Service Oriented Computing and Applications (2021) 15:75–92
https://doi.org/10.1007/s11761-020-00311-z

ORIGINAL RESEARCH PAPER

ReLock: a resilient two-phase locking RESTful transaction model

Luca Frosini[1] · Pasquale Pagano[1] · Leonardo Candela[1] · Manuele Simi[1,2] · Cinzia Bernardeschi[3]

Received: 14 May 2020 / Revised: 22 September 2020 / Accepted: 24 November 2020 / Published online: 13 January 2021

Abstract
Service composition and supporting *transactions* across composed services are among the major challenges characterizing service-oriented computing. REpresentational State Transfer (REST) is one of the approaches used for implementing Web services that is gaining momentum thanks to its features making it suitable for cloud computing and microservices-based contexts. This paper introduces ReLock, a resilient RESTful transaction model introducing general purpose transactions on RESTful services by a layered approach and a two-phase locking mechanism not requesting any change to the RESTful services involved in a transaction.

Keywords RESTful web services · RESTful transaction model · ACID · Two-phase locking · Resiliency

1 Introduction

Service-oriented computing is a paradigm promoting the development of applications and business processes by combining loosely coupled *services* (application components) spanning organizations and computing platforms [5,25]. Service composition and supporting *transactions* across the composed services are among the challenges characterizing the paradigm [5,9,17,24]. In fact, transaction processing are mechanisms aiming at guaranteeing that a set of interdependent operations are either all completed successfully or all cancelled successfully thus to keep the overall state of the "system" consistent [4]. When the interdependent operations are performed by services on the Web, implementing transaction processing mechanisms become even more challenging [16,17].

Web services are implemented by using a variety of approaches, methodologies, and technologies. Among these approaches, there is the REpresentational State Transfer (REST) architectural style [11]. It is not an Internet standard. Nevertheless, it has gained popularity and emerged as best practice especially in the context of cloud computing and microservices mainly because it favors high scalability while keeping the complexity of Web services design, implementation, and deployment at very affordable costs [28].

REST is characterized by the following architectural constraints: (*i*) client–server architecture, (*ii*) statelessness, (*iii*) cacheability, (*iv*) layered system, (*v*) uniform interface. Each of these constraints ensures that the service gains desirable non-functional properties, such as scalability, simplicity, portability, performance, and reliability. Web services compliant with all of them are usually referred as *RESTful services*. The REST architectural style promotes a specific idea of organizing a modern client–server application governing state transition, whose states are potentially extensible to infinite.

A common misuse of the REST architectural style results from the misinterpretation of the "representational" adjective and the idea that it relates to the resource state. This often leads to build applications (*i*) erroneously focusing on object structures to communicate/transfer to the calling clients and

✉ Leonardo Candela
leonardo.candela@isti.cnr.it

Luca Frosini
luca.frosini@isti.cnr.it

Pasquale Pagano
pasquale.pagano@isti.cnr.it

Manuele Simi
manuele.simi@isti.cnr.it; mas2182@med.cornell.edu

Cinzia Bernardeschi
cinzia.bernardeschi@unipi.it

1 Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", Consiglio Nazionale delle Ricerche, Pisa, Italy

2 HRH Prince Alwaleed Bin Talal Bin Abdulaziz Alsaud Institute for Computational Biomedicine, Weill Cornell Medical College, New York, NY, USA

3 Dipartimento di Ingegneria dell'Informazione Università di Pisa, Pisa, Italy

(*ii*) encapsulating information about resource state into the object structures to be transferred. Rather, it is the transfer to be "representational", and so it is important to design applications that make use of the "representational" ability to support transfers among client–server components. Apart from violating one or more REST constraints, this misuse hides the problem of transactional operations. Such violations are commonly accepted to simulate what other approaches offers (e.g., see the WS-Transaction standard [6]). However, this results in a dangerous binding of the transaction information to the specific call.

In this manuscript, we introduce *ReLock*, a resilient two-phase locking RESTful transaction model. The objective of the proposed approach is to manage general purpose transactions that are completely unrelated to the actual data exchanges of the specific REST call and capable to work across calls and services. The proposed approach outperforms state of the art approaches discussed by Mihindukulasooriya et al. [20,21] by (*i*) being capable to satisfy all the scenarios discussed in preliminary works; (*ii*) overcoming all the challenges identified so far; and (*iii*) being RESTful-friendly, i.e., it does not violate any of the REST constraints.

The remainder of this paper is organized as follows: Sect. 2 gives an overview of the background principles needed to contextualize the ReLock approach; Sect. 3 presents the ReLock solution; Sect. 4 discusses the state of the art of RESTful transactions; Sect. 5 provides an analysis of the presented solution and some comparisons with other works. Finally, Sect. 6 concludes the paper and describes future work.

2 Background

To fully appreciate the ReLock approach, it is needed to establish a common understanding on three aspects characterizing the settings of transactions management across RESTful services, namely: the REST architectural style (cf. Sect. 2.1); the resource oriented architecture (ROA) proposing a concrete architecture for REST (cf. Sect. 2.2); and the properties characterizing transactions (cf. Sect. 2.3).

2.1 REpresentational state transfer (REST)

By having the design rationale of the web architecture as a driving principle, six principles have been defined to characterize the REST architectural style [11]:

- *Client–server paradigm* it refers to the separation of concern between client and server. Both the server and the client can evolve independently, provided that the exposed interface be left unaltered;
- *Statelessness* it indicates that the server does not store any information regarding previous interaction with the client. Instead, the client sends to the server all the information required to understand and elaborate the request correctly. The intent of this principle is to improve "visibility, reliability and scalability while decreasing network performances" [11];
- *Cacheability* it indicates to clients (and intermediates such as a proxy) that they have to cache the responses based on server indication (explicit or implicit). Cacheability allows clients to reuse data and reduce the needs of some interactions with servers. Cacheability helps to balance and/or mitigate the potential inefficiency in network performance introduced with the stateless principle;
- *Uniform interface* it allows decoupling the architecture from the implementation. This principle introduces four specific constraints:
  - *Identification of resources* it clarifies the key concept of resource by stating that "A resource is any information that can be named. It is any concept that might be the target of an author's hypertext reference" [11]. Every resource must be individually identifiable and the identifier does not change if the resource representation changes;
  - *Manipulation of resources through representations* it implies that the resource representation captures the state of the resource transferred between components;
  - *Self-descriptive messages* it means that every message exchanged between client and server must contain all information regarding how to elaborate the message itself "in order to support intermediate processing of interactions [11]" (e.g., the media type of the resource representation, cache control information);
  - *Hypermedia as the engine of application state (HATEOAS)* it suggests that the use of hypermedia drives clients interaction with servers.
- *Layered system* it envisages the possibility to add an arbitrary number of intermediary components between the client and the server. This property allows to decouple the service logic from higher level facilities. Layers can be used for reasons including: to improve the system scalability; to provide encapsulation facility for legacy and non-rest systems (by exposing them through the uniform interface); to achieve transparent cacheability across different clients; to add higher-level facilities such as security. As a counterpart, layering adds overhead and network latency;
- *Code on demand* this is an optional principle depending on the application and context. It indicates that the server can extend its functionality by providing clients code to execute. It allows distributing the computational load. It

is only an optional constraint because in some cases some intermediary can/must limit the transfer of code (e.g., for security reason).

Overall, these principles highlight that (*i*) the interaction between client and server is based on the representation of a "resource" they manage to exchange, and (*ii*) whenever the client receives the resource representation, it is posed in a specific *state*.

2.2 Resource-oriented architecture (ROA)

The ROA is a concrete architecture for REST relying on technologies such as uniform resource identifier (URI), HyperText transfer protocol (HTTP) and eXtensible Markup Language (XML) [30]. It uses standard HTTP methods applied to URI to realize Create, Read, Update, Delete (CRUD) operations on resources [19].

ROA gives particular emphasis on "make it a resource" paradigm and proposes descriptive and predictable URI as technology to satisfy the resource identification constraint. Hence, any resource in ROA has a URI. Moreover, to satisfy the uniform interface constraint, ROA indicates the way to construct URI for resources and how to use HTTP methods to them. An example extracted from [30] on how to create a bookmark service adhering to the ROA architecture is given in Table 1.

For the implementation of CRUD operations, ROA suggests to use `POST`, `GET`, `PUT` and `DELETE`. Table 2 shows how CRUD operations are performed by HTTP methods. Moreover, it shows the per operation expectation in terms of *safety* and *idempotency* property. In particular—according to HTTP specification [12,23]:

- the `POST` method is used to create a new resource without providing the URI of the resource to create. The representation of the resource is sent, as part of the HTTP body, to the collection that will contain the resource. The server determines its appropriate location, and provides the client with the resulting URI.
- the `GET` method is used to obtain the representation of a resource.
- the `PUT` method is used to update an existing resource. This operation instructs the server to apply a new representation as a replacement of the previous one. Moreover, the PUT method can be used to create a new resource if the client is willing to provide the URI of the resource to create.
- the `DELETE` method is used to remove an existing resource.
- the `GET`, `PUT` and `DELETE` methods must be *idempotent*, i.e., the same operation executed multiple times has the same effect than executing it one time only. Moreover, "repeating the request will have the same intended effect, even if the original request succeeded, though the response might differ" [12].
- the `GET` method must be *safe*, i.e., it must have no side effect. "This does not prevent an implementation from including behavior that is potentially harmful, that is not entirely read only, or that causes side effects while invoking a safe method" [12].

Table 2 Mapping between CRUD operations and HTTP methods enriched by safety and idempotency property they must satisfy

Operation	HTTP method	Safe	Idempotent
Create	`POST`	No	No
Read	`GET`	Yes	Yes
Update	`PUT`	No	Yes[1]
Delete	`DELETE`	No	Yes[2]

[1]PUT can be also used to create a resource when used with the URI where the resource will be available

[2]Allamaraju [1] argues that `DELETE`idempotency should be accomplished client-side. The server should inform the client if a delete succeeded because the resource was really deleted or it was not found i.e., 404 Not Found error is suggested instead of 204 No Content. The latter situation should be treated as idempotent by the client

Table 1 ROA compliant URI compared to non REST service

Operation	HTTP method	ROA URI[1]	Non-ROA API[2]
Listing	`GET`	`/users/{USERNAME}/bookmarks`	`GET/posts/list`
Create	`POST`	`/users/{USERNAME}/bookmarks`	`GET/posts/add`
Create	`PUT`	`/users/{USERNAME}/bookmarks/{URI-MD5}`	`GET/posts/add`
Read	`GET`	`/users/{USERNAME}/bookmarks/{URI-MD5}`	`GET/posts/get`
Update	`PUT`	`/users/{USERNAME}/bookmarks/{URI-MD5}`	`GET/posts/update`
Delete	`DELETE`	`/users/{USERNAME}/bookmarks/{URI-MD5}`	`GET/posts/delete`

[1]The path variable parameters are shown within { } and using capital letters. This convention will be used in the rest of the paper

[2]The information regarding which username and URL save as bookmark or which URL retrieve are provided via `GET`URL parameters not shown in the table

Apart from the above methods, HTTP 1.1 defines also the HTTP methods HEAD, CONNECT, OPTIONS and TRACE [12]:

- the HEAD method is identical to GET except that the server must not send a message body in the response. This method can be used for obtaining metadata about the selected representation without transferring the representation data and is often used for testing hypertext links for validity, accessibility, and recent modification;
- the CONNECT method requests that the recipient establishes a tunnel to the destination origin server identified by the request-target and, if successful, thereafter restricts its behavior to blind forwarding of packets, in both directions, until the tunnel is closed. CONNECT is intended only for use in requests to a proxy;
- the OPTIONS method requests information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options or the requirements associated with a resource, or the capabilities of a server, without implying a resource action. A OPTIONS request with an asterisk ("*") as the request-target applies to the server in general rather than to a specific resource;
- the TRACE method requests a remote, application-level loop-back of the request message.

From the ROA perspective [30]: (*a*) HEAD and OPTIONS methods are also part of the uniform interface design, their use is suggested; (*b*) HTTP methods usage defined for Web-based Distributed Authoring and Versioning (WebDAV) [8] is a plus to respect the uniform interface (e.g., see MOVE and COPY) yet might lead to deviations from "make it as resource" paradigm. Their suggestion was to create lock collections and manipulate them like all the others collections (by using POST, GET, PUT, DELETE) in place of using LOCK and UNLOCK methods to resource URI.

2.3 Transaction properties

A transaction is a group of Web service interactions that achieve a logic (sub-)goal within a service composition only if all interactions complete successfully [4]. If error occurs in a transaction, the actions of the transaction that have already been performed must be compensated, that is, rolled back until the status right before the transaction started.

Four properties characterize them and are commonly referred by the ACID acronym: (*i*) *Atomicity*, i.e., it executes completely or not at all, no possibility to execute part of a transaction is allowed; (*ii*) *Consistency*, i.e., the transaction maintains the consistency of the "system" and it is a shared responsibility between the transaction developer and the transaction processing implementation; (*iii*) *Isolation*, i.e., the effect of the entire transaction on the "system" state is equal to the effect of the single interactions executed one by one; and (*iv*) *Durability*, i.e., when a transaction completes executing all its updates are stored.

3 The ReLock approach

ReLock is an approach for transactions management where transactions involves *RESTful services*, i.e., web services compliant with the set of ROA and REST principles discussed in Sect. 2. The approach is called to propose an implementation of the two-phase locking protocol [35]. Moreover, the approach is conceived thus to guarantee the "independence" of both (*i*) RESTful services involved in each transaction, i.e., the development and operation of RESTful services is not impacted by any change for supporting transactions; (*ii*) clients that are not conceived to develop their business logic by relying on transactions.

Figure 1 depicts the ReLock transaction management architecture highlighting the layering of the approach. In practice, the ReLock services implementing the transaction model realize an overlay layer on top of the RESTful service(s) the clients will interface with.

Three services are conceived to implement the ReLock transaction management:

- the *Transaction Proxy* called to intercept all requests made by clients to the target RESTful service and forwards these requests to the RESTful service when suitable according to the transaction management protocol. Hereafter, we will refer to this component as proxy or Transaction Proxy;
- the *Transaction Service* implementing transactions as resource facilities. It also exposes transaction logging facilities as resources. Logging is used to achieve compensations in case of rollbacks;

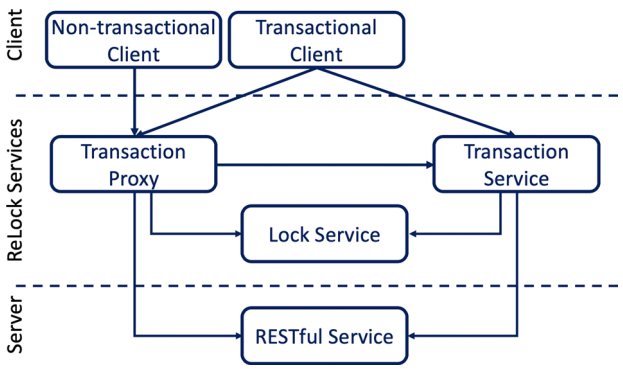

Fig. 1 ReLock transaction model architecture

- the *Lock Service* implementing lock capabilities for resources exposed by the target RESTful service.

Networking policies play a key important role in this architecture. The following rules are defined:

(i) the RESTful service(s) is only accessible from Transaction Proxy and Transaction Service;
(ii) the Lock Service is only accessible from Transaction Proxy and Transaction Service;
(iii) the Transaction Proxy intercepts every request coming from clients and directed to the target RESTful service.

Two typologies of *clients* are envisaged in the ReLock approach: non-transactional clients and transactional clients.

A *Non-transactional Client* is either a client which is not aware of ReLock and its transaction model (e.g., any legacy client) or a client which is not interested in creating a transaction. This typology of client behaves as it would have been if there were no transactions. It performs REST requests to the RESTful Service and uses the responses (intercepted and managed by the Transaction Proxy) to continue its workflow.

A *Transactional Client* is a client willing to benefit from transactions. Its interaction with the ReLock services and the target RESTful service(s) is described hereafter. It discovers the Transaction Service by requesting `OPTIONS` to the resource collection URI. The request is intercepted and managed by the Transaction Proxy which provides the list of supported Transaction Services. The client creates a transaction by sending a `POST` to the Transaction Service and then it requests all the transaction operations via the Transaction Proxy. This client always sends the owned transaction URI in the header of the HTTP request (using `X-Transaction-URI` header). The Transactional Client collects every lock URI it receives (via `X-Lock-URI` header in the response) and associates them to the proper resource (using the resource URI). Anytime the Transactional Client requests an action, it indicates the owned locks using the HTTP headers. If a Transactional Client does not include the lock it already owns, it gets the error code `423 Locked` [8, Section 11.3]. If a Transactional Client receives a parent lock URI, it associates the lock to the resource collection which it sends within any succeeding resources creation and deletion requests. The Transactional Client can terminate the transaction by sending either a commit (by using `PUT`) or rollback (by using `DELETE`) request.

ReLock services support both XML and JavaScript Object Notation (JSON) as the content format to represent the resources involved in the transaction. Thus the proposed solution does not enforce transactional clients to deal with different formats, the content format adopted by the RESTful Service is made available by using the format selected by the client for managing the transaction. The transactional client indicates the required format resources by indicating it (i.e., `application/xml` or `application/json`) in `Accept` HTTP header [12, Section 5.3.2] or in `Content-Type` HTTP header [12, Section 3.1.1.5]. Each ReLock service replies indicating the format in `Content-Type` header field according to the received request. In this paper, we present all the examples using the JSON format.

In the remainder of the section, the details of the ReLock approach are presented by first showcasing a sequence diagram of a transaction supported by ReLock and then describing the behavior of the Transaction Proxy, the Transaction Service, and the Lock Service.

3.1 The sequence diagram of a transaction

Figure 2 shows a sequence diagram representing the interactions among the different components involved in a transaction consisting of a read and an update operation on a resource made by a Transactional Client.

This diagram highlights how the entire process is mediated by the ReLock services that catch and manage every request originated from the client to interact with the target RESTful service.

3.2 The ReLock Transaction Proxy

The Transaction Proxy intercepts all requests directed to every RESTful Service. It forwards the requests to the RESTful service only after it has performed the actions required to guarantee the transaction properties (c.f. Sect. 2.3).

Listing 1 describes the algorithm the Transaction Proxy uses to manage the requests.

As first action, the Transaction Proxy extracts the content of the request type (Listing 1: line 2) to use it both for (*a*) any interaction with ReLock Services and (*b*) to generate a suitable response independently of the response format offered by the RESTful Service.

Then, the Transaction Proxy checks the HTTP Method. If the HTTP method is `OPTIONS` (line 4), it sends the list of supported Transaction Services to the client (Listing 1: line 5). It exposes the `OPTIONS` Application Programming Interface (API) to the resources collection shown in Table 3.

An example of `OPTIONS` response is shown in listing 2. `OPTIONS` allows any Transactional client to know which Transaction Service(s) can be used to perform the transaction on the RESTful Service with no prior knowledge (see Sect. 5).

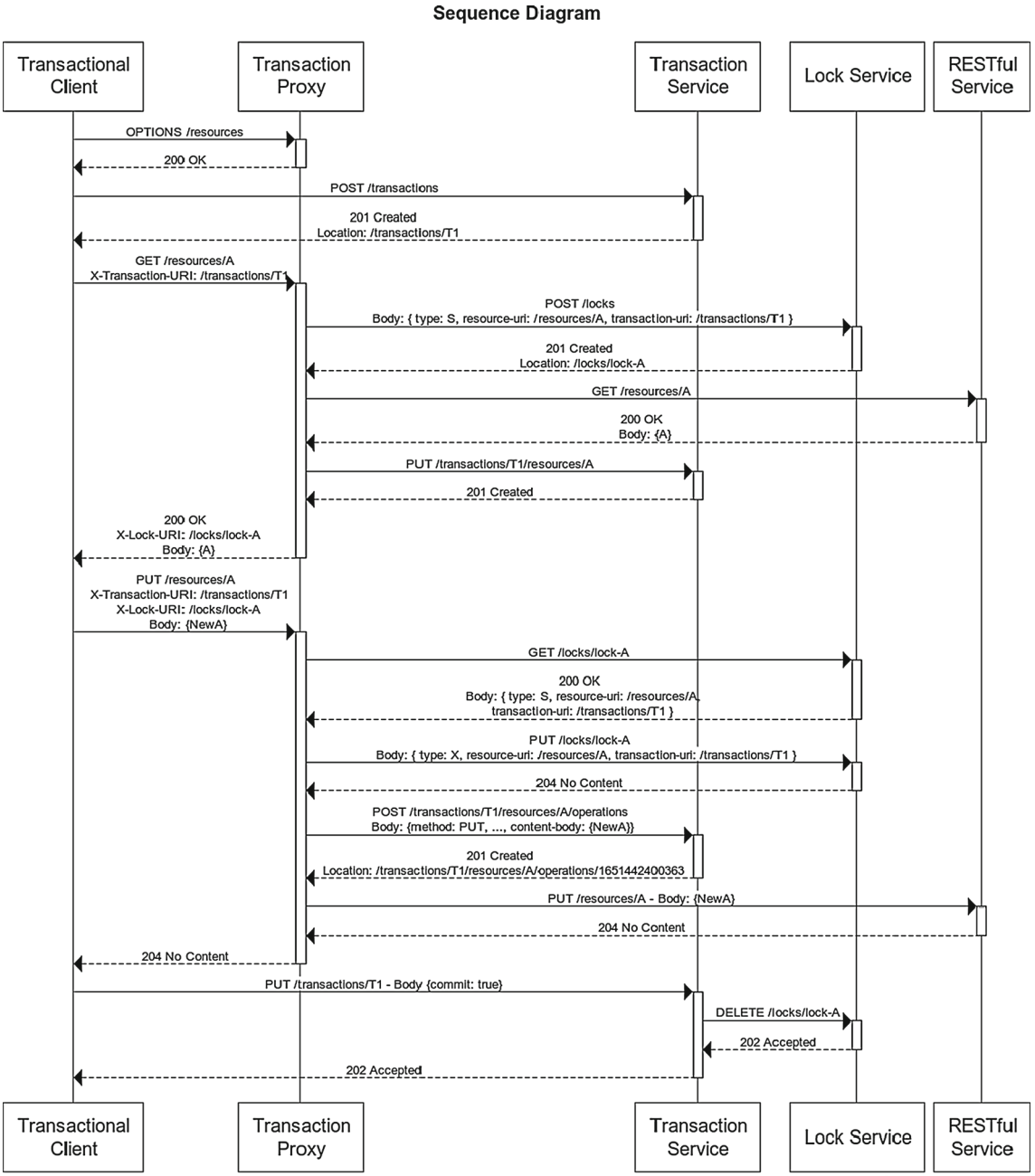

Fig. 2 Example of a sequence diagram showing the interactions among the components involved in a transaction. A transactional client creates the transaction, reads a resource, and then updates such a resource. Finally, the client commits the transaction

Table 3 Transaction Proxy exposed APIs

Operation	HTTP method	URL
Supported Transaction Service	`OPTIONS`	`/resources`

Listing 1 Pseudo Code for Proxy function receiving HTTP request from clients

```
1  function onReceive(httpRequest):
2      global var contentType = getContentType(httpRequest)
3
4      if httpMethod == OPTIONS then
5          return sendSupportedTransactionServicesList()
6      endif
7
8      if httpMethod in [ HEAD, GET, PUT, DELETE ] then
9          var transactionURI = httpRequest.getHeader("X-Transaction-URI")
10         if transactionURI != null then
11             // Transactional Client
12             return manageTransactionAction(httpRequest, transactionURI)
13         else
14             // Non-Transactional Client
15             return createMiniTransaction(httpRequest)
16         endif
17     endif
18
19     // POST and other methods are not allowed
20     return sendErrorResponseToClient(405, "Method Not Allowed")
```

Listing 2 Example of Transaction Proxy response to `OPTIONS` request

```
{
   "transaction-managers": [{
     "uri": "http://transaction.
        example.org/transactions"
   }]
}
```

If the HTTP method is not one of `HEAD`, `GET`, `PUT` or `DELETE` (Listing 1: lines 8 and 20, the Transaction Proxy replies to the client with an error `405 Method Not Allowed` [12, Section 6.5.5].

When the HTTP method is supported, the Transaction Proxy checks if the request contains the HTTP header `X-Transaction-URI` (Listing 1: line 9, 10). Requests containing this header (Listing 1: line 12) came from a Transactional Client and are managed as described in Sect. 3.2.1. Instead, requests arriving without `X-Transaction-URI` (Listing 1: line 15) came from a Non-Transactional Client and it are managed like a "mini transaction" as described in Sect. 3.2.2.

3.2.1 Managing requests from transactional clients

Listing 3 shows how the Transaction Proxy manages requests in the context of a transaction.

First of all, it checks whether the request contains any references to previous obtained locks (i.e., lockURI, Listing 3: lines 2 and 3).

If the Transactional Client does not provide any lock reference then the Transaction Proxy creates the lock for the target resource URI by the *createLock()* function (line 4). The *createLock()* function sends a `POST` request to the Lock Service that creates the lock representation by using the provided argument as discussed in Sect. 3.4 (Listing 9). The HTTP Method (i.e., httpMethod) argument is used to specify the lock type. Table 4 shows the mapping between the HTTP Method and the required lock type. When the lock is granted, the Transaction Proxy reads the resource on the RESTful Service with `GET` (Listing 3: line 5) and creates the initial resource on the Transaction Service (cf. Sect. 3.3, Listing 3; line 7) using the function *createInitialResource()*.

If the Transactional Client provides the lock URI, then the Transaction Proxy verifies and eventually upgrades the lock (from Shared to eXclusive depending on the requested HTTP Method) by invoking the Lock Service (Sect. 3.4) (Listing 3: line 9).

Lines 14–28 in Listing 3 deal with the management of *parent locks* (i.e., locking at the collection containing the target resource):

- A Transactional Client owning the parent lock URI (i.e., `X-Parent-Lock-URI` HTTP header) always sends it in any request. The Transaction Proxy verifies this header (Listing 3: line 16) as it does for `X-Lock-URI` HTTP

Table 4 Transaction Proxy mapping with HTTP method and required Lock

HTTP method	Required lock type
`POST`	Not supported
`HEAD`	Shared
`GET`	Shared
`PUT`	eXclusive
`DELETE`	eXclusive

Listing 3 Pseudo Code for Proxy function which handles any transaction action

```
1  function manageTransactionAction(httpRequest, transactionURI):
2      var lockURI = httpRequest.getHeader("X-Lock-URI")
3      if lockURI == null then
4          lockURI = createLock(httpMethod, transactionURI, resourceURI)
5          var response = getResource(resourceURI)
6          var content = response.content
7          var initialResourceURI = createInitialResource(transactionURI,
               resourceURI, lockURI, content)
8      else
9          verifyAndUpgradeLock(httpMethod, lockURI, transactionURI,
               resourceURI)
10         var initialResourceURI = getInitialResourceURI(transactionURI,
               resourceURI)
11         var content = getContentFromInitialResource(initialResourceURI)
12     endif
13
14     if httpRequest.getHeader("X-Parent-Lock-URI") != null then
15         var parentLockURI = httpRequest.getHeader("X-Parent-Lock-URI")
16         verifyParentLock(parentLockURI, transactionURI)
17     else
18         if httpMethod == PUT AND content == null then
19             var create = true;
20         endif
21         if create OR httpMethod == DELETE  then
22             var parentURI = getParentURI(resourceURI)
23             var parentLockURI = createLock(httpMethod, transactionURI,
                   parentURI)
24             // initial resource of collection is not needed
25         endif
26     else
27
28     endif
29
30     logRequest(initialResourceURI, httpRequest)
31
32     HttpResponse actionResponse = forwardAction(httpRequest)
33
34     actionResponse.setHeader("X-Lock-URI", lockURI)
35     if parentLockURI != null then
36         actionResponse.setHeader("X-Parent-Lock-URI", parentLockURI)
37     endif
38     return actionResponse
```

header (Listing 3: line 14). The Transaction Proxy adds the parent lock URI to the response header if any (Listing 3: lines 35–37).

- If the Transaction Proxy receives a `PUT` request, it checks it to discriminate between a create and an update request (Listing 3: lines 18–20). A create request is identified by checking the initial content of the resource (Listing 3: line row 18).
- If a client sends a request either to delete or to create a resource, the Transaction Proxy also locks the parent URI (which represents the resource collection) with an eXclusive lock (Listing 3: lines 22 and 23). This lock is required for a proper implementation of the isolation property (see Sect. 5).

The Transaction Proxy logs the action with a `POST` to the collection "operations" subordinated to the initial resource before forwarding the request to the target RESTful service (line 30). The `POST` content contains the received HTTP request information (see Listing 8);

Finally, the Transaction Proxy forwards the request to the target RESTful service and collects the received response (Listing 3: line 32). The Transaction Proxy adds the lock URI (Listing 3: line row 34) to the received response using `X-Lock-URI` HTTP header and returns it to the requesting client (Listing 3: row 38).

3.2.2 Managing requests from non-transactional clients

When the Transaction Proxy receives a request not containing any transaction reference, the client is considered Non-transactional and the request is managed according to Listing 4.

The Transaction Proxy creates a transaction (line 2), then it manages the requested operation like a request coming from a Transactional client (line 3) just discussed in Sect. 3.2.1. The Transaction Proxy commits the transaction (line 4) on behalf of the client. It also cleans the obtained response from the additional header before returning the response to the Non-transactional Client (lines 5–8)

In practice, every request made from a non-transactional Client is de-facto a mini-transaction involving a single operation on the target RESTful service.

3.3 The ReLock transaction service

The Transaction Service accepts `POST`, `PUT` and `DELETE` methods to manage transactions (see Table 5). Worth highlighting that this service does not allow the use of `PUT` to create a new transaction because it is for other intents.

The Transaction Service receives a request to *create a new transaction* by a `POST`. The URI of the created transaction resource is returned in `Location` HTTP header [12, Sec. 7.1.2]. The HTTP `201 Created` status code is returned to indicate the request succeeded [12, Sec. 6.3.2]. Listing 5 contains the content of the response containing (i) the 'timestamp' of the transaction i.e., unix timestamp expressed in milliseconds; (ii) the 'timeout' i.e., the amount of time (expressed in milliseconds) the transaction will be automatically rollbacked if not committed or rollbacked from the client; (iii) the protocol version i.e., 1.0.

Listing 5 Example of representation of a transaction resource

```
{
   "timestamp": 1651442400000,
   "timeout": 2400,
   "protocol-version": "1.0"
}
```

In case of failure (i.e., the transaction cannot be created), either a `4xx` or a `5xx` error [12, Section 6.5, Section 6.6] is returned to client depending on which failure occurred.

The Transaction Service receives a request to *commit a transaction* by a `PUT`. The Transaction Service allows only to add the property "commit" with value true to the representation of the transaction resource (Listing 6).

Listing 6 Example of representation of transaction resource sent to commit the transaction

```
{
   "timestamp": 1651442400000,
   "timeout": 2400,
   "protocol-version": "1.0",
   "commit" : true
}
```

Once the property *commit* is set to *true* the transaction is closed and the Transaction Service denies any further operation on the transaction resource and subordinates by returning `403 Forbidden` [12, Sec. 6.5.3].

The Transaction Service receives a request to *rollback a transaction* by a `DELETE`. If the target transaction resource exists and it has not been already committed, the Transaction Service starts a compensation procedure (see Sect. 3.3.1). The Transaction Service starts the compensation procedure also if the transaction expires due to the timeout being exceeded.

3.3.1 The ReLock compensation approach

The compensation procedure is activated whenever a client sends a request to rollback a transaction or the transaction timeout expires before the commit. It consists in restoring the overall system state thus to make it consistent like the transaction was not initiated at all.

In order to support this procedure, the Transaction Service introduces *initial resources* and a *logging mechanism*.

For each ongoing transaction, the Transaction Service allows the Transaction Proxy to create subordinated resources (i.e., "resources that exist in relation to some other "parent" resources" [30]). The Transaction Proxy creates as subordinates of the transaction the *initial resources*, i.e., resources storing the representation of any resource of a target RESTful service partaking to the transaction before any attempt of modification to the resource sent to the target RESTful service. Listing 7 shows an example of the representation of the initial resource containing: (i) the resource URI; (ii) the URI of the lock conceived by the Lock Service to access the resource available to the resource URI; (iii) the content type of the resource representation; (iv) the initial representation of the resource on the effective service.

Listing 7 Example of initial resource

```
{
   "resource-uri": "http://example
      .org/resources/A",
   "lock-uri": "http://lock.
      example.org/locks/lock-a",
   "content-type": "application/
      json"
   "content": {
     // Representation of resource
   }
}
```

Listing 4 Pseudo Code for Proxy function which handles non-transactional client requests

```
function createMiniTransaction(httpRequest):
    String transactionURI = createTransaction()
    var actionResponse = manageTransactionAction(httpRequest, transactionURI)
    commitTransation(transactionURI)
    actionResponse.removeHeader("X-Lock-URI")
    actionResponse.removeHeader("X-Parent-Lock-URI")
    actionResponse.removeHeader("X-Transaction-URI")
    return actionResponse
}
```

Every initial resource is created by a `PUT` to the URI calculated by composing the transaction URI with the relative URI of the resource on the target RESTful service (e.g., http://transactions.example.org/transactions/T1 + `/resources/A` results in http://transactions.example.org/transactions/T1/resources/A).

For each initial resource, the Transaction Service exposes a collection, named "operations", to enable logging of the operations made by the client to the RESTful service's resource (via Transaction Proxy). The Transaction Service allows log creation by `POST` to the collection "operations" e.g., `POST` http://transactions.example.org/transactions/T1/resources/A/operations. Listing 8 shows an example of a log resource which contains the HTTP request made by the client.

Listing 8 Example of log resource

```
{
  "method": "PUT",
  "headers": {
    "Accept": "application/json",
    "Content-Type": "application/
       json"
  },
  "content-body": { ... }
```

The URI of the created log resource will be created by appending the timestamp to the log collection URI (e.g., http://transactions.example.org/transactions/T1/res/A/operations/1651442400363)

The *compensation procedure* consists in analyzing all the subordinated resources of the transaction (initial resources and logs) putting in place the compensation operations summarized in Table 6. The resources accessed only with safe operations (`GET` or `HEAD`) do not require any action. Updated resources require (`PUT`) compensation to the initial values. Deleted resources have to be re-created with a `PUT` to the original resource URI and using the initial values. The resources created within the transaction must be deleted (by using `DELETE`).

3.4 The ReLock lock service

The Lock Service provides locking capabilities. It exposes two types of locks: eXclusive (X) and Shared (S). A shared lock allows multiple clients to read the referenced resource but does not allow any client to modify such a resource. Multiple shared locks can exist for the same resource. The

Table 5 Transaction service RESTful APIs

Operation	HTTP method	URL
Create Transaction	`POST`	`/transactions`
Commit a Transaction	`PUT`	`/transactions/{TRANSACTION_ID}`
Rollback a Transaction	`DELETE`	`/transactions/{TRANSACTION_ID}`
Create Initial Resource	`PUT`	`/transactions/{TRANSACTION_ID}/{RESOURCE_RELATIVE_PATH}`
Create Log Resource	`POST`	`/transactions/{TRANSACTION_ID}/{RESOURCE_RELATIVE_PATH}/operations/`

Table 6 Compensation operations

CRUD operation	HTTP method	Compensation HTTP method	Compensation body content
Create	`PUT`	`DELETE`	No
Exists	`HEAD`	Unneeded	N/A
Read	`GET`	Unneeded	N/A
Update	`PUT`	`PUT`	Initial representation
Delete	`DELETE`	`PUT`	Initial representation

Table 7 Lock compatibility table

	Shared	eXclusive
Shared	True	False
eXclusive	False	False

Table 8 Lock service RESTful APIs

Operation	HTTP method	URL
Create	`POST`	`/locks/`
Read	`GET`	`/locks/{LOCK_ID}`
Update	`PUT`	`/locks/{LOCK_ID}`
Delete	`DELETE`	`/locks/{LOCK_ID}`

exclusive lock allows only one client to modify (and read) the referenced resource.

The algorithm to grant a lock is pretty straightforward. The Lock Service checks only the locks for the correspondent resource (considering the URI) and applies the following rules summarized in Table 7: (*i*) when a Shared lock is present on a certain URI other Shared locks can be granted to other clients but not an eXclusive lock; (*ii*) when an eXclusive lock is present on a certain URI no other locks can be granted to other clients. Moreover, a client holding a Shared lock can obtain an upgrade of the lock from Shared to eXclusive if and only if there are no other clients holding a Shared locks related to the same URI. Lock service does not allow to change an eXclusive lock to a Shared one. This procedure is pretty straightforward and allows to efficiently implements the algorithm which is expected to be very fast.

Lock Service accepts `GET`, `POST`, `PUT` and `DELETE` methods to manage locks as summarized in Table 8: `POST` creates a new lock for a resource; `PUT` can only modify a lock type; `DELETE` removes a lock; `GET` allows reading the lock resource representation. This service does not allow the use of `PUT` to create a new lock.

When the Lock Service receives a request to create a lock by a `POST` request and grants the lock, it creates the resource representing such a lock. Listing 9 presents an example of the representation of lock resource.

Listing 9 Example of lock resource

```
{
   "type": "X",
   "resource-uri": "http://example
      .org/resources/A",
   "transaction-uri": "http://
      transaction.example.org/
      transactions/T1"
}
```

The URI of the created lock resource is returned in `Location` HTTP header. The HTTP `201 Created` status code is returned to indicate the requested succeeded.

The Lock Service receives a request to modify a lock by `PUT` request it allows only to upgrade the lock type from 'S' to 'X'. It verifies if the lock upgrade can be conceived and updates the resource. The Lock Service can use either `200 OK` [12, Sec. 6.3.1] or `204 No Content` [12, Sec. 6.3.5] status code to indicate that the request succeeded. The use of `202 Accepted` [12, Sec. 6.3.3] which is used for asynchronous operations is not allowed.

When the Lock Service receives a request to delete a lock with `DELETE`, it removes the associated lock resource. The Transaction Service is the only service authorized to invoke the removal of a lock and it does it either on commit or if the rollback procedure (c.f. Sect. 3.3.1) is terminated.

3.5 Resiliency

ReLock services are stateless and can be replicated and load balanced to improve scalability and availability.

The Transaction Service annotates the progress of the rollback by annotating each step on the resources representing the transaction. In particular, the Transaction Service

- adds a property to the transaction resource representation to indicate it is starting the rollback;
- for each initial resource representation:
  - adds a property to indicate it is going to compensate the corresponding resource;
  - modifies the previous property to indicate that the corresponding resource has been compensated;
  - add a property to indicate it is going to release the associated lock;
  - modifies the previous property to indicate it has released the associated lock;
- modifies the properties on the transaction resource representation to indicate that the rollback is terminated successfully.

This procedure behaves like a journal which enables any Transaction Service to take in charge the rollback procedure of another instance of the service by restarting it from the last completed step.

Clearly, the Lock Service must prevent starvation and deadlock. It is possible to replicate and distribute the Lock Service by using one of the distributed deadlock detection algorithms surveyed in [10].

4 Related work

Existing solutions for transaction management on RESTful services are reported in Table 9 and briefly discussed below.

The overloaded `POST` pattern is the oldest approach to RESTful transactions known [20,21]. It is characterized by putting several HTTP operations in the payload of a single `POST` operation. This approach fits well for batched and short-lived transactions. Unfortunately, the approach does not respect the HATEOAS constraint (cf. Sect. 2.1) and is not suitable for distributed transactions.

In 2007, Richardson and Ruby proposed a transaction as resource approach [30]. In their proposal, the client opens a transaction by creating a resource to a transaction service. The client performs every subsequent operation by creating a resource with `PUT` as a subordinate of the transaction resource. This solution supports only the resources creation.

Khare [13] proposed an enhancement of the REST architectural style for distributed and decentralized systems which includes five different extensions. One of these extensions, REST with Delegation (REST+D), dealing with Atomicity, Isolation, Durability and Consistency (ACID) transactions, proposes a mutex lock proxy component which provides mutually exclusive access to the origin server and ensures total serialization of all updates to a resource. Their proposal, like REST which tries to extend, is an architectural style and it "does not provide any details regarding how to execute the scenario" [21].

da Silva Maciel et al. [31] proposed an optimistic technique. In their model, the REST service must support resource versioning. The proposal uses compensation techniques to rollback a transaction, which is done within locks.

Marinos et al. proposed RETRO [18,29] which uses the concepts of transaction as resource, locks and temporary resources to achieve isolation. Their solution uses the hyperlink to meet HATEOAS constraint. The temporary resources approach introduces link transparency issues [21]. Our solution shares some ideas with RETRO, i.e., transaction as resource, locks and use of hyperlinks but instead of using the concept of temporary resources, it uses an approach based on initial resource representation and logging as resource to support rollbacks via the appropriate compensation technique. RETRO uses non-standard HTTP methods and header. On the contrary, our solution uses standard HTTP methods only.

da Silva Maciel et al. [32,33] proposed a timestamp-based two-phase commit RESTful transactions (TS2PC4RS) designed for reservation-based services. In their last version [34], they also introduced the concept of logs "as a fault-tolerant mechanism capable of recovery connection and server failures". ReLock does not require the effective service to adhere to a specific pattern. ReLock uses the logs to support compensation and resiliency, but we expose logs as resource to fully comply with ROA.

Pardon and Pautasso [26,27] proposed a try-cancel/confirm (TCC) approach. Their solution shares some ideas with TS2PC4RS, but TCC is only applicable if the reservation "fits directly into the business model" [27] (i.e., a ticket reservation). ReLock does not require services to adhere to any particular business model.

Kochman et al. [15] proposed a solution based on batched transactions which use mediators and proxies. Their solution allows transactional and non-transactional clients to coexist. Our solution uses a proxy to support the same clients, but we use a two-phase locking protocol instead of the batched transactions.

Dey et al. proposed REST with Transaction (REST+T) [7]. REST+T is not stateless and uses non-standard HTTP methods.

5 Discussion

Table 10 summarizes the results of the analysis of ReLock and the solutions for transaction management listed in Table 9 and discussed in the previous section. In particular, the analysis is based on a set of 20 properties aiming at assessing the capability of the various solutions to satisfy transaction properties, RESTfulness properties, HTTP properties, and miscellaneous properties. The set of properties was initially proposed by Mihindukulasooriya et al. [20,21] and extended in this work.

Regarding the ability of the various approaches to provide *transaction properties* (cf. Sect. 2.3), only atomicity and isolation are considered because consistency and durability are guaranteed by the implementation of clients and RESTful services. ReLock guarantees the properties considered thanks to the protocol implemented by the Transaction Service (cf. Sect. 3.3). In fact, a client reads the resource collection by using the HTTP `GET` method. The only way to modify the resource collection is by either deleting a resource or creating a new one. The Lock service grants a lock only by analyzing if another lock exists for the same resource (by using its URI). To delete a resource, the client performs a `DELETE` to the resource URI. ReLock requires two exclusive locks, one for the resource and one for the resource collection giving that it is going to modify two representations (the resource and the collection). The lock for the resource prevents that any other client can interact with such a resource. The lock for the collection prevents that any other client can interact with the resource collection. To create a resource, the client has to use the HTTP `PUT` request. With `PUT`, the client performs a request to the URI where the resource will be available. The resource creation with `PUT` requires two exclusive locks, one for the resource and one for the resource collection for the same considerations of the delete operation.

Table 9 Approaches for transaction management on RESTful services

	Approach	Year	Refs.
#1	Batched transaction with overloaded `POST`	2000	[20,21]
#2	Transaction as resource	2007	[30]
#3	Optimistic technique for transaction using REST	2009	[31]
#4	A consistent and recoverable RESTful transaction model (RETRO)	2009	[18,29]
#5	Timestamp-based two phase commit protocol for RESTful services (TS2PC4RS)	2010	[32,33]
#6	Try-Cancel/Confirm pattern (TCC)	2011	[26,27]
#7	Atomic REST batched transactions	2012	[15]
#8	REST+T	2015	[7]

Table 10 Analysis of RESTful transaction approaches

Property	Approach								
	#1	#2	#3	#4	#5	#6	#7	#8	ReLock
Transaction properties									
Atomicity	Y	Y	Y	Y	Y	Y	Y	Y	Y
Isolation	Y	Y[1]	N	Y	N	N	Y	Y[2]	Y
REST properties									
Stateless	Y	Y[3]	N	Y[3]	N	Y	Y	N	Y
Uniform interface	Y	N	Y	Y	Y	Y	Y	Y	Y
Identification of resources	N	Y	N	Y	N	Y	N	Y	Y
Manipulation of resources through representations	Y	Y	Y	Y	N	N	Y	Y	Y
Self-descriptive messages	Y	Y	Y	Y	N	N	Y	Y	Y
HATEOAS	N	N	N	Y	N	N	N[4]	N	Y
Layered system	Y	Y	N	Y	N	N	Y	N	Y
HTTP related properties									
Semantic not violated	Y	Y	Y	Y	N	Y	Y	?	Y
Common verb supported	Y	Y	N	N	Y	Y	Y	N	Y[5]
HTTP Methods adherence	Y	Y	N	N	Y	Y	Y	N	Y
Low overhead	Y	Y	Y	N	N	Y	Y	Y	Y[6]
ROA URI adherence	Y	Y	Y	Y	Y	Y	Y	Y	Y
Miscellaneous properties									
CRUD operations supported	?	?	Y	RU	CR(U)	CD	Y	Y	Y[7]
Optionality	Y	?	?	Y	?	?	Y	N	Y
Discoverable	?	?	?	Y	?	?	Y	N	Y
Distributed transactions	N	N	Y	?	Y	Y	?	?	Y
Service paradigm required	N	N	Y	?	Y	Y	N	Y	N
Heterogeneity of service types	N/A	N/A	Y(?)	?	?	N	?	?	Y

[1]Possible lost update problem
[2]Clients are made aware of pending uncommitted actions
[3]Temporary resources having uniform resource locator (URL) have been criticized
[4]Could be added
[5]Three non standard HTTP header field
[6]Clients perform three additional requests. The first uses `OPTIONS` to discover the Transaction Service. The client may cache the result. Hence, the client performs the `OPTIONS` request only before the first transaction. The remaining two requests are used to create and commit (or rollback the transaction). It is the minimum number of additional operations of every transaction mechanism.
[7]Create is supported via `PUT` (not via `POST`)

It is also worth highlighting that ReLock works with RESTful services which do not provide nested resource collection (e.g., the bookmark ROA example presented in Table 1). To delete a resource having subordinates, no one should be capable of interact with the subordinates. To support this scenario, the Lock Service should be enhanced to use a different algorithm that supports subordinated resources. Marinos et al. [18,29] have formally demonstrated that their locking mechanism (approach #4) is well formed and sound. However, it supports only read and update operations with a two-phase locking protocol by releasing acquired locks only at commit or rollback time. ReLock follows a similar approach. ReLock also supports the creation (via HTTP `PUT` request only) and the deletion of resources. These operations also generate an update of the resource collection (which is a resource per se), and therefore our proposal provides a solution to safely interact with the resource collection besides the created/deleted resource.

Regarding the *RESTfulness of the proposed approach*, i.e., the adherence of the overall solution with respect to the REST principles (cf. Sect. 2.1): (*a*) ReLock services are *stateless*. The Transaction Service saves an application state [30, p. 90] to support compensation and resiliency (cf. Sect. 3.5). Conversely from approach #4 [18,29], such an application state does not create temporary resources thus avoiding the link transparency issue [20, Sec. 3.3]. (*b*) the ReLock approach is layered, it respects the uniform interface constraint, and the fours additional constraints the uniform interface introduces (cf. Sect. 2.1).

Regarding the *compliance with HTTP*: (*a*) the ReLock approach respects the semantics of HTTP 1.1 methods. The usage of `PUT` method to commit the transaction could be disputed. ReLock services use three non-standards HTTP headers: `X-Transaction-URI`, `X-Lock-URI` and `X-Parent-Lock-URI`. `Lock-Token` header defined for WebDAV [8, Sec. 10.5] could be a possible standard alternative to `X-Lock-URI`; (*b*) the resource collection URI returns the list of the resources (list of URIs) according to ROA. Every URI can be considered a resource per se.

Regarding the *miscellaneous properties* considered: (*a*) ReLock supports *CRUD operations*. However, resource creation with `POST` is problematic since the URI of the resource created by `POST` is not known, and the Transaction Proxy cannot request a lock for the created resource URIs. The lock for the resource collection is not enough because a concurrent client could create a resource with `PUT` directly to the resource URI (which could result in an update which violates the isolation property). The Transaction Proxy could transform a `POST` request into a `PUT` request only if it is capable of defining the appropriate URI for the resource. This is possible only if the Transaction Proxy knows the semantic of the target RESTful Service. For this reason, our transaction model could support `POST` only when it is exploited to implement transactions on "known" RESTful Service(s). ReLock can support the `POST` Method but the Proxy must be tailored for the Effective Service. Anyway, it is be possible to develop a Generic Transaction Proxy which could use a mix of configurations and plug-in-based approach to support such a method; (*b*) clients not conceived to rely on ReLock can coexist with the ones exploiting it, thus ReLock cater for *optionality*. Services are always not aware of the extended behavior introduced by our transaction model, and they do not need to conform to any particular pattern. They have only to be RESTful and do not expose subordinated resources; (*c*) *discoverability* is achieved by the `OPTIONS` method. This means that "all the metadata needed to execute the transactions can be discovered in a RESTful manner without out-of-band knowledge (i.e., following links)" [20]. By exploiting this feature, our model can exploit the out-of-band knowledge to support a distributed transaction. By querying all the involved proxies, a client discovers if a common transaction service exists. If it exists, then all the lock services used by the proxies can interact by implementing a protocol for deadlock detection; (*d*) ReLock supports RESTful *distributed transactions* [13,14] under the following additional constraints. First of all, all the services should trust a common Transaction Service. The Lock services must share a common distributed deadlock detection algorithm. Moreover, distributing the transactions will imply to build the initial resource URIs encoding the effective services base URI. This will allow to distinguish different services and expose their collections to the same relative URI; (*e*) ReLock is not posing any request for RESTful service(s) to adhere to specific service paradigms.

5.1 Common transaction scenarios

Nine scenarios have been proposed [21] to evaluate the coverage of any RESTful transaction model.

In Scenario I, two resources belonging to a single application are updated. Table 11 shows how our protocol supports such a scenario. Our model requires seven client calls to execute the scenario instead of the four ideals calls (3–6). Three additional calls represent 75% of overhead in the presented scenario, but this number is constant because the number does not change with the number of operations made in the transaction.

Table 12 presents Scenario II using `PUT` (`POST` is supported in certain conditions only).

Scenario III is similar to Scenario I, but the update operation must be asynchronous because the call is expected to take a long time to execute. In this case, the Response to the client `PUT` operation is `202 Accepted` in place of `204 No Content`. Our proposal could support this scenario, but it does not have a good fit for long-running transactions because it uses pessimistic locks which block resources.

Table 11 ReLock and Scenario I: a transaction that updates two resources

HTTP method	URL	Response
OPTIONS	http://example.org/resources	200 OK
PUT	http://transaction.example.org/transactions/	201 Created Location: http://transaction.example.org/transactions/T1
GET	http://example.org/resources/A	200 OK
GET	http://example.org/resources/B	200 OK
PUT	http://example.org/resources/A	204 No content
PUT	http://example.org/resources/B	204 No content
PUT	http://transaction.example.org/transactions/T1	204 No content

Table 12 ReLock and Scenario II: a transaction involving update, creation, and deletion operations of a resource

HTTP method	URL	Response
OPTIONS	http://example.org/resources	200 OK
PUT	http://transaction.example.org/transactions/	201 Created Location: http://transaction.example.org/transactions/T1
GET	http://example.org/resources	200 OK
PUT	http://example.org/resources/C	201 Created
GET	http://example.org/resources/A	200 OK
PUT	http://example.org/resources/B	204 No Content
PUT	http://transaction.example.org/transactions/T1	204 No Content

Scenario IV is also similar to the Scenario I, but the two resources belong to two different applications, see Table 13. We support such a scenario for distributed services but not for decentralized ones. Steps one and two are executed to discover the supported Transaction Service. Clearly, the client can proceed only if it finds a common Transaction Service.

Scenario V is a rollback scenario where a server rejects an update (i.e., the second update) by responding with `409 Conflict` [12, Sec. 6.5.8] because another client updated the same resource before. This scenario cannot occur in our transaction model because the actions of the first client lock the resource and the second client cannot obtain an Exclusive lock for the same resource. In such a case, the second client gets a `403 Forbidden`, while it tries to read the resource. It is a duty of the client either to retry after a delay (Scenario V.a Table 14) or rollback by deleting the transaction resource (Scenario V.b Table 15).

Scenario VI (see Table 16) shows a voluntary rollback of the client.

In Scenario VII, the client fails in the middle of a transaction, for instance after the second PUT in Scenario I. In such a situation, the transaction is not committed. The Transaction Service rollbacks the transaction when the timeout expires.

In Scenario VIII, the server fails in the middle of a transaction, for instance with a '500 Internal Server Error' [12, Sec. 6.6.1]. This situation is similar to Scenario V, the client can decide either to retry with a delay (Scenario V.a) or rollback the transaction (Scenario V.b).

Scenario IX is about communication losses and message losses. Either the request or the response message could get lost due to unreliable network communication. This scenario opens different cases to analyze.

The first case occurs when the client does not receive a reply from the Transaction Proxy while updating a resource. If the Transaction Proxy fails before the lock is requested, then the client is still able to get the lock and proceed by retrying the operation. If instead the Transaction Proxy fails after the lock is obtained (or the reply message is lost due to network issues), then the lock has been set on the resource, but the client does not receive the lock URI. If the client retries the operation, it gets a `403 Forbidden` because the Transaction Proxy is not able to obtain the lock and the client can only rollback the transaction. In any case, the Transaction Service rollbacks the transaction if the timeout expires. Alternatively, the Lock Service could use a `301 Moved Permanently` [12, Sec. 6.4.2] if it recognizes that a Transaction Proxy is trying to create the same lock for the same resource and transaction.

The second case occurs when the client does not receive the response. The client can retry the operation because the `PUT` operation is idempotent.

Table 13 ReLock and Scenario IV: a transaction that updates resources from different applications

HTTP method	URL	Response
OPTIONS	http://example.org/resources	200 OK
OPTIONS	http://remote.example.org/res	200 OK
PUT	http://transaction.example.org/transactions/	201 Created Location: http://transaction.example.org/transactions/T1
GET	http://example.org/resources/A	200 OK
GET	http://remote.example.org/res/B	200 OK
PUT	http://example.org/resources/A	204 No content
PUT	http://remote.example.org/res/B	204 No content
PUT	http://transaction.example.org/transactions/T1	204 No content

Table 14 ReLock and Scenario V with delayed retry: a transaction with multiple updates where the server rejects an update because of a conflict with a parallel transaction

HTTP method	URL	Response
OPTIONS	http://example.org/resources	200 OK
PUT	http://transaction.example.org/transactions/	201 Created Location: http://transaction.example.org/transactions/T1
GET	http://example.org/resources/A	200 OK
PUT	http://example.org/resources/A	204 No content
GET	http://example.org/resources/B	403 Forbidden
GET	http://example.org/resources/B	200 OK
PUT	http://example.org/resources/B	204 No content
PUT	http://transaction.example.org/transactions/T1	204 No content

Table 15 ReLock and Scenario V with rollback: a transaction with multiple updates where the server rejects an update because of a conflict with a parallel transaction

HTTP method	URL	Response
OPTIONS	http://example.org/resources	200 OK
PUT	http://transaction.example.org/transactions/	201 Created Location: http://transaction.example.org/transactions/T1
GET	http://example.org/resources/A	200 OK
PUT	http://example.org/resources/A	204 No content
GET	http://example.org/resources/B	403 forbidden
DELETE	http://transaction.example.org/transactions/T1	202 accepted

Table 16 ReLock and Scenario VI: a transaction with multiple updates where the client rollbacks due to some condition in its business logic

HTTP method	URL	Response
OPTIONS	http://example.org/resources	200 OK
PUT	http://transaction.example.org/transactions/	201 Created Location: http://transaction.example.org/transactions/T1
GET	http://example.org/resources/A	200 OK
GET	http://example.org/resources/B	200 OK
PUT	http://example.org/resources/A	204 No content
DELETE	http://transaction.example.org/transactions/T1	202 accepted

6 Conclusion and future works

Service composition and supporting *transactions* across the composed services are among the major challenges characterizing the service-oriented computing. This paper presented ReLock, an approach for transaction management of RESTful services. In particular, this approach consist of a transaction model and three specific services implementing it: (*i*) the *Transaction Proxy* called to intercept all requests made by clients to the target RESTful service if allowed by the transaction protocol; (*ii*) the *Transaction Service* implementing transactions as resource facilities as well as exposing transaction logging facilities as resources to support compensations in case of rollbacks; (*iii*) the *Lock Service* realizing lock capabilities for resources exposed by the target RESTful service.

ReLock is a good fit for short-lived ACID transactions. ReLock uses two-phase locking. It exposes transaction as resource holding a timeout. When the timeout is raised the situation is managed as when a client explicitly invokes a rollback. The procedure is based on compensation. The compensation procedure uses initial state representation and logging facilities to properly work. The proposed solution is resilient to failures.

The proposed solution is not immune from some limitations.

`POST` could be supported only in a scenario where the RESTful Service(s) are known to the Transaction Proxy (cf. Sect. 5) while services exposing subordinated resources are not supported. Approaches aiming at overcoming these limitations will be integrated in future works.

An extensive assessment of the proposed approach is ongoing to evaluate its efficiency under different operational settings and implementation decisions. The ReLock approach is currently under testing in the context of the D4Science infrastructure [2,3]. This large scale infrastructure supports workflows and processes across diverse services, often not initially conceived to work together.

Thanks to the layered approach, reservation-based services could coexist with non-reservation based. The idea for reservation-based services is to remove lock service and propose a different proxy behind them, which would still take advantage of the transaction service. The transaction service must be extended to use try/cancel pattern in place of compensation for this type of services. This idea will be presented in a future work.

As a future work, we evaluate the possibility to support decentralized transactions. Blockchain [22] technologies could be a way to approach decentralization.

Acknowledgements This work has received funding from the European Union's Horizon 2020 research and innovation programme under Blue Cloud project (Grant Agreement No. 862409).

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Allamaraju S (2010) RESTful web services cookbook: solutions for improving scalability and simplicity, 1st edn. O'Reilly, Newton
2. Assante M, Candela L, Castelli D, Cirillo R, Coro G, Frosini L, Lelii L, Mangiacrapa F, Pagano P, Panichi G, Sinibaldi F (2019) Enacting open science by D4Sscience. Future Gener Comput Syst 101:555–563. https://doi.org/10.1016/j.future.2019.05.063
3. Assante M, Candela L, Castelli D, Cirillo R, Coro G, Frosini L, Lelii L, Mangiacrapa F, Marioli V, Pagano P, Panichi G, Perciante C, Sinibaldi F (2019) The gcube system: delivering virtual research environments as-a-service. Future Gener Comput Syst 95:445–453. https://doi.org/10.1016/j.future.2018.10.035
4. Bernstein PA, Newcomer E (2009) Principles of transaction processing. Morgan Kaufmann, Burlington
5. Bouguettaya A, Singh M, Huhns M, Sheng QZ, Dong H, Yu Q, Neiat AG, Mistry S, Benatallah B, Medjahed B, Ouzzani M, Casati F, Liu X, Wang H, Georgakopoulos D, Chen L, Nepal S, Malik Z, Erradi A, Wang Y, Blake B, Dustdar S, Leymann F, Papazoglou M (2017) A service computing manifesto: the next 10 years. Commun ACM 60(4):64–72. https://doi.org/10.1145/2983528
6. Cabrera F, Copeland G, Cox B, Freund T, Klein J, Storey T, Thatte S (2002) Web services transaction (ws-transaction). Technical report, BEA Systems, International Business Machines Corporation, Microsoft Corporation, Inc. http://xml.coverpages.org/WS-Transaction2002.pdf
7. da Silva Maciel LAH, Hirata CM (2010) A timestamp-based two phase commit protocol for web services using rest architectural style. J Web Eng 9(3):266–282
8. da Silva Maciel LAH, Hirata CM (2013) Fault-tolerant timestamp-based two-phase commit protocol for restful services. Softw Pract Exp 43(12):1459–1488. https://doi.org/10.1002/spe.2151
9. da Silva Maciel LAH, Hirata CM (2009) An optimistic technique for transactions control using rest architectural style. In: Proceedings of the 2009 ACM symposium on applied computing, SAC '09, pp 664–669. ACM, New York, NY, USA. https://doi.org/10.1145/1529282.1529419
10. da Silva Maciel LAH, Hirata CM (2011) Extending timestamp-based two phase commit protocol for restful services to meet business rules. In: Proceedings of the 2011 ACM symposium on applied computing, SAC '11, pp 778–785. ACM, New York, NY, USA. https://doi.org/10.1145/1982185.1982354
11. Dey A, Fekete A, Röhm U (2015) Rest+t: scalable transactions over http. In: 2015 IEEE international conference on cloud engineering, pp 36–41. https://doi.org/10.1109/IC2E.2015.11
12. Dusseault LM (2007) HTTP extensions for web distributed authoring and versioning (WebDAV). RFC 4918. https://doi.org/10.17487/RFC4918
13. Dustdar S, Schreiner W (2005) A survey on web services composition. Int J Web Grid Serv (IJWGS) 1(1):1–30. https://doi.org/10.1504/IJWGS.2005.007545

14. Elmagarmid AK (1986) A survey of distributed deadlock detection algorithms. SIGMOD Rec 15(3):37–45. https://doi.org/10.1145/15833.15837
15. Fielding RT, Reschke J (2014) Hypertext transfer protocol (HTTP/1.1): semantics and content. RFC 7231. https://doi.org/10.17487/RFC7231
16. Fielding RT (2000) Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine
17. Khare R (2003) Extending the representational state transfer (rest) architectural style for decentralized systems. Ph.D. thesis, University of California, Irvine
18. Khare R, Taylor RN (2004) Extending the representational state transfer (rest) architectural style for decentralized systems. In: Proceedings of the 26th international conference on software engineering, ICSE '04, pp 428–437. IEEE Computer Society, Washington, DC, USA
19. Kochman S, Wojciechowski PT, Kmieciak M (2012) Batched transactions for restful web services. In: Proceedings of the 11th international conference on current trends in web engineering, ICWE'11, pp 86–98. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27997-3_8
20. Lampesberger H (2016) Technologies for web and cloud service interaction: a survey. SOCA 10(2):71–110. https://doi.org/10.1007/s11761-015-0174-1
21. Lemos AL, Daniel F, Benatallah B (2015) Web service composition: a survey of techniques and tools. ACM Comput Surv. https://doi.org/10.1145/2831270
22. Marinos A, Razavi A, Moschoyiannis S, Krause P (2009) Retro: a consistent and recoverable restful transaction model. In: Proceedings of the 2009 IEEE international conference on web services, ICWS '09, pp 181–188. IEEE Computer Society, Washington, DC, USA. https://doi.org/10.1109/ICWS.2009.99
23. Martin J (1983) Managing the database environment. Prentice-Hall, Prentice
24. Mihindukulasooriya N, García-Castro R, Esteban-Gutiérrez M, Gómez-Pérez A (2016) A survey of restful transaction models: One model does not fit all. J Web Eng 15(1–2):130–169
25. Mihindukulasooriya N, Esteban-Gutiérrez M, García-Castro R (2014) Seven challenges for restful transaction models. In: Proceedings of the 23rd international conference on world wide web, WWW '14 companion, pp 949–952. ACM, New York, NY. https://doi.org/10.1145/2567948.2579218
26. Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf
27. Nielsen HF, Mogul J, Masinter LM, Fielding RT, Gettys J, Leach PJ, Berners-Lee T (1999) Hypertext Transfer Protocol—HTTP/1.1. RFC 2616. https://doi.org/10.17487/RFC2616
28. Papazoglou MP, Kratz B (2007) Web services technology in support of business transactions. SOCA 1(1):51–63. https://doi.org/10.1007/s11761-007-0002-3
29. Papazoglou MP, Traverso P, Dustdar S, Leymann F (2007) Service-oriented computing: state of the art and research challenges. Computer 40(11):38–45. https://doi.org/10.1109/MC.2007.400
30. Pardon G, Pautasso C (2011) Towards distributed atomic transactions over restful services. In: REST: From research to practice, pp 507–524. Springer, New York, NY. https://doi.org/10.1007/978-1-4419-8303-9_23
31. Pardon G, Pautasso C (2014) Atomic distributed transactions: a restful design. In: Proceedings of the 23rd international conference on world wide web, WWW '14 companion, pp 943–948. ACM, New York, NY. https://doi.org/10.1145/2567948.2579221
32. Pautasso C, Zimmermann O, Leymann F (2008) Restful web services vs. "big" web services: Making the right architectural decision. In: Proceedings of the 17th international conference on world wide web, WWW '08, p. 805–814. Association for Computing Machinery, New York, NY. https://doi.org/10.1145/1367497.1367606
33. Razavi A, Marinos A, Moschoyiannis S, Krause P (2009) Restful transactions supported by the isolation theorems. In: Proceedings of the 9th international conference on web engineering, ICWE '9, pp 394–409. Springer, Berlin. https://doi.org/10.1007/978-3-642-02818-2_32
34. Richardson L, Ruby S (2007) Restful Web Services, 1st edn. O'Reilly, Newton
35. Weikum G, Vossen G (2002) Chapter Four—Concurrency control algorithms. In: Weikum G, Vossen G (eds) Transactional information systems. The Morgan Kaufmann series in data management systems, pp 125–183. Morgan Kaufmann, San Francisco. https://doi.org/10.1016/B978-155860508-4/50005-3