belive that there is a need to define restricted corruption models and healing techniques that leverage existing cloud infrastructure. One may consider allowing only noncritical variable corruption as in [22] by leveraging the cloud infrastructure (such as ZooKeeper and key-value stores) to protect precious variables. Or an alternate model is to allow only specification/API-level variable corruption (reasoning those are the state that matter) and deal with implementation state corruption by leveraging cloud infrastructure.

References

1. Apache cassandra. `http://cassandra.apache.org/`, 2017.
2. ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.
3. AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 159–174.
4. AILIJIANG, A., CHARAPKO, A., AND DEMIRBAS, M. Consensus in the cloud: Paxos systems demystified. In *Computer Communication and Networks (ICCCN), 2016 25th International Conference on* (2016), IEEE, pp. 1–10.
5. ARORA, A., AND KULKARNI, S. S. Detectors and correctors: A theory of fault-tolerance components. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on* (1998), IEEE, pp. 436–443.
6. ARORA, A., AND WANG, Y.-M. Practical self-stabilization for tolerating unanticipated faults in networked systems. Tech. rep., The Ohio State University, 2003.
7. Aws lambda. `https://aws.amazon.com/lambda/`, 2017.
8. BARR, J., NARIN, A., AND VARIA, J. Building fault-tolerant applications on aws. *Amazon Web Services* (2011), 1–15.
9. BIRMAN, K., CHOCKLER, G., AND VAN RENESSE, R. Toward a cloud computing research agenda. *ACM SIGACt News 40*, 2 (2009), 68–80.
10. BREWER, E. Towards robust distributed systems. In *Proceedings of PODC* (2000), p. 7.
11. CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

12. Netflix chaos monkey. `https://github.com/Netflix/chaosmonkey`, 2017.

13. What are containers? `https://aws.amazon.com/containers/`, 2017.

14. Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

15. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

16. Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

17. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

18. Dolev, S. *Self-Stabilization*. MIT Press, 2000.

19. Floratou, A., Agrawal, A., Graham, B., Rao, S., and Ramasamy, K. Dhalion: Self-regulating stream processing in heron. In *Proceedings of the 2017 VLDB Endowment* (2017), vol. 10.

20. Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News 33*, 2 (2002), 51–59.

21. Gilbert, S., and Lynch, N. Perspectives on the cap theorem. *Computer 45*, 2 (2012), 30–36.

22. Gouda, M. G. Elements of security: Closure, convergence, and protection. *Information Processing Letters 77*, 2 (2001), 109–114.

23. Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., and Eliazar, K. J. Why does the cloud stop computing? lessons from hundreds of service outages. In *SoCC* (2016), pp. 1–16.

24. Helland, P. Life beyond distributed transactions: an apostate's opinion. In *CIDR* (2007), vol. 2007, pp. 132–141.

25. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with open-lambda. *Elastic 60* (2016), 80.

26. Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference* (2010), vol. 8, p. 9.

27. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 239–250.

28. Leal, W., and Arora, A. Scalable self-stabilization via composition. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on* (2004), IEEE, pp. 12–21.

29. Leesatapornwongsa, T., Lukman, J. F., Lu, S., and Gunawi, H. S. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, pp. 517–530.

30. McAffery, C. Talk on distributed sagas. `https://www.youtube.com/watch?v=1H6tounpnG8`, 2017.

31. MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

32. NEWCOMBE, C. Why amazon chose tla+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* (2014), Springer, pp. 25–39.

33. SCHWARZKOPF, M. Operating system support for warehouse-scale computing. *PhD. University of Cambridge* (2015).

34. SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400.

35. Shard (database architecture). `https://en.wikipedia.org/wiki/Shard_(database_architecture)`, 2017.

36. SMITH, J. E., AND NAIR, R. The architecture of virtual machines. *Computer 38*, 5 (2005), 32–38.

37. Chapter 1: Service oriented architecture (soa). `https://msdn.microsoft.com/en-us/library/bb833022.aspx`, 2017.

38. TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 328–343.

39. XING, E. P., HO, Q., DAI, W., KIM, J. K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data 1*, 2 (2015), 49–67.