Consistent Disaster Recovery for Microservices: the BAC Theorem

Guy Pardon
Atomikos

Mechelen, Belgium

Cesare Pautasso
Università della Svizzera Italiana, Lugano, Switzerland

Olaf Zimmermann
Hochschule für Technik Rapperswil (HSR FHO), Switzerland

How do you back up a microservice? You dump its database. But how do you back up an entire application decomposed into microservices? In this article, we discuss the tradeoff between the availability and consistency of a microservice-based architecture when a backup of the entire application is being performed. We demonstrate that service designers have to select two out of three qualities: backup, availability, and/or consistency (BAC). Service designers must also consider how to deal with consequences such as broken links, orphan state, and missing state.

Microservices are about the design of fine-grained services, which can be developed and operated by independent teams, ensuring that an architecture can organically grow and rapidly evolve.[1] By definition, each microservice is independently deployable and scalable; each stateful one relies on its own ("polyglot") persistent storage mechanism. Integration at the database layer is not recommended, because it introduces coupling between the data representation internally used by multiple microservices. Instead, microservices should interact only through well-defined APIs, which—following the REST architectural style[2]—provide a clear mechanism for managing the state of the resources exposed by each microservice. Relationships between related entities are implemented using hypermedia,[3] so that representations retrieved from one microservice API can include links to other entities found on other microservice APIs. While there is no guarantee that a link retrieved from one microservice will point to a valid URL served by another, a basic notion of consistency can be introduced for the microservice-based application, requiring that such references can always be resolved, thus avoiding broken links. As the scale of the

system grows, such a guarantee can be gradually weakened, as is currently the case for the World Wide Web.

Microservice architectures are designed to survive individual microservice failures and to prevent cascading failures of all services.[4] Backing up and recovering a microservice is an important operation, which allows it to survive significant failures affecting its operating environment.[5] For example, if one node of a cluster crashes, it should be possible to restart the microservice on another node and connect it to the same database so that it can access its corresponding state. If the database server itself crashes, it is possible to recover the database from previously executed dumps that are stored elsewhere. Backups can be performed from within the database or using a backup data pump, which periodically and incrementally synchronizes replicated databases.

This article is concerned with the problem of backing up an entire microservice architecture where a running application decomposed into multiple microservices needs to be backed up so that it can be recovered after a disaster strikes. How should one do so and still guarantee a certain level of consistency between the various microservices? How should one do so without affecting the availability of the application? In particular, we are concerned with the full availability of the application, during which it is possible to both read its state and perform updates to any of its microservices. Depending on the chosen type of backup technique, it might be necessary to suspend normal operation and only allow for performing read operations, to ensure that the underlying state of each microservice is being backed up consistently.

EVENTUAL INCONSISTENCY

Following a domain-driven design methodology,[6] a monolithic application (see the left image in Figure 1) may be split into different Bounded Contexts. This practice is beneficial to ensure the internal semantic consistency of each domain context, as well as to enable the interoperability of data exchanged between separate domains across a Context Map. This method is also recommended when splitting up the monolith into multiple microservices (see the right image in Figure 1) that remain logically connected by a set of loose relationships between their data models. For example, a sales-related application is split into one microservice to handle customers, one to manage the product catalog and warehouse inventory, another to manage orders, and yet another to handle the shipping process. Clearly, a shipment must refer to a given customer and to a specific order, which should refer to the actual products (see Figure 2).

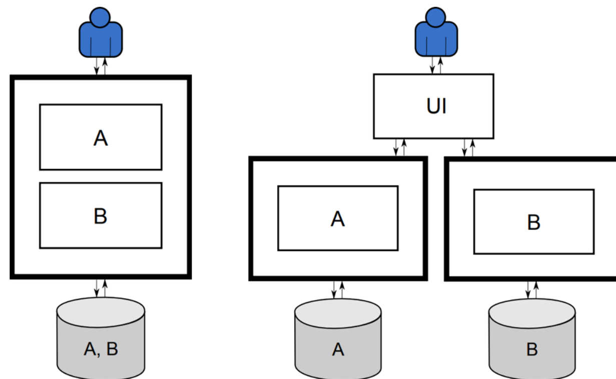

Figure 1: (Left) monolithic architecture and (right) microservice architecture.

While normal operations will lead to the eventual consistency[7–8] of the state of the application partitioned across multiple microservices, when disaster strikes, the state of the recovered application will eventually become inconsistent due to snapshot timing issues.

Even if every backup of individual microservices can be trusted for their independent recovery, as a whole, it is likely that the state of the application will not converge and will remain inconsistent when each microservice has been backed up and recovered independently. For example, it

might be problematic if after the independent recovery of each microservice some orders would refer to missing product descriptions or some shipments would point to customers no longer present in the corresponding recovered microservice.

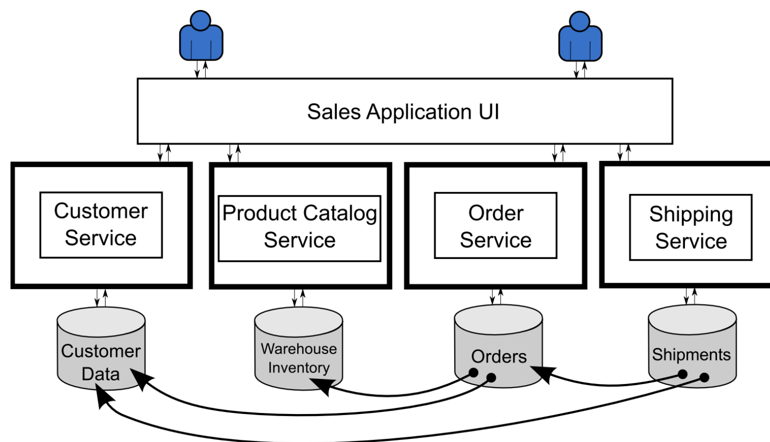

Figure 2: An example microservice architecture with references spanning across individual microservices.

To avoid this, it is important that the snapshot taken to back up every microservice is taken not necessarily at the same time, but when the overall application state was consistent. While the backup is underway, clients may not be allowed to perform arbitrary update operations spanning multiple microservices. Depending on the size of the snapshot and whether a full backup or an incremental backup technique is performed, this might require a non-negligible amount of time.

Alternatively, if introducing extra data loss during recovery is not a problem, it would be possible to achieve consistency across all microservices by rolling back the state of every microservice to the point in time when the oldest backup of one microservice was taken. This is not acceptable in many application scenarios and contexts.

THE BAC THEOREM

In this article, we present the BAC theorem for microservices, which states:

> *When backing up an entire microservices architecture, it is not possible to have both availability and consistency.*

The theorem is inspired by the CAP theorem,[9] trading off consistency against availability in distributed (replicated) databases. The informal proof is similar. If we allow each microservice to evolve its state independently and perform a backup of every microservice at a different time, the whole set of backups will not offer a consistent view over the state of the whole application. In case of recovery, each of its microservices will be restarted from a snapshot taken independently, and, thus, it is possible that related entities between microservices will be missing. As opposed to normal operation when this is allowed to happen because the various microservices will eventually reach a consistent state,[10] in case of recovery, this will not happen because the clients performing distributed transactions[11] over the microservices will be long gone.

The alternative to ensure consistency would be to lock the entire application during the backup and make sure that the state of its microservices is snapshotted at the same time. Locking the entire application would compromise its full availability, because only read operations would be allowed while backing up. Depending on the duration of the backup (which is bound by the slowest microservice), this might affect the clients whose requests to change the state of some microservice might have to be denied or delayed until the backup of the entire application is completed.

Conceptual Model: Commands and Events

Our abstract model of a microservice is inspired by domain-driven design (DDD),[6] which introduces patterns such as Service, Aggregate, and Domain Event to decompose the business logic of an application into discrete, loosely coupled processing units, each of which has a single reason to change.[12]

A microservice accepts "commands"—either through user input or by processing "domain events" from other microservices. Commands can fail due to validation or integrity constraints by the receiving service. When processed successfully, commands can result in the publication of one or more events reflecting the outcome of the command. Events do not "fail" because they are merely notifications of things that already happened. Events can be lost, though, in particular (for this article) due to recovery from an obsolete backup. A microservice that receives an event can convert it into a command for local processing. Such a command in turn can fail (in particular if the microservice does not understand the event after disaster recovery).

Microservices refer to each other's data through loosely coupled references. If microservices expose a RESTful HTTP API, these references naturally correspond to resource identifiers. Each data item is owned by exactly one microservice and referenced by any number of other microservices.

Without loss of generality, microservices are using the event storage pattern[13] for managing changes applied to their state. The database of the microservice thus contains a log of business events, which can be replayed to bring the state of the microservice to its latest version, while keeping track of its history for auditing and debugging purposes.

Some of the events in the log are local (such as the creation of a new customer record). Some events conceptually cross the boundary between the two microservices. For example, when a customer orders a new shipment, an event will be added to the shipment microservice (the shipment order) and another will be added to the customer microservice (the reference to the shipment order in the customer shipment collection). More generally, in this case, there is an interaction between microservices (the request to create a new shipment order for a given customer), which results in a state transition for both of them. Thus, a new event is added to the log of each microservice.

Eventual Inconsistency with Full Availability

When making an independent backup of the state (the event log) of each service, it is possible that one service will be backed up before the event is added to the log, while the other service will be backed up after the corresponding event is added to its log. When restoring the services from backup, only one of the two services will recover its complete log. The global state of the microservice architecture will thus become inconsistent after restoring it from backup. In a similar way, during the execution of a particular use case, it is possible that one service is backed up before a command is executed in it, while another service is backed up after the corresponding command (of the same use case) is executed in it.

In a hypermedia context, the inconsistency typically manifests itself after the recovery from backup of the services in the following ways:

- *Broken link*. This is when the reference can no longer be followed—for example, where, using database terminology, foreign key records are recovered without the corresponding primary key records. In Figure 3, the up-to-date Microservice A refers to an obsolete version of the other Microservice B, where the referenced entity cannot be found after it has been recovered. The inconsistency can thus be detected when clients try to follow the reference from A to B and instead find that the link is broken.
- *Orphan state*. This is when there is no reference to be followed—for example, where primary key records are recovered without the corresponding foreign key records. In Figure 4, the state of the up-to-date Microservice A is no longer referenced from the state of the Microservice B recovered from an obsolete backup. This situation might be more difficult to detect from clients, because there are no immediately visible inconsist-

encies when they interact with either service. However, this might introduce storage space leaks for A, if there is no garbage-collection mechanism for such orphan states. Moreover, duplicate event log entries for A might be introduced (which might lead to unwanted side effects), as the obsolete Microservice B is brought up to date.

In the general case, we call this phenomenon "*missing state*" – an example of which is shown in Figure 5: Microservice B remains consistent with A until it crashes. After recovery from an obsolete backup, B does not have the state corresponding to the latest events logged by A.

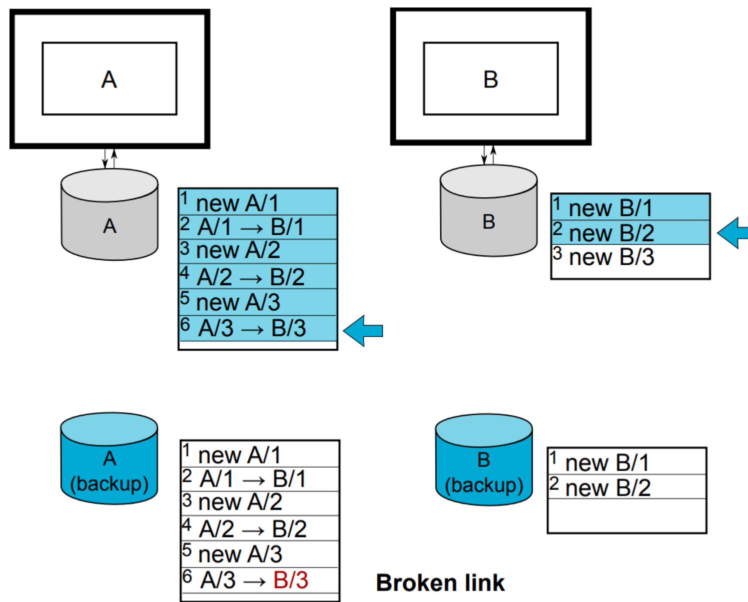

Figure 3: Broken link. The event logs tracking the state updates of each microservice are backed up independently at different times. The colored arrows indicate the points in time in which the backup snapshot of each log was taken. The colored events are the ones that have been copied to the backup. A/3 → B/3 shows that there is a reference from A to B. When the state of Microservice B is recovered, it will be inconsistent with respect to Microservice A referring to it: B/3 is no longer part of the recovered state of service B, hence the broken link.

Consistent Backups with Limited Availability

It becomes possible to avoid the inconsistency by coordinating the backup of the two services to ensure that a snapshot is taken either before the interaction takes place or after the effects of the interaction have been logged on both sides. This means that, while the backup is running, no events can be added to the microservice logs, effectively reducing the availability of the microservices for clients that need to perform some state transitions/write operations.

This, however, violates the independence of the two microservice disaster recovery processes, as it introduces tight coupling into their operational lifecycles. While this might be possible to achieve for a small number of microservices, chances are that it will become impractical as the number of related microservices grows. The mere number of microservices is not problematic as such, but things get difficult if many or all of them have relationships with each other. The semantic complexity of the domain and the chosen microservice decomposition granularity will avoid or exacerbate the problem.

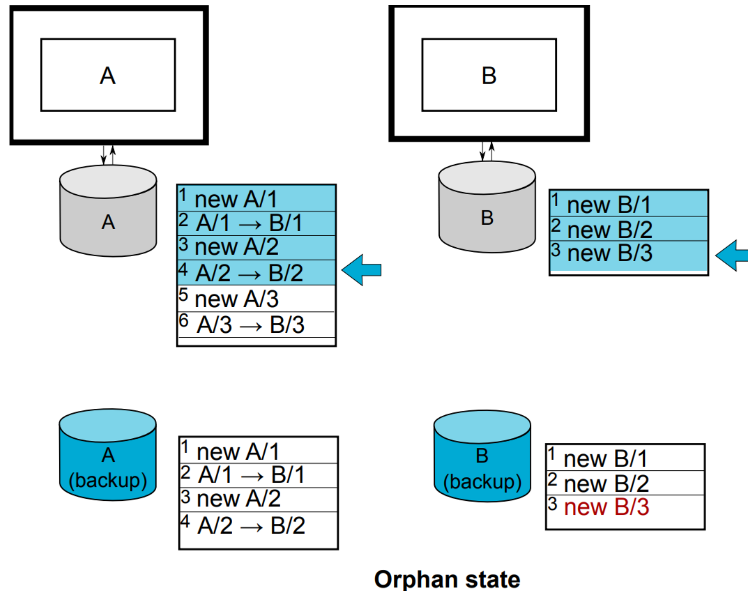

Figure 4: Orphan state. The event logs are backed up independently at different times, indicated by the coloring. When the state of Microservice B is recovered, it will be inconsistent with respect to Microservice A, which is no longer referring to it. In the figure, the reference A/3 → B/3 is not backed up. So, after recovery, B/3 is orphaned, as there is no longer a reference to it from A.

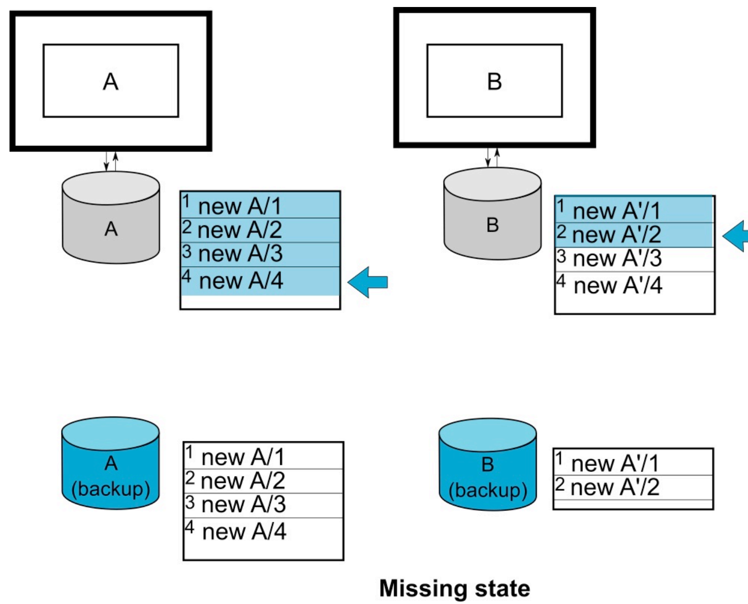

Figure 5: Missing state. A and A’ are service-specific representations of related entities affected by the same events or commands. The event logs are backed up independently at different times (shown in color). When the state of Microservice B is recovered, it will be inconsistent with respect to Microservice A, as the entities A’/3, A’/4 corresponding to A/3, A/4 are missing.

DEALING WITH THE CONSEQUENCES OF THE BAC THEOREM

How should one design microservices that are aware of the consequences of the BAC theorem and can thus deal with the possible eventual inconsistency introduced by their independent back-up and recovery?

Dealing with Broken Links

Broken links in A after a restore (with old data) of B are equivalent to long-duration unavailability of B during normal operation. Typical strategies include:

1. *Reconstructing B to contain a valid reference*. A human (possibly the user) could be asked to restore the state of B to a correct one, including the missing reference. Until reconstruction happens, A will have to tolerate the missing reference (which it has to do during normal operations anyway, when temporary network glitches cause the unavailability of B). In some cases, reconstruction of B might not be possible. For instance, if B was a reservation of a scarce resource (like an airplane seat on a flight), the flight might be fully booked in the meantime. Restoring a reservation system with data loss probably results in overbooking, a common source of headaches for most airlines.
2. *Doing nothing*. The system and its users can accept that some parts of the system can be inconsistent. This strategy can be acceptable if the reference to B is only there for informational purposes (not really needed for A to function). As the scale of the application grows larger with an increasingly large number of microservices, users might not expect full consistency at all times. Business criticality and quality attributes such as accuracy and auditability, as well as related compliance controls, are key decision drivers here.
3. *Caching*. Microservice A could cache whatever data it needs from B, so that it has at least something to work with. Caching can work through either batch exports and imports, storing events from B in a cache at A, or fetching all relevant data from B when A establishes the reference.

When processing user commands, Strategies 1 and 3 seem like reasonable options: either the user rectifies any problems, or the system uses whatever it remembers in its cache to proceed. But what about processing commands that result from incoming events? In other words, what does a service do if it receives events and detects broken links? In that case, Strategy 1 is not very practical because the user is typically not involved in event processing. Instead, the system could use Strategy 3 to have the missing data cached, or it could process with warnings and resort to Strategy 1 at a later time when a user logs in. Lastly, the system could ignore any missing data, using Strategy 2. In any case, broken links should be logged as they are detected so that operators can decide which strategy to follow when attempting to restore the consistency of the application during or after disaster recovery.

Dealing with Orphan State

Orphan state may or may not be problematic, depending on the application. Some form of human intervention is required to decide what to do with orphan state.

When processing user commands, orphans can be detected, as some microservices might already store information provided by the user. Thus, a solution could be to have the user interface or an API Gateway assume responsibility for flagging orphan candidates so that the users of the system can easily fix any anomalies. It is recommended to regularly run a validation service and/or database consistency checker.

What about processing commands that result from incoming events? Here, orphan state might cause validation errors and, consequently, the rejection of the incoming event. This is probably not desirable because orphan state should not disturb the normal system operation after a restore from backup. A lot of events can come in in high-volume systems—rejecting incoming events could cause a spike of errors or warnings that will be difficult to deal with in practice. A safer

strategy would be to treat incoming events as the latest version of the truth (which, in a way, they are) and ignore validation errors when processing incoming events. The careful developer will place a warning in the logs when this happens.

Dealing with Missing State

If events are easy to reproduce, B might simply ask for a replay of missed events. This requires some authoritative source where B can ask.

If events are not easy to reproduce from their source, hopefully the event delivery infrastructure (if any) offers some form of reliability. If not, manual intervention in B might be needed to (re)apply the missing state in the form of one or more user commands.

In all other cases, the only option might very well be to accept that events can be lost.

DISCUSSION

Avoiding the Problem

When the challenges posed by BAC are too hard to solve, avoiding the problem can be your only option: make sure your backups are consistent. If you want to benefit from "referential integrity" (in the loose sense of the word), place related data in the same database (possibly using separate schemas) so they are backed up together. Hence, your microservices should be merged together as far as their state management is concerned (see Figure 6).

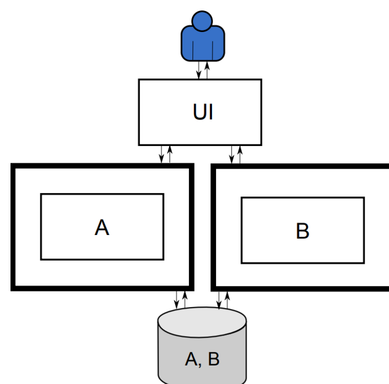

Figure 6: Microservice architecture with shared database, trivially enabling consistent recovery but introducing undesired coupling between the microservices.

Acknowledging the Problem

An alternative strategy might simply be to be aware of the problem and the resulting data losses or eventual inconsistencies. For some applications, this can be an acceptable option. This strategy must be carefully chosen, documented, and approved; at runtime, suited log entries should be created, and warnings should be sent to system administrators if justified in the given application scenario.

Let the Business Guide You

Microservices can be modelled around the existing departments and value streams of the business. Each business department is responsible for its own data. The proven communication between and autonomy of these departments can hint at how to ensure that your microservices

are sufficiently decoupled to work in practice.[12] In addition, the anomalies discussed in this article are probably known (indirectly) by the business already; chances are that appropriate policies exist already to take care of them.

The Architect's Job

We've discussed a few options for dealing with BAC. There is no one-size-fits-all solution. The architect's job is to decide which options work best for which set of microservices, especially in light of introducing an appropriate disaster recovery plan, dealing with how to perform backups (independently for individual microservices or globally for the entire system), and dealing with the consequences of restoring a system from potentially inconsistent backups (broken links, orphan state, and missing state). Thinking about these concerns before actually building your microservice ecosystem can save a lot of headaches afterwards.

RELATED WORK

High Availability and Replication

When microservices rely on a highly available, replicated database,[14] or when they are built as a replicated state machine,[15] one could argue that there is no need for an additional backup-and-recovery solution. Because there should always be $N > 1$ replicas of the state (possibly stored in databases distributed across different locations, availability zones, or datacenters), the backup process is happening continuously, and the recovery can theoretically be instantaneous.

Introducing a high-availability data layer for every microservice is, however, an expensive solution that greatly increases the operational complexity of the system. Likewise, even if microservices design principles married with DevOps and continuous integration practices appear to hint towards a world made of unstoppable systems whose services are in permanent operation, the question is whether it is wise to bet there will never be a need for recovering such systems from scratch. In general, the steep investment into a dedicated, highly available database solution with full replication to support every microservice might turn out to be difficult to justify. This ultimately needs to be compared against the cost of dealing with the eventual inconsistency of the application after its recovery.

Distributed Snapshots

Distributed snapshot algorithms[16] are also applicable in our scenario for the global state detection and check-pointing of the entire architecture. Without the need for sharing a common clock or accessing shared memory, it is possible to coordinate the various microservices as they store their state into a snapshot for backup purposes. Additionally, the distributed snapshot algorithm is not supposed to interfere with the ongoing distributed computation, which would fit with the need to preserve the full availability of the microservices.

However, the assumption of message-based, asynchronous communication over error-free, order-preserving channels whose content can also be snapshotted only partially fits with existing microservice architectures, where interactions are implemented using both message queues and the synchronous HTTP protocol. Likewise, ensuring that all microservices are designed to comply with Lamport and Chandy's requirements[16] would strongly couple all microservices and limit the flexibility in their independent deployment, operation, and evolution.

Interaction Contracts

Going beyond traditional database recovery, Barga, Lomet, and Weikum proposed a comprehensive form of recovery for multi-tier applications with communicating components.[17] The approach was based on interaction contracts between persistent components assumed to behave piecewise deterministically. This way, it is possible to recover the state of a component, starting

from a known initial state (such as when the component was deployed) by replaying every message it received in the same order they reached the component before the crash. This approach is, hence, also based on the notion of restoring a consistent state by replaying messages that would carry some form of commands and/or events in our frame of reference.

Other Service Design Issues and Coupling Criteria

Many architectural decisions and other hard design problems in service-oriented systems implemented with microservices are identified and partially answered in C. Pautasso et al.[12] How to handle backups remains an open problem according to the literature on microservice design in industry and academia.

Loose coupling has many dimensions, including time, location, platform, and format.[18] The Service Cutter methods and tool suggests microservices decompositions and re-compositions based on 16 functional and non-functional coupling criteria.[19] These criteria can help decide for one or more of the strategies for dealing with the consequences of the BAC theorem that we presented in this article. Backup requirements and the BAC theorem can also be seen as an additional coupling criterion.

CONCLUSION

While splitting the monolith is the mantra of microservice architectures, it is important to realize that the granularity of the result depends on what you are trying to achieve. Increased development velocity can be obtained with many loosely coupled development teams, which can deploy new versions of their microservice at will. However, the long-term sustainability of the application might be harmed if, in case of disaster, it will be impossible to achieve a holistic recovery that brings back all microservices in a globally consistent state. Thus, it might be necessary to cluster together multiple microservices, which should be backed up in lockstep (see Figure 6). While this might not be necessary for all microservices into which the monolith has been decomposed, chances are that the granularity of the microservices to be consistently backed-up will be larger.

REFERENCES

1. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, 2015.
2. L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*, O'Reilly Media, 2013.
3. M. Amundsen, *Building Hypermedia APIs with HTML5 and Node*, O'Reilly Media, 2011.
4. T. Killalea, “The hidden dividends of microservices,” *Communications of the ACM*, vol. 59, no. 8, 2016, pp. 42–45; https://dl.acm.org/citation.cfm?doid=2948985.
5. L. Bass, I. Weber, and L. Zhu, *DevOps:A Software Architect's Perspective*, Addison-Wesley, 2015.
6. E. Evans, *Domain-driven design: tackling complexity in the heart of software*, Addison-Wesley, 2004.
7. W. Vogels, “Eventually consistent,” *Communications of the ACM*, vol. 52, no. 1, 2009, pp. 40–44; https://dl.acm.org/citation.cfm?id=1435432.
8. D. Bermbach and S. Tai, “Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior,” *Proc. of the 6th Workshop on Middleware for Service Oriented Computing* (MW4SOC), 2011, p. 1:1; https://dl.acm.org/citation.cfm?doid=2093185.2093186.
9. E. Brewer, “CAP twelve years later: How the "rules" have changed,” *Computer*, vol. 45, no. 2, 2012, pp. 23–29; http://ieeexplore.ieee.org/document/6133253/.
10. P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Communications of the ACM*, vol. 56, no. 5, 2013, pp. 55–63; https://dl.acm.org/citation.cfm?doid=2447976.2447992.

11. G. Pardon and C. Pautasso, “Atomic distributed transactions: A RESTful design,” *Proc. of the 5th International Workshop on Web APIs and RESTful Design* (WS-REST 2014), 2014.
12. C. Pautasso et al., “Microservices in practice, part 1: Reality check and service design,” *IEEE Software*, vol. 34, no. 1, 2017, pp. 91–98; http://ieeexplore.ieee.org/document/7819415/.
13. M. Fowler: “Event Sourcing”, https://martinfowler.com/eaaDev/EventSourcing.html (2005).
14. B. Kemme and G. Alonso, “Database Replication: A tale of research across communities,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010, pp. 5–12.
15. F.B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, 1990, pp. 299–319; https://dl.acm.org/citation.cfm?id=98167.
16. K.M. Chandy and L. Lamport, “Distributed Snapshots: Determining global states of distributed systems,” *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, 1985, pp. 63–75; https://dl.acm.org/citation.cfm?id=214456.
17. R.S. Barga, D.B. Lomet, and G. Weikum, “Recovery guarantees for general multi-tier applications,” *Proceedings of the 18th International Conference on Data Engineering* (ICDE2002), 2002, pp. 543–554.
18. C. Pautasso and E. Wilde, “Why is the web loosely coupled? A multi-faceted metric for service design,” *Proc. of the 18th World Wide Web Conference* ((WWW2009)), 2009, pp. 911–920.
19. M. Gysel et al., “Service Cutter: A systematic approach to service decomposition,” *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference* (ESOCC), 2016, pp. 185–200.

ABOUT THE AUTHORS

Guy Pardon is the chief architect at Atomikos (www.atomikos.com), where he leads the development of both traditional and modern (service-oriented) transaction technology. He occasionally speaks at conferences, publishes technical articles, and teaches trainings related to mission-critical enterprise application development—his main professional interest. His main research interest concerns reliability for distributed systems. He has a PhD from ETH Zurich, Switzerland. Contact him at guy@atomikos.com or follow him @guypardon.

Cesare Pautasso is a professor at the Faculty of Informatics, Università della Svizzera Italiana (USI) Lugano, Switzerland, where he leads the Architecture, Design, and Web Information Systems Engineering research group. He supervises the research of five PhD students who are building experimental systems to explore the intersection of cloud computing, software architecture, Web engineering, and business-process management. Previously, he was a researcher at the IBM Zurich Research Lab and a senior researcher at ETH Zurich, from which he has a PhD. He is the coauthor of the books “SOA with REST” and “Just Send an Email: Anti-patterns for email-centric organizations,” and he is the co-editor of the *IEEE Software* Insights department. Find more information at www.pautasso.info. Follow him @pautasso. Contact him at c.pautasso@ieee.org.

Olaf Zimmermann is a professor of software architecture and an institute partner at the Institute for Software at the Hochschule für Technik Rapperswil (HSR). His research areas include service-oriented computing and architectural knowledge management. Previously, he was a senior principal scientist at ABB Corporate Research and a researcher and executive IT architect at IBM. He has a PhD from Stuttgart University. The Open Group awarded him a Distinguished IT Architect (Chief/Lead) Certification. He is a book author and a co-editor of the Insights column in *IEEE Software*. Contact him at olaf.zimmermann@hsr.ch.