Microservice Disaster Crash Recovery: A Weak Global Referential Integrity Management

Maude Manouvrier[1(✉)], Cesare Pautasso[2], and Marta Rukoz[1,3]

[1] Université Paris-Dauphine, PSL Research University CNRS UMR [7243] LAMSADE, Paris, France
maude.manouvrier@dauphine.fr
[2] Software Institute, Faculty of Informatics, USI, Lugano, Switzerland
[3] Université Paris-Nanterre, Nanterre, France

Abstract. Microservices which use polyglot persistence (using multiple data storage techniques) cannot be recovered in a consistent state from backups taken independently. As a consequence, references across microservice boundaries may break after disaster recovery. In this paper, we give a weak global consistency definition for microservice architectures and present a recovery protocol which takes advantage of cached referenced data to reduce the amnesia interval for the recovered microservice, i.e., the time interval after the most recent backup, during which state changes may have been lost.

Keywords: Microservices · Referential integrity · Backup · Weak global consistency

1 Introduction

Microservices are small autonomous services, deployed independently, that implement a single, generally limited, business functionality [6,14,21,23]. Microservices may need to store data. Different data storage pattern exist for microservices [21]. In the Database per Service pattern, defined in [19]: each microservice stores its persistent data in a private database. Each microservice has full control of a private database, persistent data being accessible to other services only via an API [24]. The invocation of a service API will result in transactions which only involve its database.

Relationships between related entities of an application based on a microservice architecture are represented by links: the state of a microservice can include links to other entities found on other microservice APIs [18]. Following the hypermedia design principle of the REST architectural style, these links can be expressed with Uniform Resource Identifiers (URIs) which globally address the referenced entities.

Since microservices are autonomous, not only do they use the most appropriate database technology for persistent storage of their state, but they also

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12138, pp. 482–495, 2020.
https://doi.org/10.1007/978-3-030-50417-5_36

operate following an independent lifecycle, when their database is periodically backed up. For an entire microservice architecture, in practice, it is not very feasible to take an atomic snapshot of the state of all microservices. Thus, in case of one microservice crashes, which then needs to be recovered from its backup, the overall state of the microservice architecture may become inconsistent after recovery [18]. After recovery, such inconsistency may manifest itself as broken links between different microservices.

This paper presents a solution to ensure that the links between different entities managed by different microservices remain valid and intact even in the case of a database crash. The solution assumes that microservices referring to entities managed by other microservices will not only store the corresponding link, but also conserve a cached representation of the most recent known values. We present a recovery protocol when the crashed microservice can merge its own possibly stale backup with the possibly more recent cached representations obtained from other microservices. Thus, we revisit the definition of weak referential integrity across distributed microservice architectures.

2 Background and Related Work

2.1 Database Consistency, Durability, Backup and Disaster Crash Recovery

A database has a state, which is a value for each of its elements. The state of a database is consistent if it satisfies all the constraints [22]. Among constraints that ensure database consistency, referential integrity [8] is a database constraint that ensures that references between data are indeed valid and intact [4]. In a relational database, the referential integrity constraint states that a tuple/row in one relation referring, using a foreign key, to another relation, must refer to an existing tuple/row in that relation [11]. When a reference is defined, i.e. a value is assigned to a foreign key, the validity of the reference is checked, i.e. the referenced tuple should exist. In case of deletion, depending on the foreign key definition, the deletion of a tuple is forbidden if there are dependent foreign-key records, or the deletion of a record may cause the deletion of corresponding foreign-key records, or the corresponding foreign keys are set to null. Referential integrity is really broader and encompasses databases in general and not only relational ones [4].

Durability means that once a transaction, i.e. a set of update operations on the data, is committed, it cannot be abrogated. In the centralized databases systems, checkpoint and log are normally used to recover the state of the database in case of a system failure (e.g. the contents of main memory disappear due to a power loss and the content of a broken disk becoming illegible) [22]. Checkpoint is the point of synchronization between the database and transaction log file when all buffers are force-written to secondary storage [7]. For this kind of failure, the database can be reconstructed only if:

- the log has been stored on another disk, separately from the failure one(s),
- the log has been kept after a checkpoint, and

– the log provides information to redo changes performed by transactions before the failure, and after the latest checkpoint.

To protect the database against media failures an up-to-date backup of the database, i.e. a copy of the database separate from the database itself, is used [22]. A backup of the database and its log can be periodically copied onto offline storage media [7]. In case of database corruption or device failure, the database can be restored from one of the backup copies, typically the most recent one [3]. In this case, the recovery is carried out using the backup and the log – see [15], for more details.

A database has a Disaster Crash when the main memory and the log, or a part of the log, are lost. Therefore, to recover the database, an old, maybe obsolete, backup of the database is used. Data which was not part of the backup will be lost. In case of a disaster crash, the system cannot guarantee the durability property. However, in a centralized database, recovery from a backup provides a database which has a consistent state.

2.2 Microservices as a Federated Multidatabase

Each database of a microservice can be seen as a centralized database. Seen across an entire microservice architecture, the microservice databases represent a distributed database system. A multidatabase is a distributed database system in which each site maintains complete autonomy. Federated multidatabase is a hybrid between distributed and centralized databases. It is a distributed system for global users and a centralized one for local users [7]. According to the definitions above, stateful microservice architectures can therefore be seen as a federated multidatabase.

A microservice database can store either a snapshot of the current state of the data, containing the most recent value of data, or an event log, i.e. the current state of the data can be rebuilt by replaying the log entries, which record the changes to the microservice state in the database transaction log. Let's consider an example of a microservice managing orders. Using the snapshot architecture, the current state of an order can be stored in a row of a relational table *Order*. When using the event sourcing (log) [17], the application persists each order as a sequence of events e.g., listing the creation of the order, its update with customer details and the addition of each line item.

Each microservice ensures the durability and the consistency of its database, like in centralized databases. In the microservice context, each microservice manages its own database and stores independent backup of its own database, in order to permit disaster recovery from backup. However, while managing consistent backup is simple in a centralized database, maintaining consistent backups with distributed persistence in a federated multidatabase is challenging, as shown in the survey of [13]. So a model providing global consistent backup is necessary for microservices.

Microservice architecture deals with breaking foreign key relationships [16]. Each microservice can refer to other microservices data through loosely coupled references (i.e., URLs or links), which can be dereferenced using the API

provided by the microservice managing the referenced data. Microservices are independent and the managing reference integrity between them is challenging. As for the World Wide Web [9,12], there is no guarantee that a link retrieved from a microservice points to a valid URL [18]. In the following section, we propose a model providing global reference integrity for microservices.

2.3 Microservice Disaster Recovery

In [18], the authors have addressed the problem of backing up an entire microservice architecture and recovering it in case of a disaster crash affecting one microservice. They defined the BAC theorem, inspired from the CAP Theorem [5], which states that when backing up an entire microservice architecture, it is not possible to have both availability and consistency.

Let us consider the microservice architecture defined in [18], where each microservice manages its own database and can refer to other microservices data through loosely coupled references. Each microservice does independent backup of its own database for the purpose of allowing disaster recovery from backup.

Figure 1 presents an example of two microservices with their independent backup, data of the microservice *Order* referring data of microservice *Customer*. Database of each microservice is represented in gray and data in black. Each database contains three entities. Entities C/i ($i \in \{1,2,3\}$) correspond to customers, described by a name, and are managed by microservice *Customer*. Entities O/i ($i \in \{1,2,3\}$) correspond to orders and are managed by microservice *Order*. Each order O/i refers to a customer C/i. Backups of the database are represented in blue. The backup of microservice *Customer* only contains a copy of customers $C/1$ and $C/2$. The backup and the database of microservice *Order* are, on the contrary, synchronized.

As explained in [18], in case of disaster crash, independent backup may lead to broken link (see Fig. 2): no more customer $C/3$ exists after *Customer* recovery, then $O/3$ has a broken link.

A solution to avoid broken link is to synchronize the backup of all microservices, leading to limited autonomy of microservices and loss of data. In Fig. 3, both order and customer $C/3$ and $O/3$ are lost after the recovery.

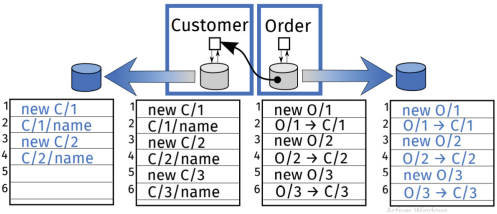

Fig. 1. An example of microservice architecture with independent backup (Color figure online)

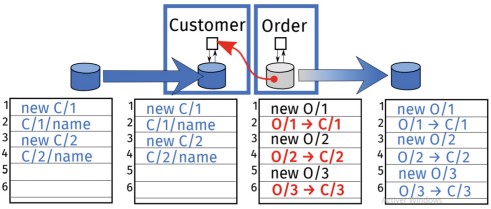

Fig. 2. The link from the *Order* microservice to entity $C/3$ is broken after the recovery of *Customer* microservice from an old backup

Please note that broken link can also appear when a referenced data is deleted, e.g. when a customer is deleted in the local database of microservice *Customer*. In this case, the referential integrity is not respected.

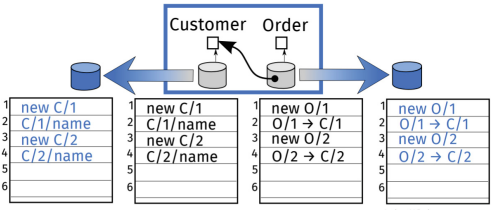

Fig. 3. Synchronized backup of an entire microservice architecture

As aforementioned, several approaches indicate that microservice architecture implies some challenging problems of data integrity and consistency management [2,18], as well as the difficulty of managing consistent backups due to distributed persistence [13]. However, as far as we know, no approach proposes a solution to such problems. In the following, we present a solution that can bypass such referential integrity violation and broken links.

3 Our Solution: A Weak Referential Integrity Management

In this work, we focus on referential integrity. We present a solution to help the user in the recovery of the system referential integrity in case of a disaster crash. We define the global consistency as a time-dependent property. We propose a new global consistency definition, called *the weak global referential consistency*. Our solution provides information about the global state in case of a disaster crash that the users can pinpoint exactly the location, and time interval, of missing data which needs to be manually repaired.

In the following, we first present the context and assumptions (Subsect. 3.1), without taking disaster crash into consideration. Then, we introduce our definition of global consistency (Subsect. 3.2). Based on this definition, we show the

Table 1. Table of symbols

Symbol	Description
μs	Microservice
D	Database of microservice μs
e	Entity of a database D
URI^e	Uniform Resource Identifier of an entity e
t_j	Date of the last update of an entity e in D
d^e	Dependency counter associated with entity e
E_k	Epoch identity, k being a timestamp
$t_{(i,k)}$	ith timestamp related to epoch E_k
B	Backup of the local database of microservice μs
$]t_{(i,k)}, t_{(j,k')}]$	Amnesia interval of an entity e

method of recovery from a disaster crash affecting one microservice. All symbols used in this article can be found in Table 1 above.

3.1 Context and Assumptions

In this article, microservice follows the pattern called Database per Service (defined in [19]), where each microservice has full control of a private database, persistent data being accessible to other services only via an API. Each microservice also use an event-driven architecture, such as the one defined in [20], consuming and publishing a sequence of state-changing events.

The following are our assumptions:

- Microservices are part of the same application.
- All microservices of an application trust each other.
- Each microservice μs has a database D storing a set of entities.
- Each entity $e \in D$ can be either a RESTful API resource, a relational tuple, a key-value record, a document or graph database item.
- Each entity e has a Uniform Resource Identifier, URI^e, that identifies the entity.
- The state of each entity e is read, updated and deleted using standard HTTP protocol primitives (GET, PUT and DELETE). In addition, we introduce two additional operations: `getReference`, `deleteReference`.
- Each microservice μs ensures the consistency and the durability of its own database D.

Taking into consideration the following ; handling the references between the different microservices and ensuring that the system is reliable when no failure occurs:

- Each microservice has its own clock. The clock of different microservices are not necessarily synchronized.
- An entity e', managed by a microservice $\mu s'$, can refer to another entity e managed by a microservice μs.
- The reference from microservice $\mu s'$ to an entity e, managed by a microservice μs, is the couple (URI^e, t_j) with the timestamp t_j marking the date of the last update of entity e in D as it is known by the microservice $\mu s'$, i.e. exactly when $\mu s'$ queries the microservice μs, using the clock of microservice μs.

There are 2 cases as far as reference storage is concerned:

1. the minimalist case consists in just storing the reference and the most recent modified timestamp, i.e. couple (URI^e, t_j);
2. the eager/self-contained backup case consists in storing a copy of the referenced entity state, that can be cached by $\mu s'$. When microservice $\mu s'$ stores a copy of the referenced entity in its cache, this former copy is considered as detached, identical to detached entity in object-relational mapping using JPA specification [10]. Detached means that the copy is not managed by μs, microservice $\mu s'$ being responsible for keeping its cache up-to-date. Cached representation is only a representation of the original entity state, thus it may only contain a projection. For our solution, we assume that it is possible to reconstruct the original entity state from its cached representation.

3.2 Global Consistency

In case of no disaster crash, the global consistency can be defined as follows:

Definition 1. The global consistency
A global state is consistent if:

- *(local database consistency) each local database is locally consistent in the traditional sense of a database, i.e. all its integrity constraints are satisfied.*

and

- ***(global referential integrity)** the timestamp value associated with each reference is less than or equal to the timestamp value of the corresponding referenced entity.*
  Formally: for each couple (URI^e, t_j) associated with an entity e' referencing another entity e of μs, $t_j \leq t_i$, with t_i the most recent update timestamp of e in μs.

Case of Snapshot Data Storage Pattern. When using the snapshot data storage pattern, each local database contains the current state of its microservice. In order to guarantee the referential integrity in the microservice architecture, a microservice μs, cannot delete an entity e, if there is an entity e' managed by another microservice $\mu s'$ that refers to the entity e. We suggest a referential integrity mechanism based on dependency counters, as follows:

- Each entity e managed by a microservice μs is associated with a dependency counter d^e. This counter indicates how many other entities managed by other microservices refer to entity e. It is initially set at 0.
- When a microservice $\mu s'$ wants to create the entity e' that refers an entity e managed by μs, it sends a `getReference` message to microservice μs. The corresponding dependency counter d^e is incremented. Then, μs sends the couple (URI^e, t_j) back to $\mu s'$, with t_j, the date of the most recent update of entity e.
- When microservice $\mu s'$ receives the information about e, it creates the entity e'.
- When microservice $\mu s'$ deletes an entity e' that refers e, it sends a message `deleteReference` to microservice μs, indicating that the reference to e does not exist any more. d^e is therefore decremented.
- Microservice μs cannot delete an entity e if its dependency counter is $d^e > 0$. It retains the most recent value of entity e with its most recent update time, t_j, and flags the entity by $\perp$ indicating that e must be deleted when its dependency counter reaches the value of 0.

According to Definition 1, a reliable microservice system using the referential integrity mechanism based on dependency counters, will always be globally consistent.

Case of Event Sourcing Data Storage Pattern. When choosing event sourcing as data storage pattern [20], each local database contains an event log, which records all changes of the microservice state. Thus, it is possible to rebuild the current state of the data by replaying the event log. In this case, we propose the following referential integrity mechanism:

- When a microservice $\mu s'$ wants to create the entity e' that refers an entity e managed by μs, it sends a `getReference` message to microservice μs. When $\mu s'$ receives the information about e, it creates the entity e' and a creation event, associated with the corresponding reference (URI^e, t_j), is stored in the log of $\mu s'$.
- When an entity e of microservice μs must be deleted, instead of deleting it, the microservice μs flags it by $\perp$, and a deletion event, associated with the related timestamp, is stored in its log, representing the most recent valid value of entity e.

Thus, it is easy to prove that global consistency state can be obtained from the logs. For each couple (URI^e, t_j) of a referenced entity e, timestamp t_j must appear in the event log of microservice μs. Moreover the most recent record associated with entity e, corresponding to an update or deletion of e, in the event log of μs, has a timestamp $t_i \geq t_j$, with t_j, any timestamp appearing in any reference couple (URI^e, t_j) stored in the event log of any other microservice referencing e.

3.3 Fault Tolerant Management of Microservice Referential Integrity

As explained in Sect. 2.3, disaster crash can occur in microservice architectures. In the following, we consider disaster crash affecting only one microservice.

To protect the local database from media-failure, each microservice stores an up-to-date backup of its database, i.e. a copy of the database separate from the database itself. Each microservice individually manages the backup of its database. The way in which microservices independently manage their backup is out of the scope of this paper.

A disaster crash of a microservice μs means that its local database and its log are lost and we have to recover the database from a past backup. The backup provides a consistent state of the local database. However, as the database has been recovered from a past backup, data could have been lost. In order to provide a state of the local database as close as possible to the one of the database before the failure, data cached by other microservices can be used. When a microservice $\mu s'$ refers to an entity managed by another microservice μs, it can store a detached copy of the referenced entity. Therefore, these detached copies can be used to update the state of the database obtained after recovery from the backup.

In the following, we present the concepts used to manage disaster crash, our recovery protocol, how to optimize it and we define the Weak Global Referential Integrity.

Backup and Recovery, Amnesia Interval and New Epoch. To manage disaster crash, our assumptions are:

- Each entity of the local database of microservice μs is associated with an epoch identity E_k. An epoch is a new period after a disaster crash recovery. A new epoch E_k begins at the first access of an entity after recovery. Therefore, a timestamp $t_{(i,k)}$ associated with an entity e represents the ith timestamp related to epoch E_k. $k = 0$ when no crash has occurred, $k > 0$ otherwise. The value of k always increases, being associated with time.
- When a backup B of the local database of microservice μs is done, operation BCK, the backup is associated with clock epoch identity $E_{k'}$, and with a creation timestamp $t_{(i,k')}$, such that: all entities e, stored in backup B, have an updated timestamp $t_{(j,k')} \leq t_{(i,k')}$. Epoch $E_{k'}$ associated with backup and epoch E_k associated with the local database are such that: $k' \leq k$.
- As long as there is no disaster crash, the local database and the backup are associated with the same epoch identity.
- In case of disaster crash of microservice μs, when the local database is locally recovered from an past obsolete backup created at time $t_{(c,k')}$, it is known that local database has an amnesia interval starting from $t_{(c,k')}$. This amnesia interval is associated with all entities saved in the backup and lasts until such entities are accessed again (see Definition 2).

– Each entity of the recovered database is associated with a timestamp related to epoch E'_k of the backup. This timestamp remains as long as no updates have been carried out. A new epoch E_k begins at the first reading or written access of an entity, k containing the current date. A written operation, PUT, overwrites whatever value was recovered. However, epoch should also be updated after a reading operation, GET. Any other microservice reading from the state of the recovered entity will establish a causal dependency, which would be in conflict with further more recent recovered values from the previous epoch (see [1] for more details).

Definition 2. Amnesia Interval
An amnesia interval of microservice μs is a time interval indicating that a disaster crash has occurred for the local database of μs. This interval is associated with each entity managed by μs. An amnesia interval $]t_{(i,k')}, t_{(j,k)}]$ of an entity e means that:

– *Epoch $E_{k'}$ is the epoch associated with the backup used for the database recovery.*
– *Timestamp $t_{(i,k')}$ corresponds to the time of most recent known update of e. It is either the timestamp associated with the backup used for recovery, or the timestamp of a cached copy of e stored in a microservice referring entity e.*
– *Timestamp $t_{(j,k)}$ corresponds to the first reading or written operation on e from another microservice $\mu s'$, after $t_{(i,k')}$ $(k' < k)$.*

Weak Global Referential Integrity. After a crash recovery, data can be lost, so we define a weak global referential integrity of the microservice architecture. Weak means that either the global referential integrity has been checked, verifying Definition 1, or an amnesia is discovered; data has been lost as well as the interval of time when the data was lost. This makes it possible to focus on the manual data recovery and reconstruction effort within the amnesia interval.

Definition 3. Weak global consistency
After a disaster crash recovery of a microservice μs, the system checks a weak global consistency iff:

– *(local database consistency) each local database is locally consistent in the traditional sense of a database, i.e. all its integrity constraints are satisfied.*

and

– ***(weak global referential integrity)** the timestamp value associated with each reference is either less than or equal to the timestamp value of the corresponding referenced entity or included in an amnesia interval.*
Formally: for each couple $(URI^e, t_{(\ell,\kappa)})$ associated with an entity of $\mu s'$ referencing another entity e of μs:
  - *either $t_{(\ell,\kappa)} \leq t_{(i,k')}$, with $t_{(i,k')}$ the most recent update timestamp of e in μs, epochs κ and k' being comparable $(\kappa \leq k')$;*

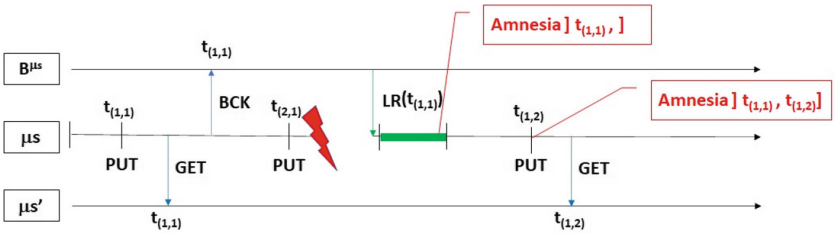

Fig. 4. Example of a scenario with 2 microservices, without cached data. *PUT* represents a state change of the referenced entity. *BCK* indicates when a backup snapshot is taken. *LR* shows when the microservice is locally recovered from the backup. (Color figure online)

- *or* $t_{(\ell,\kappa)} \in]t_{(i,k')}, t_{(j,k)}]$, *with* $]t_{(i,k')}, t_{(j,k)}]$ *the amnesia interval associated with the referenced entity* e*, after a disaster crash of* μs *that manages* e*.*

Consider a scenario of two microservices μs and $\mu s'$, $\mu s'$ referencing an entity managed by μs (see Fig. 4), but without storing any cache of the referenced entity. In figures, only timestamps of one entity of microservice μs are considered, timestamps and epoch identities being only represented by numbers. In Fig. 4, at time $t_{(1,1)}$ an entity is created by microservice μs ; operation `PUT`. A backup B is made, storing entities of microservice μs created before $t_{(2,1)}$; operation `BCK`. Microservice $\mu s'$ refers the entity of μs created at time $t_{(1,1)}$; operation `GET`. An update of the entity is carried out by the microservice μs at time $t_{(2,1)}$; operation `PUT`. When disaster crash appears to μs (see red flash), μs must recover using the backup, update of time $t_{(2,1)}$ is lost, therefore it has amnesia that begins from $t_{(1,1)}$. An update is done to the entity, then a new epoch 2 begins and timestamp $t_{(1,2)}$ is associated to the updated value. The amnesia interval is then updated to $]t_{(1,1)}, t_{(1,2)}]$. If $\mu s'$ does another `GET` to refresh the referenced value, the up-to-date timestamp $t_{(1,2)}$ is sent by μs.

Recovery Protocol. When a disaster crash occurs to a microservice μs, μs informs all other services of its recovery. Moreover, when microservices stored copies of the entities they refer to, in their cache, the amnesia interval associated with each recovered entity of μs can be reduced using cached replicas. In order to do so, the steps are:

- After the recovery of μs, an event indicating that there is amnesia is sent, or broadcast, to other microservices.
- When a microservice $\mu s'$ receives an amnesia event from microservice μs, managing an entity e it refers to; if it has stored a replica of e in its cache, then $\mu s'$ sends the replica of the entity it refers to, to μs.
- When microservice μs receives replies carrying information from $\mu s'$ about its entity e, it compares the value of e, associated with timestamp $t_{(i,k')}$, with the value, associated with timestamp $t_{(j,k)}$, it stored, if epochs k and k'

are comparable. Then, it retains the more up-to-date value and shrinks the amnesia interval associated with e if necessary.
- Once a read operation or an update operation is done on e, a new epoch begins, and the first timestamp associated with this new epoch represents the end of the amnesia interval.
- The beginning of the amnesia interval can still be shifted if more up-to-date values are received from belated replies from other cached replicas.

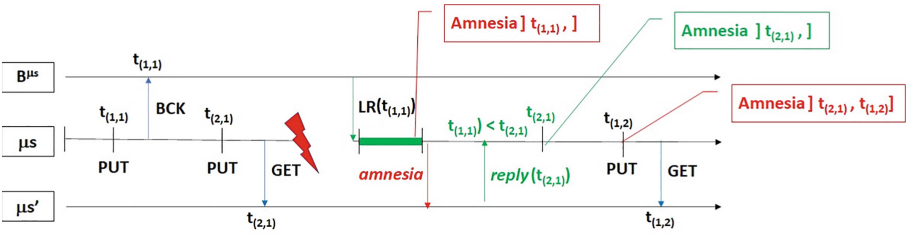

Fig. 5. Recovery scenario using cached data more recent than the backup.

In Fig. 5, $\mu s'$ stores an up-to-date value of the referenced entity, after the backup of μs ; operation `GET`. After the recovery from the past backup of time $t_{(1,1)}$, μs sends an event about its amnesia, associated with interval $]t_{(1,1)},]$. After receiving this amnesia event, $\mu s'$ sends its up-to-date value, associated with $t_{(2,1)} > t_{(1,1)}$, to μs ; event `reply`. μs stores this up-to-date value, associated with timestamp $t_{(2,1)}$, and updates the amnesia interval to $]t_{(2,1)},]$. After a update is done to the entity, a new epoch 2 begins and timestamp $t_{(1,2)}$ is associated to the updated value ; operation `PUT`. The amnesia interval is then updated to $]t_{(2,1)}, t_{(1,2)}]$.

Availability vs Consistency. After a disaster crash of μs: either μs is immediately available after its local recovery, or it expects information sent by other microservices that refer its entities before the disaster crash, to provide a more recent database snapshot than the past used backup, updating the value stored in the backup with the copy stored in the cache of the other microservices.

If we are uncertain that all microservices will answer the amnesia event or if μs ignores or partially knows which microservices refer to (case 1), μs can wait for a defined timeout.

If we assume that all microservices are available and will answer to the amnesia event (case 2), μs waits until all microservices have sent their reply to the amnesia event.

After the timeout (case 1) or the reception of all responses (case 2), the recovery is ended and μs is available.

When dependency counters are used (see Sect. 3.2) and if we are sure that the identity of all microservices that refer to μs is known after the disaster crash: an

optimization of the recovery process can be used. In this case, an amnesia event is sent only to all microservices referring to μs, instead of broadcast, and μs waits until all the aforementioned microservices reply. To do so, the address of each microservice referring to μs should be stored by μs, when the dependency counter is updated. Each referencing microservice can either send the value it stores in its cache, or a message indicating that it is no longer concerned by the amnesia, because it currently does not refer to any entity of μs.

The choice between the aforementioned steps depends on the focus on availability (μs is available as soon as possible after its disaster crash, with a large amnesia interval) or on consistency (we prefer to wait in order to provide a more recent snapshot than the one used for the recovery before making μs available).

4 Conclusions and Future Work

In this paper, we have focused on preserving referential integrity within microservice architecture during disaster recovery. We have introduced a definition of weak global referential consistency and a recovery protocol taking advantage of replicas found in microservice caches. These are merged with local backup to reduce the amnesia interval of the recovered microservice. The approach has been validated under several assumptions: direct references to simple entities, single crashes and no concurrent recovery of more than one failed microservice.

In this paper we focused on reliability aspects, whereas as part of future work we plan to assess the performance implications of our approach in depth. We will also address more complex relationships between microservices, e.g., transitive or circular dependencies, which may span across multiple microservices. While microservice architecture is known for its ability to isolate failures, which should not cascade across multiple microservices, it remains an open question how to apply our approach to perform the concurrent recovery of multiple microservices which may have failed independently over an overlapping period of time.

Acknowledgements. The authors would like to thank Guy Pardon, Eirlys Da Costa Seixas and the referees of the article for their insightful feedback.

References

1. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (1995). https://doi.org/10.1007/BF01784241
2. Baresi, L., Garriga, M.: Microservices: the evolution and extinction of web services? Microservices, pp. 3–28. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-31646-4_1
3. Bhattacharya, S., Mohan, C., Brannon, K.W., Narang, I., Hsiao, H.I., Subramanian, M.: Coordinating backup/recovery and data consistency between database and file systems. In: ACM SIGMOD International Conference on Management of data, pp. 500–511. ACM (2002)

4. Blaha, M.: Referential integrity is important for databases. Modelsoft Consulting Corp. (2005)
5. Brewer, E.: CAP twelve years later: how the "rules" have changed. Computer **45**(2), 23–29 (2012)
6. Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M.: From monolithic to microservices: an experience report from the banking domain. IEEE Softw. **35**(3), 50–55 (2018)
7. Connoly, T., Begg, C.: Database Systems. ke-3. Addison-Wesley, England (1998)
8. Date, C.J.: Referential integrity. In: 7th International Conference on Very Large Data Bases (VLDB), pp. 2–12 (1981)
9. Davis, H.C.: Referential integrity of links in open hypermedia systems. In: 9th ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space, pp. 207–216 (1998)
10. DeMichiel, L., Keith, M.: Java persistence API. JSR 220 (2006)
11. Elmasri, R., Navathe, S.: Fundamentals of Database Systems. Addison-Wesley, Boston (2010)
12. Ingham, D., Caughey, S., Little, M.: Fixing the "broken-link" problem: the W3objects approach. Comput. Netw. ISDN Syst. **28**(7–11), 1255–1268 (1996)
13. Knoche, H., Hasselbring, W.: Drivers and barriers for microservice adoption-a survey among professionals in Germany. Enterp. Model. Inf. Syst. Architect. (EMISAJ) **14**, 1–1 (2019)
14. Lewis, J., Fowler, M.: Microservices a definition of this new architectural term (2014). http://martinfowler.com/articles/microservices.html
15. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. (TODS) **17**(1), 94–162 (1992)
16. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly, Newton (2015)
17. Overeem, M., Spoor, M., Jansen, S.: The dark side of event sourcing: managing data conversion. In: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 193–204. IEEE (2017)
18. Pardon, G., Pautasso, C., Zimmermann, O.: Consistent disaster recovery for microservices: the BAC theorem. IEEE Cloud Comput. **5**(1), 49–59 (2018)
19. Richardson, C.: Pattern: database per service (2018). https://microservices.io/patterns/data/database-per-service.html. Accessed 02 Apr 2020
20. Richardson, C.: Pattern: event sourcing (2018). https://microservices.io/patterns/data/event-sourcing.html. Accessed 01 Apr 2019
21. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: a systematic mapping study. In: CLOSER, pp. 221–232 (2018)
22. Ullman, J.D., Garcia-Molina, H., Widom, J.: Database Systems: The Complete Book, 1st edn. Prentice Hall, Upper Saddle River (2001)
23. Zimmermann, O.: Microservices tenets. Comput. Sci. Res. Dev. **32**, 301–310 (2016). https://doi.org/10.1007/s00450-016-0337-0
24. Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U.: Introduction to microservice API patterns (MAP). In: Joint Post-Proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019). OpenAccess Series in Informatics (OASIcs), vol. 78, pp. 4:1–4:17 (2020)