Journal of Web Engineering, Vol. 15, No. 1&2 (2016) 130–169

A SURVEY OF RESTFUL TRANSACTION MODELS: ONE MODEL DOES NOT FIT ALL

NANDANA MIHINDUKULASOORIYA RAÚL GARCÍA-CASTRO,
MIGUEL ESTEBAN-GUTIÉRREZ ASUNCIÓN GÓMEZ-PÉREZ
Center for Open Middleware, Ontology Engineering Group
Universidad Politécnica de Madrid, Spain
{*nmihindu,rgarcia,mesteban,asun*}*@fi.upm.es*

Received April 15, 2014
Revised July 27, 2015

The REpresentational State Transfer (REST) architectural style is getting traction as a light-weight alternative to SOAP-based Web Services in industry for building loosely coupled applications. In addition, the REST architectural constraints induce scalability and the World Wide Web is a great example of a distributed hypermedia system that is built using REST principles. Despite these benefits, one of the main drawbacks of RESTful services is the lack of standard mechanisms to support advanced quality-of-service requirements such as transactions, which are vital to maintain the high-level of consistency required in common enterprise scenarios. To fill this gap, several RESTful transaction models have been proposed in the past decade; the goal of this paper is to survey such transaction models and to analyse them based on the common transactional scenarios that appear in most enterprise systems.

To this end, this paper presents a systematic literature review that was conducted to identify and summarize the state of the art of the RESTful transaction models; the review is followed by a detailed analysis of the models found in the survey. For the analysis, the paper proposes a comparison framework for the RESTful transaction models to evaluate them according to various dimensions, such as their capability to satisfy common transactional scenarios, the level of transaction guarantees provided, compliance to the REST constraints, and other miscellaneous properties. The results of the survey provide a good overview of the current RESTful transaction models and their evolution over the past decades and help to identify the current gaps in the state of the art. In addition, the paper identifies a set of challenges for the current RESTful transaction models by examining the limitations identified in the analysis. A main conclusion of this analysis is that building a generic RESTful transaction model capable of satisfying the requirements of all the scenarios is hard though several models solidly satisfy some specific scenarios in some specific domains.

Keywords: REST, Web applications, Transactions
Communicated by: D. Schwabe & A. Ginige

1 Introduction

The REpresentational State Transfer (REST) architectural style is getting traction as a lightweight alternative to SOAP-based Web Services (also known as "Big" Web Services) in the industry [1]. One of the main reasons for this shift is that the SOAP-based WS-* specification stack has grown significantly and has become extremely heavyweight for simple application integration scenarios. For instance, light-weight approaches are preferred in

cross-platform applications where accessibility from mobile devices with restricted computing resources is vital. The REST architectural principles induce several qualities on the systems such as scalability, loose-coupling and independent evolution, simplicity, and portability [2]. Further, RESTful architectures bring several advantages when building Internet-scale applications by taking the full advantage of the Web infrastructure with features such as caching. Thus, there is a trend of providing RESTful interfaces as well as the SOAP-based Web Services infrastructure that most organizations have in place. The industrial surveys undertaken by analysts at Forrester confirm the shift towards RESTful application architectures; for example, SOAP-only Web Service projects only make up less than 10 percent of the organizations that they have surveyed [3].

However, one of the main criticisms, and also a barrier for adopting RESTful services in the enterprise, is the lack of standard mechanisms to support the advanced quality of service requirements that are essential to enterprises [1]. These requirements include security, support for transactions, reliability, support for business processes, and service composition [1]. This paper focuses on one of those requirements, support for transactions, with the objective of producing a survey of the current RESTful transaction models and analysing them according to common transactional scenarios found in traditional applications (by adapting those scenarios to RESTful systems). This survey also helps to identify the current gaps and challenges in the state of the art with regard to RESTful transaction models.

Transaction support is a must-have quality-of-service requirement for enterprise business scenarios where data consistency is vital. Data consistency is important in business applications because invalid data could lead to undesired consequences such as incorrect execution, monetary losses, and legal issues. The concept of transaction has been a key element in businesses from early days. For example, when goods are exchanged for money through several actions such as one party providing the goods and another paying the money, both actions have to be completed as one atomic unit. In computer applications, a transaction is defined as *a sequence of operations on the physical or abstract application state that can be considered as a single unit of work* [4]. Atomicity, Consistency, Isolation, and Durability (known by the acronym ACID) are identified as the four main properties that a transaction should guarantee. Transactions ensure that systems always end up in a consistent state by establishing that each transaction moves the system from one consistent state to another consistent state. During the execution, each transaction is isolated from other parallel transactions, providing an abstraction layer for system developers so that they do not have to worry about other concurrent operations [4].

On the one hand, RESTful transaction models have been a controversial topic in the REST community where the term "RESTful transactions" is sometimes referred to as an oxymoron for it states that a RESTful design should avoid transactions entirely because transaction support makes a system more complex and, as a result, hinders its scalability. On the other hand, some researchers and practitioners believe that RESTful transactions are a real necessity for the enterprise applications and that support for transactions is an essential element for the wide adoption of RESTful designs in the industry [5]. Though this discussion is still not concluded, when looking at public forums, we can observe the need for transactions in RESTful applications, especially in brownfield development environments (i.e., when the systems built with RESTful designs should coexist with legacy systems and support legacy

processes).

There are two areas of RESTful service design in which transaction support is very relevant: composite REST services and RESTful business workflows. In both use cases, a component that is responsible for a composite service and/or executes the business workflow has to communicate with a set of RESTful services that are possibly distributed and decentralized. To ensure the overall consistency, these actions have to be performed in a transactional manner. In addition, there are many other scenarios such as travel booking scenarios and business exchanges in which a user wants to carry out a set of interactions with multiple REST services in a transactional manner.

The rest of the paper is organized as follows: Section 2 provides a background of the main topics covered in this paper, *i.e.*, the REST architectural style and transactions processing. Section 3 presents the research methodology used in this work which includes a systematic literature review and a comparison framework for RESTful transaction models. Section 4 discusses the RESTful transaction models identified in the systematic literature review, while Section 5 analyses different transaction models with examples for common transaction scenarios applicable to RESTful architectures. Section 6 discusses the different qualities of these transaction models according to the comparison framework and Section 7 presents the challenges for the current RESTful transaction models. Finally, Section 8 draws some conclusions derived from the survey.

2 Background

This section presents a high-level background of the two main topics related to RESTful transaction models: the REST architectural style and the concept of transaction.

2.1 The REpresentational State Transfer architectural style

The REST architectural style, initially known as "HTTP object model", was developed as a means of communicating Web concepts, and it provided the foundation for the modern Web architecture [2]. In his dissertation, Roy T. Fielding explores the different architectural aspects one should consider when building Internet-scale distributed hypermedia systems [6]. One of the goals of that work was to understand, evaluate, and describe the architectural designs that have made the World Wide Web successful and have allowed the Web to become a highly distributed Internet-scale hypermedia system. An architectural style provides guidance on developing concrete architectures for systems that are required to have certain quality characteristics. The architectural aspects analysed include how a system is partitioned into components and layers, how those components are connected to each other and exchange information, and how these components can evolve interdependently [6].

The REST architectural style puts high emphasis on the scalability of the system and the independent evolution of its components. The other properties related to the REST architectural style include generality of interfaces, independent deployment of components, and intermediary components that help reducing interaction latency, enforcing security, and encapsulating legacy systems. REST introduces a set of named, coordinated architectural constraints on hypermedia systems design and each of these constraints induces a set of desirable properties (see Table 1) that enable the development of loosely coupled scalable distributed systems; It also has a set of associated drawbacks.

Table 1. REST constraints, induced properties, and trade-offs

Constraint	**Induced property**	**Trade-offs**
Client-server	portability, simplify server components, independent evolution	-
Stateless	visibility, reliability, scalability	repetitive data, reduced server control
Cache	efficiency, scalability, user-perceived performance	decreased reliability
Uniform Interface	simplicity, visibility, loose coupling, independent evolution	decreased efficiency, not optimal for specific scenarios
Layered system	reduced complexity, scalability	overhead, latency
Code-on-demand (optional)	extensibility	reduced visibility

The REST uniform interface constraint is further elaborated with a set of sub-constraints *resource identification* (addressability), *uniform interface*, *stateless interactions*, *self-describing messages*, and *hypermedia as the engine of application state* (HATEOAS).

However, in recent years, the term "*RESTful*" has been used as a buzz word or a marketing term, and most of the organizations use the term REST as tantamount to using the HTTP protocol at the transport layer without considering the architectural constraints. As a result, most web applications (including those calling themselves *RESTful*) do not consider REST constraints in their designs. Models such as the *Richardson Maturity Model* categorize the systems according to their adherence to the REST constraints [7] and provide an insight into the impact and consequences of dropping these constraints. For example, if the *uniform interface* constraint is not adhered and the interfaces are made resource-specific without being uniform, URIs will not be enough for interacting with the resources and some out-of-band knowledge will be required about the interface offered by the service, which leads to tightly-coupled clients and services. Similarly, if the *stateless interaction* constraint is relaxed, servers will have to maintain the conversational state for each client, making servers more complex and less scalable.

2.2 Transactions

Transaction support is an important quality-of-service requirement in most enterprise business scenarios. Transferring money from one account to another in a banking application is a real-world example of where transaction support is important to ensure consistency. In the transfer, both the deduction of money from one account and the addition of money to the other account should occur in an "all-or-nothing" manner. The intermediate inconsistent states such as when only one account is modified should not be visible outside the transaction and if something goes wrong in the middle of the transfer, all accounts should be restored to their initial values before the transaction. This section provides a background on transactions and an overview of the different transaction types that the authors took into account when developing the comparison framework for RESTful transaction models in Section 3.2.

Gray defined the transaction concept with *atomicity*, *durability*, and *consistency* [8], and Haerder and Reuter coined the acronym *ACID*, adding *isolation* to the aforementioned three properties [9]. *Atomicity* ensures that transactions change the state of the application in an

atomic manner, *i.e.*, either the state changes performed by all operations are reflected or no changes are reflected (all or nothing). *Consistency* ensures that the result of a transaction is correct and complete, thus it produces consistent results by transforming a system in one consistent state into another consistent state. *Isolation* ensures that no concurrency anomalies will be present when transactions are run in parallel, i.e., the outcome of a set of parallel transactions is the same as if they were executed serially one after the other (*serializability*). Finally, *durability* guarantees the persistence of the successfully completed transactions, *i.e.*, all the changes made by a transaction are guaranteed to survive any software or hardware failures subsequent to the commit operation of the transaction.

However, it is not always possible to guarantee the strong consistency property of the ACID model because strong consistency may hinder other quality aspects of distributed data-sharing systems. According to the *CAP* theorem [10, 11], these systems can only exhibit, at most, two of the following three properties: *consistency*, *availability*, and *tolerance to network partitions*. Furthermore, even in the absence of network partitions, data-replication-based high-availability systems require a trade-off between *consistency* and *latency* as stated in the *PACELC* theorem [12]. To overcome these issues, other consistency models propose to make a compromise between *consistency* and *availability/latency* by relaxing consistency guarantees in order to cater for network partitioning, fault-tolerance, and high-availability (see *eventual consistency* [13] and *BASE* [14]).

Another orthogonal aspect to consider is the structure of the transactions. The simplest transactions are called *flat transactions* [4]. Flat transactions have only one layer of control, and they normally consist of a begin operation, a collection of business operations that have to be carried out as a single unit of work, and, finally, a commit or a rollback operation. Though flat transactions provide a good foundation to build transactional systems, they are too restrictive for some complex use cases. For example, if a transaction consists of a large number of bulk operations, it might not be efficient to rollback all the successful operations and restart from scratch because of a failure in a single operation at the end. Thus, transactions evolved providing more control over the flow of the transaction, *i.e.*, when and how the commit and rollback operations can performed.

Flat transactions with save points [4] provide an improvement to flat transactions by adding the ability to rollback to specific points of the history that were previously established instead of completely reversing all the effects of the transaction. Save points are useful in those cases where it is too expensive to rollback completely and to restart the transaction when one operation fails, and when it is more efficient to start from an intermediate state.

Chained transaction [4] are another variation to achieve a similar goal in which, instead of having volatile save points, the transaction is organized as a set of transaction segments chained one after the other. Once each segment is finished, the work will be committed, but the chained transaction will continue with the same context, *i.e.*, keeping all the resources such as locks on objects needed for future operations.

Nested transactions [15] organize transactions into hierarchies and define sub-transactions within a transaction that can be completed or rolled back individually. The advantage of nested transactions is that they allow the transaction to be divided, thus, providing more control over which parts to be committed or rolled-back. The commit of a sub-transaction will only be visible to the parent transaction; moreover, if the parent transaction rollbacks,

all its sub-transactions are also rolled back.

Open nested transactions [16] are a less structured version of multilevel transactions without a concrete hierarchy. A *compensating transaction* is a transaction that could generate the semantically reverse effect of a previous transaction and thus can cancel out the effects of the previous transaction. Compensating transactions are used when either no commit/rollback support is available or when transactions are long lived.

Multilevel or layered transactions [16] are a generalized version of nested transactions where an early commit of a sub-transaction (pre-commit) is allowed and, unlike nested transactions, pre-commits are globally visible. Thus, in multilevel transactions the isolation property is relaxed so that commits of the sub-transactions are visible to the other transactions. The atomicity property is achieved using compensating transactions rather than state-based undo.

A *saga* [17] is an interaction pattern for long-lived transactions in which the work has to be decomposed into several sub-transactions and their steps can interleave . Each sub-transaction provides a corresponding compensating transaction that will reverse the effects if the transaction has to be rolled back. Sagas are similar to nested transactions, but sagas restrict the transaction hierarchy to only two levels and do not provide full atomicity at the outer level.

Table 2. A summary of transaction types

Name	**Description**
Flat transactions (FT)	Atomic action units
FT with save points	Flat transactions that can rollback to intermediate points
Chained transactions	A sequence of transactions that share locks and other resources
Nested transactions	A hierarchy of transactions
Sagas	A sequence of flat transactions that can be interleaved with other transactions
Multi-level transactions	Nested transactions with pre-commit compensation
Open-nested transactions	Anarchic nested transactions
Distributed transactions	Transactions in a distributed environment
Workflow transactions	Transactions adapted to workflow specific requirements

Distributed transactions [18] are transactions that run in a distributed environment and have to communicate with different nodes of a distributed system. Distributed systems are defined as systems in which hardware or software components are located at networked computers and communicate and coordinate their actions only by passing messages; nevertheless, they provide services to the users appearing as a single entity [19]. Similar to nested transactions, distributed transactions are composed of several sub-transactions, each running on a different node. Each node generally has a resource manager that manages the resources in that node and decides whether it can guarantee the commit of a transaction or not. In distributed systems, one of the challenges is that the different nodes that are members of the system can form partitions in case of network failures; as a consequence, different nodes of the system can contain different values for the shared data.

Workflow transactions [20] incorporate workflow specific requirements to the previously-

described transaction models. As mentioned earlier, a transaction is defined through a sequence of operations. In workflow transactions, these operations are mapped to workflow activities. These individual activities can also be transactions, leading to a hierarchical structure. Unlike other types of transactions, workflow transactions commonly include human tasks in their flow. Real actions such as drilling a hole cannot be reversed; thus, the atomicity property has to be relaxed. Schuldt et al. provides a framework for ensuring atomicity and isolation in the context of transactional workflows [21].

A summary of the different transaction types are listed in Table 2.

3 Research Methodology

The first objective of this work is to identify and review the current RESTful transaction models found in the research literature as well as in the industry. This task will provide a good overview of the state of the art of RESTful transaction models. The second objective is to analyse and compare the identified RESTful transaction models according to their ability to fulfil common transactional scenarios. These tasks allow us to identify capabilities and good features of the current RESTful transaction models and also the gaps in the current state of the art. This section outlines the research methodology used to achieve these objectives.

A systematic literature review is the most appropriate technique to find all studies available that investigate a specific research topic or research area with a well-defined methodology in order to identify, evaluate, analyse, and interpret such studies in a way that is unbiased and repeatable to a certain degree [22]. Systematic literature reviews are repeatable if they follow a rigorous auditable well-documented methodology that allows to evaluate the study for its completeness and fairness. B. Kitchenham and S. Charters provide a set of guidelines for performing systematic literature reviews in Software Engineering [22], including how to plan, conduct, and report the results of the review; Section 3.1 describes the methodology we have followed, which is based on those guidelines.

No standard framework for comparing RESTful transaction models exists to date; therefore we have compared and evaluated the RESTful transaction models identified in the literature review according to the comparison framework presented in Section 3.2. This framework has been formulated based on the common transactional scenarios in traditional enterprise applications (adapted to RESTful systems) and on other characteristics generally used to evaluate transaction models.

3.1 Systematic Literature Review of RESTful transaction models

The first phase of the systematic literature review process involves planning the review. Planning entails three main activities: (a) *identifying the need for the review*, (b) *specifying the research questions*, and (c) *developing a review protocol* [22].

The systems built conforming to the REST architectural style are presently getting traction, in the industry, as a lightweight alternative for SOAP-based Web Services, as highlighted by the recent Forrester research report [3]. However, there are several enterprise Quality-of-Service requirements such as security, reliability, and transactions [1] that have to be integrated into RESTful designs so that those systems can be used in industrial settings. Discussions on REST-compliant transaction models have been held since the early 2000s when the REST architectural style was first introduced but, to the best of the authors' knowledge,

no study that summarizes these existing REST-compliant transaction models and compares them is available at present. Such study would provide the state-of-art of REST-compliant transaction models and would help to identify the current gaps in REST-compliant transaction model research. Furthermore, a systematic literature review on REST-compliant transaction models would provide the necessary background for making the right decisions in future REST-compliant transaction models by learning from the experiences of the previous models. These were the motivations behind our systematic literature review of REST-compliant transaction models.

The second step, specifying the research questions, is an important step in a systematic literature review because it provides the basis for the entire review process and drives (a) the selection of the primary studies concerned with the research question; (b) the type of data needed to be extracted from the selected primary studies to answer the research questions; and (c) how the data analysis and synthesis should be done so that the review report provides a comprehensive answer to the research questions .

In this systematic literature view, the main research question that was formulated and its related sub-questions are

- RQ1. What is the state-of-the-art of the current REST-compliant transaction models?
  - RQ1.1 To which level do the current RESTful transaction models conform to the REST constraints?
  - RQ1.2 What are the transactional guarantees provided by these models?
  - RQ1.3 How can common interaction patterns of RESTful systems be executed transactionally using the current transaction models?

A review protocol defines each of the steps in the systematic literature review process. A well-defined review protocol allows an external reviewer to evaluate the completeness and fairness of the study, and further it makes the study repeatable to some extent. It also helps to reduce the possibility of the researcher bias [22].

A review protocol generally consists of the following: (a) a search strategy, (b) a study selection criteria, (c) a quality assessment criteria, (d) a data extraction strategy, and (e) a data synthesis strategy. The following sections describe the review protocol developed for this systematic literature review.

Search strategy – The *search strategy* used in the review includes the key search terms, queries, and the data sources to be used. During the development of the search strategy, a set of key terms was identified and these terms were enriched with a list of synonyms, abbreviations, alternative spellings, and similar terms. Consultations with experts in the REST community and common keywords found in well-known papers on the REST architectural style were used as guidelines for identifying these key terms. Table 3 shows the key terms identified. These key terms were then combined using the Boolean operators *AND*s and *OR*s to generate the search strings.

In the next step, the selection of the data sources was done. Systematic literature reviews generally only consider research literature. However, during the pilot searches the authors discovered that a considerable number of discussions took place outside the research literature, in particular in public forums used by the REST developer community and in technical

Table 3. Key terms

Main term	**Related terms**
REST	Representational State Transfer, RESTful, REST-ful, REST-compliant, REST-compliance, RESTy
Transaction	transaction(s), transactional
Model	model(s), technique(s), protocol(s), design pattern(s)

literature such as developer guide books (see Table 4). Thus, the authors decided that it would be useful to include in the review public forums (where most of the REST related discussions have happened in the past) as well as the technical literature written by well-respected authors in the REST community. These two information sources will add value to the literature review by providing a big picture that covers the ideas originated from the REST developer community. Inclusion of those information sources also made it possible for the authors to verify which ideas that started in public forums were propagated to research literature with a comprehensive investigation and further developed to become robust models, and which ones are still not explored in depth. Nevertheless, the models found in research literature were given more weight in the data synthesis and analysis phases.

Table 4. Data Sources

Data source	**Description**
Research Literature	Peer-reviewed papers from journals, conferences, and workshops which present the models or techniques that were well-thought, developed and evaluated. These papers are presented in a more research-oriented point-of-view.
Technical Literature	Developer books published by the respected authors in the REST community. They provide methods and techniques in a more technical-oriented manner and include implementation guidelines.
Public forums	Mailing lists such as rest-discuss[a] blog posts and technical interviews from respected visionaries of the REST community. These public forums present methods and techniques that vary from initial thoughts to ideas that are verified though an implementation and evaluated according to different aspects.

For finding relevant research literature, the following databases and citation indexes were used: Google Scholar, IEEE Xplore, ACM Digital Library, ISI Web of Knowledge, and Springer Link. In addition, because of their relevance in the REST related topics, the following individual journals were also included: Journal of Web Engineering, Journal of Future Generation Computer Systems, ACM Transactions on the Web, and International Journal of Web Engineering and Technology.

Study selection criteria – The *study selection criteria* determines which primary studies will be included and excluded from the review. The study selection criteria of this review provides guidelines on identifying the primary studies that are most relevant to the research question, and on selecting the studies that provide information necessary to answer the research questions. Study selection criteria are devised using well-documented inclusion criteria and exclusion criteria [22]. The main inclusion criteria used in this review are that (a) the domain of the primary study should be systems that implement RESTful architectural styles, and (b) the main focus of the primary study should be on transactional models. The main exclusion criteria used are that (a) a primary study uses the term REST but the domain of the study is mostly SOAP-based Web Services, or (b) a primary study uses the term transaction(s) but does not provide details of a transaction model.

Quality assessment – The *quality assessment criteria* are used to evaluate the quality of

the primary studies selected and to further strengthen the study selection criteria. The quality assessment criteria also help to eliminate validity threats and make the study more objective. Different biases such as selection bias, performance bias, detection bias, and exclusion bias can be prevented using the quality assessment criteria; protection mechanisms can also be taken [22]. In this study, we have implemented a quality assessment criteria based on a set of questions to ensure that each of the selected primary studies is focused on the topic of RESTful transactions and provides some data to answer to research questions.

Data extraction strategy – The *data extraction strategy* provides a well-defined mechanism to extract the relevant data from the primary studies to answer the research questions and also to extract other metadata required for selecting the study selection and for executing the quality assessment criteria. Generally, data extraction forms are used for data extraction to ensure that all the necessary data are extracted and are available in a structured manner for further analysis and data synthesis. Additional metadata such as the name of the reviewer and additional notes are also recorded for traceability.

This review has one main question that is focused on identifying the current REST-compliant transaction models and three sub questions that are aimed to investigate the identified models in more detail. Thus the data extraction forms were designed by expanding each such question in more detail so that the extracted information could be used to answer the research questions.

Data synthesis strategy – The *data synthesis strategy* specifies how the results from the primary studies will be analysed and presented. Synthesis can be either quantitative or descriptive; for this review the synthesis was mainly descriptive because of the nature of the research questions. The information presented was chronologically ordered whenever it was relevant to make the evolution of RESTful transaction models visible. A more detailed analysis of the identified models was done with respect to the second objective of this work using the comparison framework for RESTful transaction models which is presented next.

3.2 Comparison framework for RESTful transaction models

The second objective of this work is to analyse and compare the RESTful transaction models identified in the systematic literature review. This section describes the comparison framework used to analyse and compare the different RESTful transaction models.

3.2.1 Dimensions of transaction models

As the first step of the comparison framework, we present a set of dimensions related to RESTful transaction models that can be used to cluster these models into related groups. Table 5 illustrates the main dimensions and the different alternatives in each of those dimensions.

We analysed REST-compliant transaction models along different dimensions. One of the dimensions is the transaction guarantees provided by the model. We can identify three main types of transactions based on the transactional guaranteed provided: ACID transactions, long-lived actions, and business transactions. On the one hand, ACID transactions, which are widely used in database transactions, are based on timeliness and trust, and they are suitable for short-lived transactions within a single application (in which the actions are trusted because they originate within a single application boundary). On the other hand, long-lived transactions involve actions that could take hours or days to finish and thus, do not fit well under ACID transactions. The same techniques that are used for ACID transactions

Table 5. Analysis dimensions

Dimension	**Alternatives**
Transaction type	Short-lived ACID transactions Long-lived transactions Business transactions
Transaction structure	Flat transactions Checkpoints and save points Chained transactions Sagas Nested transactions Multi-level transactions Open-nested transactions Distributed transactions Workflow-based transactions
Concurrency control mechanism	Pessimistic locking Optimistic concurrency control Timestamp ordering
How uncommitted state is handled	Provisional resources Headers Externally visible bookings
Distributed coordination	Two phase commit Paxos TCC

cannot be used in long-lived transactions because some blocking operations could have a negative effect on the throughput of the system and could affect its overall performance.

Business transactions are consistent changes of the state in a business that is driven by a well-defined business function that could be either short-lived or long-lived transactions. In business transactions, different ACID properties are required at a higher level considering the consistency of business operations rather than the consistency of individual read/write operations on data. For instance, atomicity can be viewed at different levels such as system-level atomicity, business interaction-level atomicity, or operational level atomicity [23]. In some situations such as reservation of tickets, the isolation property can be relaxed because of the nature of the reservation business; it is expected that the reduced availability due to an ongoing booking is visible to other buyers without making their state inconsistent.

Depending on the structure of the transaction, different types of transactions can be identified. Starting from simple flat transactions, which are linear (*i.e.*, lacking any structure) and support strict ACID properties, transactions can be structured into different ways to provide more flexibility as needed by complex scenarios such as transactions with checkpoints and save points, distributed and nested transactions, chained transactions, sagas, multi-level transactions, or workflow-based transactions (see Section 2.2). Most modern transaction models, such as web service and grid transaction models, support more than one of these structures.

Another way to look at transaction models is to analyse the concurrency control mechanisms used by them. Concurrency control mechanisms can be broadly classified into two main categories: pessimistic concurrency control and optimistic concurrency control. Pessimistic concurrency control mechanisms assume that conflicts will be common and, hence, transactions are synchronized early to achieve the serializability of parallel transactions. Due to the early synchronization, the pessimistic concurrency control methods do typically block parallel transactions in order to avoid conflicts. Thus, pessimistic concurrency controls are

not suitable when the actions take a long time. Optimistic concurrency control techniques assume that conflicts are rare and delay the synchronization until the end of the transaction. Optimistic concurrency control mechanisms handle conflicts when they happen rather than avoid them. However, handling conflicts adds overhead to the process making optimistic concurrency control not suitable when conflicts are common. Timestamp ordering is a concurrency control mechanism where each transaction is assigned a timestamp and conflicting operations are executed in strict order according to their timestamp. Timestamp ordering can be either optimistic or pessimistic. In order to execute timestamp ordering algorithms, each data item should maintain a read timestamp and a write timestamp; those timestamps contain the data about the timestamps of the transactions that read from or written to those data items and that will be used for accepting or rejecting incoming operations [4].

In the case of RESTful transactions, another dimension to consider is how transaction models provide access to the uncommitted state of a resource within an ongoing transaction yet to be committed. Because the REST architectural style puts the constraint on the interactions between the clients and the resources, *i.e.*, they have to be stateless; there are different approaches proposed by RESTful transaction models that allow clients to access these intermediate states while trying to conform to the stateless constraint. One approach is to represent these intermediate states as provisional resources. However, this approach leads to a change in the identifier (URI) of the provisional resource, and if the resource representation has relative URIs to other resources they will point to different resources after the change of the identifier. This identity switch can lead to confusion in the clients and, therefore, it should be well-defined how the identity switch is handled when the state is changed from the original resource to the provisional resource and then back to the original resource.

Another approach for managing the uncommitted state is to have a header to identify the transaction; in this case the representation returned by the resource will depend on the transaction identifier. This approach is criticized as a direct violation of the stateless REST constraint. The argument in favour of this approach is that resources are free to return different representations of the same resource depending on the client parameters. For example, depending on content negotiation servers send different media types of the same resource, or depending on the client authentication and the authorization level the server can send different levels of information. However, in both cases (the different media types or the partial representations), the representation returned represents the same state of the resource. By contrast, in the case of using a transaction identifier, different representations are returned represent different states. In the special case of the reservation business model, the uncommitted state can be externally visible as a booking, since isolation of those reservations is not required by the model.

Finally, another dimension that is important to consider when clustering REST-compliant transaction models is whether the transaction model supports decentralized and distributed transaction scenarios. In these scenarios, transaction models require a coordination protocol to handle the coordination between different components that are physically distributed across the network and managed by different authorities. Distributed transactions use consensus algorithms such as Two-Phase Commit (2PC) or Paxos for coordinating among themselves to agree on the atomic outcomes of a transaction. Try-Cancel/Confirm (TCC)[5] is a simplified form of consensus algorithms that can be implemented in conformance to the REST principles

and is suitable for reservation scenarios.

3.2.2 Common transaction scenarios

This section presents the common transaction scenarios that may appear in transactional RESTful systems. These scenarios, shown in Table 6, are grouped into three categories: (a) successful (happy path) scenarios where the transaction follows the regular path and completes successfully; (b) rollback scenarios where the transaction has to be rollbacked due to some reason; and (c) failure scenarios where either the client or the server fails during the execution of a transaction.

Several other orthogonal aspects, *i.e.*, whether the resources are centralized or decentralized/distributed, whether the transaction is short-lived or long-lived, and whether the operations involved in the transaction are simple updates or include other operations such as deletions and creations, are also considered in the successful scenarios. Nevertheless, the scenarios present in Table 6 are not exhaustive (*i.e.*, it does not include all the different combinations of these aspects) and relevant combinations are discussed when applicable in the analysis.

Table 6. Common transaction scenarios in RESTful systems

ID	**Description**
Successful scenarios	
Scenario I	A transaction that updates two resources
Scenario II	A transaction that involves update, creation, and deletion operations of resources
Scenario III	A long running transaction that updates two resources where the update operations take a long time (long-lived)
Scenario IV	A transaction that updates resources from different applications (decentralized and distributed)
Rollback scenarios	
Scenario V	A transaction with multiple updates where the server rejects an update because of a conflict with a parallel transaction
Scenario VI	A transaction with multiple updates where the client rollbacks due to some condition in its business logic
Failure scenarios	
Scenario VII	A transaction in which the client fails in the middle of the transaction
Scenario VIII	A transaction in which the server fails in the middle of the transaction
Scenario IX	A transaction where there are failures related to intermediaries such as communication failures and message losses due to problems in the infrastructure

When considering failure scenarios, fault tolerance, and recovery, we have to consider different fault modes in distributed systems. These modes include: byzantine or arbitrary failures, authentication-detectable byzantine failures, performance failures, omission failures, and fail-stop failures [24]. For this work, we do not consider the byzantine or arbitrary failures, in which the servers can send incorrect information to the clients; however, the rest of the failure modes are covered in the failure scenarios.

3.2.3 Analysis dimensions for RESTful transaction models

This section presents different dimensions that the comparison framework has taken into account when analysing the RESTful transaction models.

Scenario coverage – Table 6 shows common transaction scenarios adapted to RESTful systems. Each transaction model has been analysed to check whether the model can fulfil

each scenario. Running examples have been provided for the transaction scenarios that each transaction model supports. These examples not only provide the information on whether a given transaction model supports a given scenario or not but they also provide details on how the model works in those scenarios. This information acts as the input for the analysis of the models based on the dimensions described in the following sections.

Transactional guarantees – In this dimension, the different guarantees provided by each transaction model have been analysed. Traditionally, transactions should guarantee all the ACID properties i.e. *atomicity*, *consistency*, *isolation*, and *durability* (see Section 2.2). However, because strong consistency is a trade-off between other properties such as availability, latency, and partition tolerance, there are transaction models that relax the consistency guarantees such as *BASE* [14].

REST constraints compliance – In this dimension, we analyse whether the transaction models conform to the REST constraints in their protocol interactions. These constraints include: client-server architectural style, stateless client-server interactions, labelling of cacheability, uniform interface, layered system style, and optionally code-on-demand style [6]. The uniform interface is further defined using four interface constraints: (1) identification of resources, (2) manipulation of resources through representations (3) self-descriptive messages, and (4) hypermedia as the engine of application state (HATEOAS).

HTTP compliance and support – Most systems built according to the REST architectural style often use the HTTP protocol as the application protocol. All the RESTful transaction models that we have identified use the HTTP protocol. The goal of this dimension is to analyse the transaction model in terms of HTTP-related properties. The following concerns were analysed: (a) conformance to the HTTP semantics in protocol interaction (*i.e.*, properties of different HTTP operations such as safeness and idempotency are honoured); (b) only standard HTTP verbs and headers are used in the protocol interactions; and (c) the commonly used HTTP verbs (*i.e.*, GET, PUT, POST, DELETE, ...) are supported by the transaction model.

Protocol overhead – This dimension analyses the overhead added by the protocol interaction and compares with the original non-transactional interactions. This overhead plays a vital role in whether a transaction model will be widely accepted and used in practice. When a transaction model uses the HTTP protocol, the overhead could be measured in two different ways. On the one hand, if the protocol adds additional metadata to the request or response payload, the additional data overhead can be measured (*e.g.*, the increase of the size of the payload in bytes). On the other hand, if the transaction model introduces new HTTP round trips this will have a performance overhead with additional delays. The payload overhead depends on the actual payload of the scenario and it is hard to calculate a generic value for that. Nevertheless, the performance overhead can be calculated in a generic manner considering the number of additional round trips. Thus, for this analysis we have used the performance overhead.

Industrial adoption – In this dimension we look at whether there are implementations that demonstrate the feasibility of implementing the model and whether there are case studies of the model actually being used in real-world scenarios. The factors considered in this dimension go beyond just having the availability of an implementation, for instance, as a research prototype. Factors such as whether the implementation is evaluated for performance

and overhead and whether the implementation is adopted by the industry and used in a production setting are also considered.

Further, we analyse whether the transaction model can be incrementally adopted in current systems, *i.e.*, the introduction of the transaction model will not break any of the existing clients or servers which are not aware of the transaction model and if the clients and servers that use the model can gracefully fail or reject to interact when they identify that the server or the client on the other end is not capable of providing the support for the transaction model. This allows the clients and servers that support the transaction model and the ones that do not do so to co-exist with each other.

4 Literature Review Results

This section presents the results of the systematic literature review that was presented in Section 3. These results cover the literature published from 2000 till April 2014. The first part of the section describes the RESTful transaction models that were identified in the review, and the latter part categorizes them according to various different aspect showing the similarities and differences of the identified models.

4.1 RESTful transaction models

One of the objectives of this work was to identify current RESTful transaction models. This section presents the RESTful transaction models that were identified following the methodology defined in Section 3 and those results are grouped by the data sources: public forums, technical literature, and research publications.

4.1.1 Public forums

Since early 2000s discussions about RESTful transactions are not scarce in public forums on REST design principles and practical RESTful applications, such as the *rest-discuss*[b] mailing list. The conclusion of these initial discussions was that even though there are some theoretical mismatches between the REST architectural style and transactions, there is a clear need for transactions for various practical reasons [25]. The main focus of these discussions was on whether the concept of transaction could be adapted for REST-compliant designs and if so, how RESTful transaction models could be designed and implemented. This section presents the ideas that were discussed and analyses them. These ideas were contributed by different individuals in the mailing lists and further developed as a community effort. Later we present how these ideas were evolved to more concrete proposals. The main ideas that emerged in public forums include

- *Single resource pattern*, where the state changes that have to happen in a transactional manner are identified and such state is enclosed as a separate first class resource so that transactional state can be manipulated through a single stateless interaction. This pattern is also known as *single-web-resource* model or *abstract-transactional-resources*.
- *Overloaded POST pattern*, where several HTTP operations are encoded into the body of a single POST operation with the information about each HTTP operation, the URLs of the resources on which those operations will act, and the payload information for

[b] http://groups.yahoo.com/neo/groups/rest-discuss/info

each operation when applicable. In this pattern, the POST operation submits a batch of HTTP operations at once to be executed in the server in a transactional manner.

- *Transactions as resources pattern*, where a transaction is modelled as a resource itself such that the transaction resource can be used to manage the state of the transaction in a RESTful manner by performing different operations on the uniform interface of the transaction resource. In addition, the transaction resource is used as a log of actions that are related to a corresponding transaction.

- *Provisional-final pattern*, where a separate URL space (similar to private workspaces) or a set of provisional resources are used to represent the intermediate states of resources that are inside a transaction but not yet committed. This pattern isolates the intermediate states of each transaction by creating a private or a provisional copy of the resources involved, and the provisional resources are synchronized with the actual resources at the time of the commit.

- *Batch transactions pattern*, where transactions are modelled as a batch of operations transmitted in a single interaction that will be executed in an atomic manner in the server. Batch transaction pattern can be different from overloaded POST pattern because it can use other mechanisms such as mediators for processing the batch requests.

These ideas provided the base for the more concrete RESTful transaction models presented in the following sections.

Seairth Jacobs formulated a proposal for RESTful transactions in HTTP [26] that is based on the transactions as resources pattern and on the use of separate URL spaces associated with the transaction resource for representing provisional resources (provisional-final pattern). The transaction resource is serialized as an XML document that contains the logs of the transaction history (a series of "receipts" of the actions). Entity tags are used to refer to different versions of the resources during the transaction. The proposal included both optimistic updates (or delayed pessimistic) which do not involve locking resources and pessimistic updates which involve locking. However, this proposal was not developed further as a full specification nor was widely adopted by the industry.

Fielding, who disregarded the idea of having separate URL spaces for temporal resources because it leads to a change of the identity of the resource (URL) during the intermediate stages, proposed a different approach that he used in his Waka protocol [27]. In his proposal, instead of using a separate URL space for a transaction he used an HTTP header in each operation that contained the URI of the corresponding transaction to indicate that a given operation is part of a transaction. It is argued whether it makes the design stateful (as in the case of using cookies), but he defended his proposal by saying that the interactions are still stateless because the transaction identifier is a URL that can be accessed from anywhere so that this approach can scale to multiple servers without a problem and because the resources can have multiple representations based on this header, similarly to the case of content negotiation [c].

In addition, there were discussions on how the existing transactions models could be incorporated into RESTful architectures. The majority of the existing transaction models

[c] https://groups.yahoo.com/neo/groups/rest-discuss/conversations/messages/4165

came from the database research field and one of the main arguments against adapting them was that database transaction models are based on ACID properties which are too restrictive to be used on the Web. In contrast to databases, resources on the Web are mostly distributed and are managed by decentralized parties with no central authority that controls or owns everything. However, there were attempts to adapt existing models such as the UN/CEFACTs technology neutral transaction model and RosettaNet Partner Interface Processes (PIPs) to build a RESTful transaction model based on them.[d]

REST-*[e] is an initiative from the JBoss community that aimed at creating RESTful interfaces for common Web Service middleware services such as transactions, messaging, workflow, security, and management similar to the SOAP-based Web Services. This initiative triggered several discussions in the REST community and lead to two concrete proposals for RESTful transactions; one based on ACID transactions[f] and another one based on compensating transactions.[g] These proposals were implemented using the RESTEasy[h] REST framework and were included in the JBoss transaction middleware suites. As the next step, there was an attempt to formalize these proposals as specifications to be submitted to standardization bodies. Mark Little drafted the *REST-Atomic Transactions* specification to be submitted to OASIS but that did not get enough traction and it didn't get much support from the other parties to move forward as a standard.

In addition to the mailing list and wiki discussions, standardization bodies such as the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) published several technical specifications that are relevant to RESTful transactions. The W3C note on "*Editing the Web - Detecting the Lost Update Problem Using Unreserved Checkout*" [28] describes how entity tags can be used to detect the lost update problem and provides an optimistic concurrency control mechanism for the Web. Another W3C note "*Identifying Application State*" [29] summarizes some good practices on how URIs should be used to identify web application state and this note is aligned with the provisional-final pattern.

RFC4918 from the IETF on "*HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*" [30] introduces two new HTTP verbs LOCK and UNLOCK that can be used for concurrency control of Web resources. In addition to the support for exclusive and shared lock types, the WebDAV specification defines the depth of a lock that can be used to lock collection resources along with all its members. Even though the WebDAV protocol provides foundation for concurrency control, not all web servers support it and thus these HTTP operations might not be available in some web server implementations.

4.1.2 Technical literature

The technical literature, which mainly consists of technical books, describes design patterns, idioms, best practices, and guidelines on building RESTful applications and includes sections on how transactions can be implemented in RESTful services. These ideas and proposals align with the ones expressed in public forums. Table 7 lists the books that discuss the RESTful transaction models identified in the review.

[d] https://groups.yahoo.com/neo/groups/rest-discuss/conversations/messages/1399
[e] http://www.jboss.org/reststar
[f] https://community.jboss.org/wiki/TransactionalSupportForJAXRSBasedApplications
[g] https://community.jboss.org/wiki/CompensatingRESTfulTransactions
[h] http://resteasy.jboss.org/

The most predominant approach used in books is the transactions as resources pattern by encapsulating the transaction state in a transaction resource. For representing the uncommitted state, both the provisional-final pattern (*i.e.*, using a separate URL space under the transaction resource to represent provisional resources [31, 18]) and the use of a header to indicate that an interaction belongs to a transaction [32] are used in different approaches.

Jim Webber *et al.* illustrate an approach for RESTful transactions by combining the transactions as resources approach with two-phase commit [32]. However, they advocate to avoid cross organizational boundary transactions as much as possible with the argument of the additional complexity and the fact that they do not fit well with the original design of the Web. They describe an approach, *un-transactions*, in which HTTP status codes are used as coordination metadata and the consensus is built at each step. In contrast to other approaches on RESTful transactions, error recovery is done after every interaction rather than at the end.

Thomas Erl *et al.* proposed a different approach by introducing stateful complex methods with transaction semantics [33]. These methods intentionally breach the stateless REST constraints whilst accepting that it could reduce the scalability of the system. Two new HTTP methods are introduced: (a) the TRANS method allows carrying out the interactions necessary to perform two-phase commit, and (b) the PREP-PUT method, which is similar to the PUT method but does not persist the changes until they are committed. The use of new HTTP methods hinders the wide use of the transaction model because they are not widely supported by existing implementations.

Table 7 presents transaction models found in technical literature.

Table 7. RESTful transaction models in technical literature

Year	Title	Model description
2007	RESTful Web Services [31]	Transactions as resources
2009	Principles of transaction processing [18]	Transactions as resources
2010	RESTful Web Services Cookbook [34]	Single resource pattern Try-Cancel/Confirm pattern
2010	REST in Practice [32]	Transaction as resources with two phase commit Un-transactions
2011	SOA with REST [33]	Stateful complex method (e.g., TRANS, LOCK, PREP-PUT)
2013	RESTful Web APIs [35]	References to WebDAV methods (e.g., LOCK)

4.1.3 Research Literature

One of the earliest research papers to mention RESTful transactions is a paper from R. Khare and R. N. Taylor that describes a set of constraints on the configuration of components and connectors to extend the REST architectural style for distributed and decentralized systems [36]. These extensions include: Asynchronous REST (A+REST), Routed REST (R+REST), REST with Delegation (REST+D), and REST with Estimates (REST+E). Among those, only REST with Delegation and REST with Estimates are related to transaction processing. REST+D provides ACID transactions support by introducing a *MutexLock* component that acts as a proxy to the resource and provides the concurrency control to achieve serializability of all the updates with a distributed lock protocol such as Lamport's Bakery algorithm [37]. In addition, REST+E introduces a new consensus-free architectural style that can be used to achieve BASE properties instead of ACID properties. A prototype implementation known as

MOD_PUBSUB open source ARREST toolkit[i] was developed to evaluate to the feasibility of approach.

Luiz Alexandre *et al.* proposed an optimistic technique for transaction control using the REST architectural style [38]. In this technique, each transaction has a lifecycle that consists of three phases: a read phase, a validation phase, and a possible write phase. After all the resources are read, the changes are done locally and later made global in the write phase. The proposed approach mandates each resource to have a version number that is incremented with each update and informs about the version within its representation. If a write fails due to a validation failure (*i.e.*, the resource is updated by another party after the read and before the attempt to write this transaction), the transaction has to rollback and start over again.

This approach is only suitable when such conflicts are rare, because conflicts add a big overhead to the process. For rollback, the proposal uses the concept of compensation, which consists of making an action in order to logically reverse the effect of a previously executed action. For example, an action of updating a resource with a PUT could be compensated with a PUT with the previous value. However, compensation is harder for other actions such as DELETE. Another issue of optimistic techniques is that partial updates become visible outside the transaction and other parallel transactions can read these values thus becoming *dependent transactions* of this. For the compensation to happen without falling into adverse cascading aborts while rollbacking, the operations on resources should have the commutativity property such that they manipulate the resource with additions and subtransactions of data items (i.e., it is possible to calculate the delta between two versions) [39]. Another possible problematic situation is when two transactions repeatedly run into conflict because they update two resources in a reverse order causing both transactions to continuously rollback. This proposal is functionally very similar to the optimistic concurrency control of the HTTP protocol using etags, conditional requests (e.g., If-Match), and 409 conflict header codes [40].

Alexandros Marinos *et al.* proposed "*RETRO: A Consistent and Recoverable RESTful Transaction Model*" that uses the concepts of transactions as resources and temporary or provisional resources to represent intermediate resource states [41]. RETRO uses locking for concurrency control, and to follow the REST principle of resource identification, both locks and transactions are designed as resources. The model only supports resources with structured data representations such as XML. Each resource advertises a link to its lock collection and transaction collection; these collections can be used to create locks and transactions associated to a given resource.

Transaction and lock media types are defined with their semantics so the clients can interact with them without any out-of-band knowledge. RETRO supports both shared locks (SLOCK) and exclusive locks (XLOCK), and the lock conflicts are resolved using a lock compatibility table. Each lock has an associated expiry time granted based on the server defined rules. If a lock expires, all the locks of the same transaction expires to ensure the Two-Phase Locking guarantees.

Currently, RETRO only supports GET and PUT operations and they prove the correctness of the approach with formal proofs with the isolation theorems coming from the database community by mapping the GET and PUT operations to read and write database operations [42]. Apart from the general criticisms of the provisional-final resource approaches, one of

[i] http://mod-pubsub.org/

the main criticisms of the RETRO model is that it exposes too much metadata that is not essential to be publicly available and metadata is too scattered in many resources. Further, it introduces several sets of HTTP rounds trips to discover and manage locks and transactions that could possibly be optimized.

In 2010, Luiz Alexandre *et al.* proposed a Timestamp-based two-phase commit for RESTful services (TS2PC4RS) [43] as a successor of their optimistic technique; this model uses a non-locking concurrency control mechanism based on timestamp ordering and the two-phase commit protocol for coordinating and reaching consensus in distributed transactions. This approach is an extension to the Basic Timestamp (BTS) [44] approach for transactions by adapting it to RESTful services. Timestamp-based approaches have few advantages over lock-based approaches such as being deadlock free, because if an operation cannot be executed due to a timestamp conflict it is restarted with a new timestamp instead of being blocked. However, the additional overhead of the restarting operation is a drawback of this approach.

In BTS, each transaction is assigned a timestamp when it is initiated and each action (*e.g.*, read, write) of the transaction has the timestamp of the corresponding transaction. Each data item has three related metadata: (a) the largest timestamp of a write operation on the data item (WTM), (b) the largest timestamp of a read operation on the data item.(RTM), and (c) a list of buffered prewrites in the timestamp order (LPW).

Read/write operations from different transactions are accepted and executed in such a way that the final execution schedule can be rearranged to an execution schedule where each transaction is executed serially according to their timestamp ordering. This process ensures the serializability of timestamp-based transactions. Conflicts are resolved using two main rules: (a) a transaction's request to read an object is valid only if that object was last written by an earlier transaction, *i.e.*, if the transaction's timestamp is lower than the WTM it has to be restarted with a new larger timestamp to read the data item; and (b) a transaction's request to write an object is valid only if that object was last read and written by earlier transactions, *i.e.*, if the transaction's timestamp is lower than either WTM or RTM it has to be restarted [45].

BTS is integrated with two-phase commit to ensure the atomicity in distributed transactions. Generally, agents managing the resources in the two-phase commit protocol use locking-based mechanisms and have exclusively locks to the resources that are consumed by a transaction. Timestamp-based mechanisms integrate two-phase commit using a different approach called *prewrites*. Prewrites allow clients to update resources in a provisional manner, *i.e.*, the changes are buffered and not directly applied on the data items. Write operations are only applied on the data items when the transaction commits. Once a prewrite is accepted, it is guaranteed that that prewrite will be made permanent no matter when the commit operation arrives. However, depending on the other prewrites on the queue, the time when the prewrite will be written permanently varies. A transaction coordinator ensures that all the prewrites of a given transaction are either persisted or discarded by obtaining consensus from all agents involved in the transaction using the two-phase commit protocol [43].

In 2011, Guy Pardon and Cesare Pautasso proposed the Try-Cancel/Confirm (TCC) transaction model [5] based on their previous work on a lightweight transaction model for composite systems [46]. This transaction model is suitable for the reservation business model such as hotel or flight bookings for a single trip where all the tickets have to be reserved and booked

or none of them. One of the main use case of this model is composite RESTful services that atomically carry out a transaction with a set of distributed RESTful services. The composite service acts both as the workflow engine and the transaction coordinator. The main requirement of the model is to maintain the autonomy and loose coupling (i.e., they are unaware of the fact that they are part of a global transaction) [47]. To fulfil this requirement, the model avoids using a transaction context that is shipped around to the participants. Further, the model avoids using distributed locking by adopting the "try", "cancel", and "confirm" events mapped to business services. The model does not ensure the isolation property but it is argued that for services that fall under the reservation model this is not a problem. TCC model provides a recovery protocol in the case of different modes of failures such as failure of different participants and transaction coordinator, and failures in the different steps of the transaction. The recovery protocol makes use of heuristics, such as timeouts, to handle failures of participants.

The TCC model is based on a set of assumptions: (a) each RESTful service participating in a transaction is loosely-coupled with other RESTful services participating in the same transaction and with the transaction manager coordinating the transaction; (b) requests to the RESTful services can fail in a non-transient way; (c) atomicity should be guaranteed; and (d) RESTful services temporarily reserve resources for a given request and will not reserve them forever. In 2014, the authors proposed a *Transaction as a Service* [48] approach where the transaction coordinator is provided as a RESTful service on the web. This service allows clients to delegate the confirmation and recovery logic to a third-party service, eliminating them from the need to write boilerplate code to handle the protocol logic.

Finally, in 2012, Sebastian Kochman *et al.* proposed a transaction processing system for RESTful services following the concept of batch operations called AtomicREST [49]. The goal of their system is to provide a transaction system that is transparent to non-transactional clients and that does not require changes to existing RESTful services to provide transaction support. Their approach proposes an overlay network of mediators and proxy servers that provides the transaction support over existing services. For the services that do not support rollback or restoring to previous versions, they propose a simple (best-effort) compensation mechanism.

The main components of AtomicREST include: Server, Client, Mediator, and Proxy. Mediators and Proxies provide a middleware infrastructure for carrying out batch operations with minimum changes to the existing services. AtomicREST provides algorithms for each of these components to execute a RESTful transaction. The transaction processing may involve a single mediator in the simple case of a decentralized application or multiple mediators when the participating RESTful services are distributed and decentralized. Their paper on AtomicREST [49] provides both theoretical proofs of their algorithms to guarantee ACID properties and a performance evaluation where they show that the total overhead time per transaction is linearly dependent on the number of transaction requests.

Table 8 summarizes the transaction models found in the research literature.

5 Analysis of the models

This section analyses the RESTful transaction models found in the systematic literature review presented in Section 4 according to the comparison framework defined in Section 3.2.

Table 8. RESTful transaction models in research literature

Year	Model
2004	REST+D / REST+E (ARRESTED) [36]
2009	Optimistic technique for transactions using REST [38]
2009	A consistent and recoverable RESTful transaction model (RETRO) [41, 42, 50]
2010	Timestamp-based two phase commit protocol for RESTful services (TS2PC4RS) [43, 51, 52]
2011	Try-Cancel/Confirm pattern (TCC) [5, 48]
2012	Atomic REST batched transactions [49]

The analysis includes a clustering of the transaction models based on different dimensions and an evaluation of the models based on common transaction scenarios.

5.1 Clustering of the RESTful transaction models

By analysing the results of the review for RESTful transaction models, from the REST community forums to technical and research literature, we can identify several approaches for achieving RESTful transactions. Some techniques mix aspects from different approaches (e.g., ARRESTED, which introduces new HTTP methods and also uses delegation of proxies, or AtomicREST which uses batched transaction and also mediators). Table 9 summarizes the different transactions techniques identified.

Table 9. RESTful transaction approaches

Approach	Examples
Abstract composite resources	RESTful Web Services Cookbook
Batched transactions	Atomic REST , Overloaded POST
Transactions as resources	Principles of transaction processing RESTful Web Services, RETRO
Optimistic concurrency control	Optimistic technique for transactions using REST
Timestamp ordering based approaches	TS2PC4RS
Introductions of new methods	SOA with REST, ARRESTED
Delegation with proxies or mediators	Atomic REST, ARRESTED
Reservations with cancel/confirm	TCC

5.1.1 Transaction type

Table 10 shows the different transaction types (see Section 3.2.1) that are supported by the identified models. Transaction approaches such as abstract composite resources or batched transactions can be used in different types of transaction techniques by using both a synchronous mode (e.g., with 200 OK responses) in ACID transactions and an asynchronous mode (e.g., with 202 Accepted responses) in long-lived and business transactions. In the TCC model, an appropriate expiration time can be used to support long-lived transactions though this could possibly mean that resources are reserved for a longer time in case of cancellations.

5.1.2 Transaction structure

The most simple type of transactions is a flat transaction where the transaction consists of a begin operation, a set of business operations, and finally a commit, a rollback, or an abort operation. However, to be suitable for more complex business workflows, new transaction types such as flat transactions with save points, chained transactions, distributed transactions,

Table 10. Transaction type supported

Transaction type	**Examples**
Short-lived ACID transactions	RETRO, ARRESTED, Atomic REST, RESTful Web Services Cookbook, Overloaded POST, RESTful Web Services
Long-lived transactions	Optimistic technique for transactions using REST, TCC
Business transactions	TS2PC4RS, TCC

nested transactions, multi-level transactions, and open-nested transactions (see Section 2.2) were introduced. These different types of structured transactions add more flexibility to the transactions workflows, allow some efficiency boosts, and minimize resource blocking through partial commits and partial rollbacks.

Current RESTful transaction models are still in the early stages and are only concerned in flat transactions and distributed transactions. Distributed transactions in RESTful transactions will be discussed separately in a following section.

5.1.3 Representation of the uncommitted state (isolation)

A transaction takes the state of the system from one consistent state to another consistent state. Thus, before a transaction is started and when a transaction is committed, rollbacked, or aborted the system is in a consistent state. In between these two events, the system goes through a set of intermediate states that are only visible within the transaction and these intermediate states are isolated from the parties outside the transaction.

However, within the transaction (e.g., for the client(s) who is/are executing the transaction) there should be a way to access the intermediate uncommitted state. The aforementioned transaction approaches use different mechanisms for representing uncommitted state, which are summarized in Table 11.

Table 11. Uncommitted state representation

Approach	**Examples**
Provisional resources	RETRO, RESTful Web Services
HTTP headers	REST in Practice
Reservations	TCC

In the abstract composite resource approach (e.g., in RESTful Web Services cookbook) and batched transaction models (e.g., Atomic REST or Overloaded POST), the representation of uncommitted state is irrelevant because the operations are submitted in a single HTTP interaction and executed as a batch. In transaction models where the isolation property is not guaranteed such as the optimistic technique for transactions using REST, the uncommitted state is immediately visible as the resource state after each operation. In TS2PC4RS, the uncommitted state is kept as a queue of prewrites which are not directly accessible (i.e., they do not have a resource identifier on their own) or modified.

5.1.4 Concurrency control mechanisms

Transaction processing systems use concurrency control mechanisms to control the access to the resources and to achieve the necessary serializability while executing parallel access to resources. The RESTful transaction models identified in the previous section use several

concurrency control mechanisms, which are summarized in Table 12. The TCC model uses pessimistic reservation where the clients can make concurrent reservations until the resources are fully reserved and becomes blocking beyond that point, that is, before any other client can make further reservations, a previous reservation must be cancelled.

Table 12. Concurrency control mechanisms

Concurrency control mechanism	**Examples**
Pessimistic locking	RETRO, ARRESTED, Atomic REST
Optimistic concurrency control	Optimistic technique for transactions using REST
Timestamp ordering	TS2PC4RS
Other	TCC (Pessimistic reservation)

5.1.5 Distributed transactions and coordination

Some of the transaction models identified in the study support distributed transactions. Table 13 shows how the distributed transactions are handled by each of the transaction models identified.

Table 13. Distributed coordination

Coordination protocol	**Examples**
Two-phase commit	TS2PC4RS
Try-Cancel/Confirm	TCC

Atomic REST describes a proposal for distributed transaction using a communication algorithm among many mediators [49] and the RETRO model mentions distributed transactions as part of their future work [41]. The abstract composite resource approach mentioned in the REST Web Services Cookbook, the Overloaded POST and the transaction as a resource approaches described in the RESTful Web Services book do not support distributed transactions.

5.2 Scenario analysis

In this section, transaction models are analysed for the different scenarios identified in the comparison framework. Only the models found in research literature (see Table 8) are included because they capture the ideas from the other sources in a more developed and comprehensive manner. The ARRESTED model is not included because it does not provide details on how to execute the scenarios and the "Optimistic technique for transactions using REST" model is not included because the TS2PC4RS model is an evolution of the same model from the same authors. The following sections illustrate first each scenario in detail, and then discuss how the identified models behave.

5.2.1 Scenario descriptions

First we present the successful scenarios. Scenario I involves updating two resources that belong to a single application and Table 14 shows the HTTP operations involved.

Scenario II involves additional HTTP operations (POST and DELETE) to create a new resource and delete a resource within the transaction; Table 15 shows an example of the operations involved in the scenario. In this scenario, the "http://example.org/res/" resource acts as a collection that allows creating new resources through the POST operation.

Table 14. Scenario I - Successful update

OP	**URL**	**Response**
GET	http://example.org/res/A	200 OK
GET	http://example.org/res/B	200 OK
PUT	http://example.org/res/A	204 No Content
PUT	http://example.org/res/B	204 No Content

Table 15. Scenario II - Resource creation and deletion

OP	**URL**	**Response**
GET	http://example.org/res/	200 OK
POST	http://example.org/res/	201 CREATED Location: http://example.org/res/C
GET	http://example.org/res/D	200 OK
DELETE	http://example.org/res/D	204 No Content

Scenario III is similar to Scenario I but the update operations take a longer time and are handled in an asynchronous manner. Table 16 shows an example of the HTTP operations involved.

Table 16. Scenario III - Long running operations

OP	**URL**	**Response**
GET	http://example.org/res/A	200 OK
GET	http://example.org/res/B	200 OK
PUT	http://example.org/res/A	202 Accepted
PUT	http://example.org/res/B	202 Accepted

Scenario IV is similar to Scenario I but the two resources belong to two different applications. Thus, resource management is decentralized and could span across organization boundaries. Table 17 shows an example of the HTTP operations involved where resources come from two applications hosted in "example.org" and "remote.example.org".

Table 17. Scenario IV - Distributed and decentralized resources

OP	**URL**	**Response**
GET	http://example.org/res/A	200 OK
GET	http://remote.example.org/res/B	200 OK
PUT	http://example.org/res/A	204 No Content
PUT	http://remote.example.org/res/B	204 No Content

Scenario V and Scenario VI are rollback scenarios. A possible sequence of operations for Scenario V is shown in Table 18 where the server rejects the update in the fourth operation because that update conflicts with an update for another client between the time of the first and forth operations.

Table 18. Scenario V - Rollback due to a conflict

OP	**URL**	**Response**
GET	http://example.org/res/A	200 OK
GET	http://example.org/res/B	200 OK
PUT	http://example.org/res/A	204 No Content
PUT	http://example.org/res/B	409 Conflict

Scenario VI is similar to Scenario V but instead of a server-rejected update, the client wants to voluntarily rollback the third operation due to its business logic. Thus, it is a client-initiated rollback instead of a server-forced rollback.

Scenarios VII, VIII, and IX are failure scenarios. In Scenario VII, the client fails in the middle of a transaction, for instance, after the third operation in Table 14. These failures have implications because, on the one hand the system is left in an inconsistent state and, on the other hand, these failures could lead to unreleased locks and make the system block resources more than necessary. In Scenario VIII, the server fails in the middle of a transaction, for instance, a "500 Internal Server Error" or non-responding sever. Scenario IX is about communication losses and message losses. For instance, in the third or fourth operation of any of these scenarios, the request or the response message could get lost due to an unreliable network communication.

5.2.2 RETRO model

In this section, we analyse the behaviour of the RETRO model. The implementation of Scenario I would require the HTTP operations shown in Table 19. The RETRO model works only with resources with an XML representation and the resource representation should advertise link relations to a transaction collection and a lock collection as shown in Figure 1; those collections are to be used to create transactions and to lock resources.

Table 19. RETRO Model - Scenario I

N	OP	URL	Response
1	GET	http://example.org/res/A	200 OK
2	POST	http://example.org/trans/	201 CREATED Location: http://example.org/trans/1
3	POST	http://example.org/res/A/locks	201 CREATED Location: http://example.org/res/A/locks/1
4	GET	http://example.org/res/A/locks/1	200 OK
5	GET	http://example.org/res/A	200 OK
6	PUT	http://example.org/res/A/1	201 CREATED
7	GET	http://example.org/res/B	200 OK
8	POST	http://example.org/res/B/locks	201 CREATED Location: http://example.org/res/B/locks/1
9	GET	http://example.org/res/B/locks/1	200 OK
10	GET	http://example.org/res/B	200 OK
11	PUT	http://example.org/res/B/1	201 CREATED
12	DELETE	http://example.org/trans/1	200 OK

Figure 1 - Transaction collection and lock collection link relations

```
<lockable>
  <link rel='lockcollection' href='http://example.org/res/A/locks/' />
  <link rel='transactionCollection href='http://example.org/trans/' />
</lockable>
```

After discovering the transaction collection link (Step 1), the client creates a transaction by POSTing new transaction to the transaction collection (Step2). The transaction resource semantics is defined with a specific media type (application/vnd.retro-transaction+xml); the transaction resource contains information about the associated transaction collection, lock collection, and the owner of the transaction.

Once a transaction is created, the next step is to lock the resource by POSTing a lock to the lock collection (Step 3). The created lock resource has to be retrieved to find which is the URL of the conditional resource representation associated with the locked resource (Step

4). The resource has to be retrieved again to ensure that the read operation happened within the transaction after it was locked (Step 5). Then the resource is updated by PUTing the updated resource representation to the conditional resource representation (Step 6).

For updating the second resource, resource B, similar steps have to be followed (Steps 7-11). Finally, the transaction can be committed by deleting the transaction resource (Step 12). To sum up, the RETRO model requires 12 steps (HTTP round-trips) to execute Scenario I in a transactional manner.

The RETRO model only supports the GET and PUT operations; thus Scenario II, which involves the POST and DELETE operations, is not supported. RETRO is limited to the GET and PUT operations because they can be directly mapped to the read and write operations allowing the authors to use the existing theorems from the database world to prove the correctness of the model. The RETRO model is not a good fit for long-running transactions in Scenario III because it uses pessimistic locks which block resources. Besides, it does not have support for the distributed and decentralized transaction in Scenario IV.

Server-initiated rollbacks in Scenario V are handled using locks. If a transaction already holds a lock for a resource, a second transaction cannot obtain a conflicting lock for the same resource. Such request (e.g., Step 8 in Table 19) will lead to a '403 Forbidden' response. The client can retry the transaction after a delay or rollback by deleting the lock collection. Similarly, a client-initiated abortion of a transaction in Scenario VI is done by deleting the transaction lock collection that will release all the locks and void any changes done. The RETRO model does not describe how to handle failure scenarios (Scenarios VII-IX).

In summary, the RETRO model closely follows the REST principles, allows metadata discovery using links without eliminating the need for out-of-band communication, and properly uses media types. However, the lack of support for HTTP operations such as POST and DELETE and the lack of distributed transactions are two major drawbacks of the RETRO model.

5.2.3 TS2PC4RS Model

This section presents how the timestamp-based two phase commit protocol for RESTful services (TS2PC4RS) handles the same scenarios. In the TS2PC4RS model, each transaction is assigned a timestamp based on when it was started. Table 20 shows the HTTP operations involved in Scenario I. It assumes the transaction identifier is '10c1' and according to the TS2PC4RS model, each resource is postfixed with it.

Table 20. TS2PC4RS - Scenario I

N	**OP**	**URL**	**Request Content**	**Response**
1	GET	http://example.org/res/A/10c1	-	200 OK
2	GET	http://example.org/res/B/10c1	-	200 OK
3	PUT	http://example.org/res/A/10c1	Updated A	200 OK
4	PUT	http://example.org/res/B/10c1	Updated B	200 OK
5	PUT	http://example.org/res/A/10c1	commit = true	200 OK
6	PUT	http://example.org/res/B/10c1	commit = true	200 OK

In the TS2PC4RS model, and as shown in Table 21, each resource maintains (a) WTM, the largest timestamp of a write operation; (b) RTM, the largest timestamp of a read operation; and (c) LPW, a list of buffered prewrites in timestamp order. When a transaction attempts

to execute an operation, an algorithm is used to decide whether the transaction satisfies the conditions that are required to simulate a serial schedule based on their timestamp order. Table 21 shows how values of RTM, WTM, and LPW are changed during each operation and what is the condition that is being checked to decide whether the operation is allowed or not.

Table 21. State changes of each operation of transaction 10c1

N	OP	Res	Before			Condition	After		
			RTM	WTM	LPW		RTM	WTM	LPW
1	GET	A	(3, c2)	(6, c2)	-	TS >WTM	(10, c1)	(6, c2)	-
2	GET	B	(7, c2)	(8, c2)	-	TS >WTM	(10, c1)	(8, c2)	-
3	PUT	A	(10, c1)	(6, c2)	-	TS >RTM TS >WTM	(10, c1)	(6, c2)	Ac1
4	PUT	B	(10, c1)	(8, c2)	-	TS >RTM TS >WTM	(10, c1)	(6, c2)	Bc1
5	PUT	A	(10, c1)	(6, c2)	Ac1	true	(10, c1)	(10, c1)	-
6	PUT	B	(10, c1)	(8, c2)	Bc1	true	(10, c1)	(10, c1)	-

The resource read operations (Steps 1 and 2) are only allowed if the timestamp of the transaction is greater than the WTM value. In addition, if the timestamp is greater than the current RTM, the RTM is updated to the timestamp of the current transaction (see Steps 1, 2 in Table 21). For instance in Step 1, we assume that the resource A was last read (RTM) by a transaction with an identifier of '3c2' and last written (WTM) by a transaction '6c2'. Thus, this operation is allowed as the identifier of the current transaction i.e. '10c1' is bigger than the WTM. After the operation, the RTM is changed to '10c1'.

Once a successful read of the resource is done, it can be updated by sending a prewrite to the resource using the PUT operation (Steps 4 and 5). These operations will be successful only if the timestamp is greater than both RTM and WTM values. If successful, a prewrite is added to the queue (see Table 21). If the timestamp is lesser than the RTM (i.e., a transaction with a greater timestamp has seen the current resource state) or lesser than the WTM (i.e., a transaction with a greater timestamp has written the current resource state), the operation is rejected. If all prewrites are successful, the client can send a commit message to all the resources in the transaction. At this stage, prewrites of the given transaction will be written to the resource persistent state.

Scenario II requires support for the POST and DELETE operations and the TS2PC4RS model does not support those operations. The TS2PC4RS model can be used in Scenario III which involves long-lived transactions because it is an optimistic non-blocking model. Distributed transactions in Scenario IV are supported using inbuilt two-phase commit protocol. In the TS2PC4RS model, the client acts as the two-phase commit coordinator and coordinates whether prewrites have to be persisted or discarded.

In the TS2PC4RS model, the server decides to accept or not prewrites based on the timestamp of the transaction. In Scenario V, if the RTM of the resource is greater than the timestamp of the transaction the server rejects the prewrite by sending a "409 Conflict" response. In Scenario VI, the client can abort the transaction any time by sending abort messages instead of commit messages and the corresponding prewrites will be discarded. The TS2PC4RS model does not describe the failure scenarios in detail.

In summary, the TS2PC4RS model provides an optimistic timestamp-based mechanism for RESTful transactions. The model fails to adhere to some REST constraints such as state-

lessness and does not utilize standard mechanisms such as link relations or media types. The TS2PC4RS model supports distributed transactions. Because it is an optimistic approach, this model is more suitable when conflicts are rarely expected.

5.2.4 Try-Cancel/Confirm Model

This section presents how the Try-Cancel/Confirm (TCC) model handles each scenario. The TCC model is suitable for reservation use cases where a client wants to perform multiple purchases (e.g., a travel booking example where someone wants to book a flight and a hotel or two flight segments). In these scenarios atomicity is the main requirement because a partial reservation is not useful for the user.

The scenarios explained in Section 5.2 have to be adapted to a reservation scenario in order to analyse the TCC model. Thus, the two resource updates of Scenario I are replaced with a flight and a hotel reservation and the HTTP operations involved are shown in Table 22.

Table 22. TCC - Scenario I

N	**OP**	**URL**	**Response**
1	GET	http://flight.example.org/flight/EX1	200 OK
2	GET	http://hotel.example.org/madrid/	200 OK
3	POST	http://flights.example.org/booking/	302 OK Location: http://flights.example.org/booking/1
4	POST	http://hotel.example.org/booking/	302 OK Location: http://hotel.example.org/booking/1
5	PUT	http://flight.example.org/booking/1	204 No Content
6	PUT	http://hotel.example.org/booking/2	204 No Content

A client first retrieves the flight/hotel resource where it finds the link for the reservation service and other information needed for a reservation, such as flight times, cost, or room type. Then it creates reservations by POSTing the required information to the respective services (Steps 3 and 4). In the response, the server provides both a *participantLink* object with a *tcc* link relation that can be used to cancel or confirm the reservation and an expiration time for the reservation as shown in Figure 2. The client uses the *tcc* link to confirm by PUTing a confirmation message (Step 5 and 6) or to cancel by DELETEing the reservation. If a reservation is not confirmed before it expires, the cancellation is done automatically.

Figure 2 - The 'tcc' Participant Link object

```
{ "participantLink": {
    "uri":"http://flights.example.org/booking/1",
    "expires":"2014-06-31T00:00:00.000+01:00",
    "rel":"tcc"
    }
}
```

Scenario II is irrelevant to the TCC model as the transaction semantics of the TCC model is defined in the business logic level and not at the resource management level. The reservation action is mapped to a POST operation because it creates the participant link resource. Thus, in the TCC model PUT and DELETE on service resources (not protocol resources such as participant reservation link) do not make much sense.

Because the TCC model is not based on locks and is non-blocking, long-running transactions in Scenario III can be implemented. In general, the reservation model includes long-running transactions because reservations can involve human actions that might take a long time to complete. The timeout mechanisms in the TCC model can be adjusted based on the expected duration of the transaction. Because RESTful services in the TCC model are loosely coupled, the protocol works exactly the same way with distributed and decentralized resources in Scenario IV. Whether the resources are managed by a single centralized application or by several decentralized applications is transparent to the TCC model.

In the TCC model the server can indicate the client that a resource is not available for reservation (similar to Scenario V) in two possible ways. When the client checks the availability of a resource using a GET request (Step 1), the server can respond with a 204 response code without a body to let the client know that no resources are available for reservation. However, if the resource was available when the client first checked but later becomes unavailable (Step 3) the server will reject the POST request. In Scenario VI, the clients can cancel the reservations any time during the transaction by DELETEing the reservations.

The TCC model describes the different failure models and how they are handled [5]. The client failures are handled using the timeouts. In the case of a server and communication failures that happen before confirming any of the reservations, the client can cancel the rest of the reservations. The TCC model only fails to guarantee atomicity when a client has confirmed one participant and finds out the second one is timed out.

5.2.5 The Atomic REST model

The Atomic REST model provides a middleware layer that encapsulates the transaction processing by providing a proxy for RESTful services and an API for clients that take part in transactions. The HTTP operations for the Scenario I are shown in Table 23.

Table 23. Atomic REST - Scenario I

N	OP	URL	Response
1	GET	http://example.org/res/A	200 OK
2	GET	http://example.org/res/B	200 OK
3	POST	http://example.org/mediator	200 OK

In the Atomic REST model, transactions are performed as batch operations, *i.e.*, the clients set all the operations as a batch and execute them in one HTTP interaction. Because Atomic REST is a middleware-oriented approach, the client can use a code snippet similar to the one shown in Figure 3 to execute the transaction which then will be converted to the message shown in Figure 4 and sent to the mediator.

Figure 3 - The Atomic REST Client API

```
soa.atomicrest.client.TransactionBulk transaction;
soa.atomicrest.client.Collection<Response> resp;

transaction.add(new Request("PUT", accessUrl1).content("5"));
transaction.add(new Request("PUT", accessUrl2).content("10"));
resp = transaction.run(true);
transaction.clear();
```

Figure 4 - The message to the mediator

```
<transactionElement>
 <RequestElement method="PUT" uri="http://example.org/resources/A">
     <headers><header key="Content-Type">text/plain</header></headers>
     <content mediaType="text/plain">5</content>
  </RequestElement>
  <RequestElement method="PUT" uri="http://example.org/resources/B">
     <headers><header key="Content-Type">text/plain</header></headers>
     <content mediaType="text/plain">10</content>
  </RequestElement>
</transactionElement>
```

In the Atomic REST model, Scenario II is similar to Scenario I but with different HTTP methods as parameters. The Atomic REST model does not define special mechanisms for long running transactions. However, because the transaction processing logic is mostly done in mediators and proxies, the middleware can be adapted to long running transactions of Scenario III.

A distributed transaction mechanism is proposed using multiple mediator communications [49] for Scenario IV. The mediator receiving the initial request from a client becomes the leader and broadcasts the transaction to other mediators in a decentralized system, and each mediator executes a part of the transaction as a subtransaction. The leader then collects all the results and returns them to the client. However, this algorithm is not still implemented in their system.

Scenarios V and VI, which involve rollbacks, are not applicable to batched transaction models where the operations are submitted as a batch. The rollback of individual operations is handled transparently to the client. The Atomic REST model does not define a recovery model for failure scenarios and does not describe how failures are handled.

6 Discussion

In this section, we analyse the models based on the dimensions that are presented in the comparison framework for RESTful transaction models defined in Section 3.2. The following sections include analyses on scenario coverage, transaction guarantees, REST constraint compliance, protocol compliance, protocol overhead, and industrial adoption of each transaction model.

6.1 Scenario coverage

Table 24 summarizes the scenario analysis described in Section 5.2. If a transaction model successfully handles a scenario it is marked with a "✓" and if it fails to handle or does not describe how to handle the scenario it is marked with a "X". If a scenario is not applicable to a certain model, it is marked as "N/A". Though all models handle the basic scenario of updating multiple resources, some advanced scenarios are not handled by several models.

Scenario II is not applicable to the TCC model because it operates on a business level instead of on a resource management level. Scenarios V and VI, which involve rollbacks, are not applicable to the Atomic REST model because it is a batched transaction model.

Only the TCC model defines a recovery model for the failure scenarios and only the

Table 24. Scenario coverage by transaction model

Scenario	RETRO	TS2PC4RS	TCC	Atomic REST
Scenario I	✓	✓	✓	✓
Scenario II	X	X	N/A	✓
Scenario III	X	✓	✓	✓
Scenario IV	X	✓	✓	X
Scenario V	✓	✓	✓	N/A
Scenario VI	✓	✓	✓	N/A
Scenario VII	X	X	✓	X
Scenario VIII	X	X	✓	X
Scenario IX	X	X	✓	X

TS2PC4RS and the TCC models describe and implements distributed transactions. The RETRO model has distributed transactions in their future work plan and Atomic REST has devised an algorithm, but their implementation does not support distributed transactions.

6.2 Transaction guarantees

In this section, we analyse the RESTful transaction models based on the ACID transaction guarantees. Table 25 shows a summary of the transaction guarantees provided by each model.

Table 25. ACID transaction guarantees by transaction model

Property	RETRO	TS2PC4RS	TCC	Atomic REST
Atomicity	✓	✓	✓	✓
Consistency	✓	✓	✓	✓
Isolation	✓	X	X	✓
Durability	✓	✓	✓	✓

One point to note in Table 25 is that not all properties are equally dependent on the transaction model. While the atomicity and isolation properties are highly dependent on the transaction model, the consistency and durability properties are mostly guaranteed by the application logic. Each application generally has its validation rules to verify that each operation results in a valid system state and this validation along with the atomicity property ensures that a transaction always moves the system from a consistent state to another consistent state. The durability property is guaranteed by the persistence layer of the applications.

Optimistic approaches such as TS2PC4RS do not guarantee isolation because the uncommitted intermediate states are visible outside the transaction and because isolation is not considered by design in reservation models such as TCC.

6.3 REST constraint compliance

In this section, we present the analysis of the RESTful transaction models with respect to the REST constraints. Table 26 analyses the transactional models according the REST constraints, except the Uniform Interface, whose analysis is broken down by sub-constraints in Table 27 due to its particular high-relevance in RESTful systems.

The RETRO and TS2PC4RS models are not stateless because part of the session state is maintained in the server. The Atomic REST model breaks the uniform interface constraint by overloading the POST operation. The TS2PC4RS and Atomic REST models do not use specific media types to send self-descriptive messages, nor do they utilize hypermedia controls

Table 26. REST constraints by transaction model

Constraint	**RETRO**	**TS2PC4RS**	**TCC**	**Atomic REST**
Client-server architectural style	✓	✓	✓	✓
Stateless client-server interactions	X	X	✓	✓
Cacheability	✓	✓	✓	✓
Layered system style	✓	✓	✓	✓

Table 27. Uniform Interface constraints by transaction model)

Constraint	**RETRO**	**TS2PC4RS**	**TCC**	**Atomic REST**
Identification of resources	✓	✓	✓	✓
Uniform interface	✓	✓	✓	X
Manipulation of resources through representations	✓	✓	✓	✓
Self-descriptive messages	✓	X	✓	X
Hypermedia as the engine of application state	✓	X	✓	X

to drive the application state.

6.4 HTTP compliance and support

Even though the REST architectural style is not dependent on the HTTP protocol, HTTP is the de-facto protocol used in RESTful applications. Thus, in this section we analyse the compliance with the HTTP protocol when used within a transaction model. The TS2PC4RS model violates the safe property of the GET operation, *i.e.*, the GET operation in the TS2PC4RS model has side effects.

Then we look at the support for common HTTP verbs. The RETRO, TS2PC4RS, and TCC models support the GET and PUT operations but do not support the POST and DELETE operations. Because TCC is a business level protocol, it does not require the POST and DELETE operations in business operations and it uses DELETE for cancelling a participant. The Atomic REST model supports all four aforementioned operations.

6.5 Protocol overhead

In this section, we analyse the overhead added by each transactional model. As discussed in the comparison framework for RESTful transaction models in Section 3.2, the overhead is measured using HTTP round trips. The first column of Table 28 presents the number of round trips required without the transaction support and the next columns show the round trips required by each model for each scenario when supported.

Table 28. Performance overhead by transaction model

Scenario	**Without TS**	**RETRO**	**TS2PC4RS**	**TCC**	**Atomic REST**
Scenario I	4	12	6	6	3
Scenario II	4	N/S	N/S	N/S	3
Scenario III	4	N/S	6	6	3
Scenario IV	4	N/S	6	6	N/S
Scenario V	5	14	5	4	1
Scenario VI	5	14	6	4	N/S

For Scenario V, we considered two updates where the server rejects the second update and for Scenario VII the client wants to rollback when the first update is finished and before the second update.

The RETRO model has the highest overhead because transaction and lock creations involve several steps and because the metadata are scattered in many resources. The Atomic REST model, due to its batch operations, has the least overhead, furthermore, it is even more efficient than the original scenario implementation in terms of round-trips. The TS2PC4RS and TCC models have a similar moderate overhead.

6.6 Industrial adoption

As for industrial adoption, the TCC model, which is focused on the reservation model, has more success compared to other models. The TCC model is integrated as part of the Atomikos ExtremeTransactions[j] transaction management product developed by Atomikos. In addition, it is used in the Atomic Browser [53].

The RETRO model is developed under the OPAALS research project in the context of generative Web information systems.[k] The Atomic REST model is developed under the IT-SOA project, and an implementation is available on its website.[l] However, authors didn't find industrial usages of these models outside their projects.

In order to analyse the compatibility with the existing clients and servers and to find out whether or not the protocol can be incrementally adopted, we used Scenarios I and II but with a non-transactional client (that is, a client unaware of the transaction protocol) interacting with a resource of an ongoing transaction by using a safe operation such as GET and an unsafe operation such as PUT or DELETE. In the RETRO model, GET will return the last committed value, and the unsafe operations will return 405 Method Not Allowed status code [41]. This behaviour is compatible with existing clients because the error flow is handled using standard HTTP protocol status codes. The TS2PC4RS model does not define how non-transactional client requests (*i.e.*, a request without a timestamp based transaction identifier) are handled during an ongoing transaction. In the TCC model, as it is a business level protocol, the client should be aware of the protocol to create, cancel, or confirm the reservations, thus, incremental adoption is not possible. The aforementioned scenarios of intermediate interactions does not apply to the Atomic REST model because it is a batched transaction model.

7 Challenges

Based on the analysis of the existing transaction models discussed in this paper, we have identified the following challenges for the current RESTful transaction models [54].

7.1 Decentralized and distributed services

Transactions that involve resources managed by multiple authorities is one of the main challenges in current RESTful transaction models. The main problem of decentralized authorities is the need for coordination and agreement with regard to the final outcome of a transaction whilst ensuring its atomicity, an issue that requires complex failure modes and recovery mechanisms [19]. This is a common problem in distributed computing that is typically solved using a consensus protocol, *e.g.*, the two-phase commit (usually the XA protocol). However, the majority of the RESTful transaction models (except for two models) do not cover this

[j] http://www.atomikos.com/Main/ExtremeTransactions
[k] http://www.opaals-oks.eu/
[l] http://www.it-soa.eu/en/resp/atomicrest/

scenario and the challenge is to design a stateful consensus protocol without violating the REST constraints.

Distributed systems in which the ordering and timing of events is relevant[m] require the synchronization of *logical* clocks of different nodes [55]. Currently, mechanisms such as Lamport timestamps and vector clocks are used for ordering events in distributed systems [55]. How these approaches can be applied to REST services for ordering the actions on different resources and how timestamps can be used consistently still remain a challenge to be solved.

7.2 Contradiction of statelessness and isolation

The *statelessness* REST constraint states that servers should be stateless and should not maintain any conversation state with the client (client-stateless-server) [6]. However, the *isolation* ACID property states that any intermediate change of a transaction should not be visible to ongoing parallel transactions. This requires servers to maintain intermediate states for actions that are not committed by maintaining a session state for a transaction. Thus, these two properties are in conflict.

Current isolation-preserving REST transaction models solve this problem representing these session states as a set of temporary resources that have their own identifiers (URLs). Despite this approach aligns with W3C best practices,[n] it is arguably a REST anti-pattern, as those temporary resources do not represent resource state but application (or session) state. Furthermore, this approach introduces a new challenge: link transparency. When working with temporary resources, it is necessary to distinguish links that point to temporary resources from those that point to original resources, so that when the transaction is committed, all the links of original representations point to original resources.

An alternative approach to solve this issue could be the usage of a mechanism similar to that proposed by the Memento framework[o] for providing access to representations of different resource states using the same identifier (URL). However, this approach directly violates the stateless REST constraint.

7.3 Issues related to concurrency control

Locking has been the prominent solution for achieving isolation in transactions in the database field [4] and most RESTful transaction models have followed the same path. However, there are several issues that need to be taken care of when using this technique, in particular: availability, deadlock prevention, and fairness guarantees.

Availability is a fundamental aspect of distributed applications; therefore, transaction models should minimize the negative effects of locks on the availability of resources. This issue is deepened by the fact that operations in REST applications take longer due to transport overheads (HTTP).

Deadlocks and resource starvation are common problems when locks are not used consistently or when fairness is not guaranteed. These become important specially when the acquisition and release of locks is managed by different clients. One corner case would be a misbehaved client (or a client with a defect) not releasing the locks after it has finished with a transaction.

[m] Those in which agents residing in different nodes of the system have to perform actions in a particular order.
[n] `http://www.w3.org/2001/tag/doc/IdentifyingApplicationState\#UseURIsforStates`
[o] `http://www.ietf.org/rfc/rfc7089.txt`

Current approaches use two-phase locking with a growing phase and a shrinking phase to prevent deadlocks; they also use timeouts to get some degree of fairness (lock auto-release after timeout). However, the enforcement of two-phase locking and achieving fairness remains a challenge for the RESTful transaction models.

Another alternative is to use the optimistic concurrency control mechanisms provided by HTTP using conditional updates with ETags. However, this approach does not guarantee isolation as intermediate states of the resources become visible outside the transaction.

7.4 Resource granularity and composition

REST allows resources to be at different granularity levels. Thinking in a hierarchical model, an application could simultaneously provide both a high-level view of an entity via a coarse-grained resource and a detailed view using a fine-grained resource. Also, *collection* resources found in specifications such as AtomPub[p], Hydra [56], or the Linked Data Platform[q] are special cases of resource composition.

These particular cases lead to problematic situations when locks are used with these resources, i.e., locking a specific resource might not prevent the information carried in that resource from being read or updated because this information is not exclusively bounded to such resource. Thus, resource locking might not effectively prevent the access to the locked resource state since the same data may be exposed by a different resource that is not being locked. Managing the overall consistency when the same state is exposed via multiple resources remains a challenge for RESTful transaction models.

7.5 Heuristic generation

Most of the transaction models make use of heuristics when deciding on certain transaction parameters such as timeouts [41, 5]. In this particular case, generating a suitable timeout is a challenge because it not only affects the performance but also the correctness of the model, i.e., a premature timeout can decrease the performance or make the system consistently fail [5]. In scenarios that involve decentralization and distribution, heuristics generation is even more difficult since most of the information is not known in advance, and not known by a single party. Most of the RESTful transaction models do not provide algorithms nor guidelines for heuristic generation; thus, this remains a challenge.

7.6 Gap between research and industry

Though several transaction models have been proposed in the past decade, only few are used in industry. Out of the current approaches, the overloaded POST method seems to be the most widely used mechanism for REST transactions due to its simplicity and efficiency. However, it has a main disadvantage: it cannot handle distributed and decentralized authority scenarios.

It is worth taking a look at why the other approaches are not taking so much traction. One of the key issues is the complexity and overhead added by the transaction mechanisms. Another aspect is that they are defined on their own, when in practice they have to be integrated with existing development frameworks as well as to take into account other cross-cutting concerns, i.e., security. Thus, the challenge is to define a simple yet efficient REST-compliant protocol that provides transactional guarantees, which can be seamlessly integrated

[p] http://atompub.org/
[q] http://www.w3.org/TR/ldp/

with other technologies of the REST development stack.

8 Conclusions

There has been a great deal of discussion in forums, technical literature, and research literature about RESTful transaction models. One of the main contributions of this work is to accumulate the knowledge about RESTful transaction models and present it in a succinct yet comprehensive manner. The paper presents detailed descriptions of different REST transaction approaches and models and provides running examples of common transactional scenarios for a selected representative set of models. Further, the paper analyses the transaction models found using a comparison framework for RESTful transactions defined in this work. This analysis shows the current state of the art of RESTful transaction models and illustrates the gaps and challenges that the future RESTful transaction models have to fulfil.

The main conclusion of the analysis of the existing RESTful transaction models is that *one model does not fit all*. RESTful transaction scenarios are diverse in many dimensions and no transaction model fulfils the requirements of every scenario. On the contrary, these models are designed to cover specific scenarios. According to our analysis one good example is the reservation business model, for example, booking a hotel and a flight for a single trip where all the tickets have to be reserved and booked or none of them. The analysis shows that TCC is the best candidate when developing applications following this model, as it covers all the scenarios and addresses the identified challenges.

However, if an application does not fit under the reservation model, the TCC model cannot be applied; furthermore, none of the other alternatives is able to cope with all the scenarios described in Section 5.2. For example, the create and delete operations in a decentralized environment and failure handling are not supported by the alternative models analysed in the paper. The readers can use the analysis performed in this paper to identify which existing models fit their requirements based on the properties of those models and to understand their characteristics and limitations. This understanding enables the readers to choose the most appropriate model based on the application requirements. However, because the analysis shows a gap in the current state-of-the-art regarding isolation-preserving RESTful transaction models further research [57] has to be done to understand the feasibility of improving the existing models or defining new models that can cope with those scenarios. As this paper summarizes the state-of-the-art and shows how RESTful transaction models have evolved over the time, it could provide valuable input for such research.

Further, the paper presents seven challenges based on the analysis of the existing RESTful transaction models. The root of most of the challenges is the impedance mismatch between the strong consistency transaction guarantees (*i.e.*, the ACID properties) and the REST constraints. For instance, the management of uncommitted temporary resources to ensure the isolation property conflicts with the stateless REST constraint, making it a challenge to satisfy both properties in isolation-preserving REST transaction models. Further, these strong consistency models require the serialization isolation level which is commonly achieved through pessimistic locking. Nevertheless, pessimistic locking leads to several challenges regarding to availability, fairness guarantees, faults and misbehaviour in distributed environments. The models which do not require the isolation property (*e.g.*, TCC and TS2PC4RS) address most of the challenges but the models that are designed to guarantee the ACID properties still do

not address these challenges. Further research has to be done to see how these challenges can be addressed in strong consistency RESTful transaction models.

Acknowledgements

This research has been supported by the *4V: Volumen, Velocidad, Variedad y Validez en la gestin innovadora de datos (TIN2013-46238-C4-2-R)* project with the BES-2014-068449 grant, and the *ALM iStack* project of the Center for Open Middleware at the Universidad Politécnica de Madrid.

References

1. C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision," in *Proceedings of the 17th International World Wide Web Conference (WWW'08)*. Beijing, China: ACM, Apr 2008, pp. 805–814.
2. R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.
3. E. Maler and J. Hammond, "The Forrester Wave: API Management Platforms," Forrester Research, Cambridge, USA, Tech. Rep. Q1 2013, Feb 2013.
4. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. California, USA: Morgan Kaufmann, Sep 1992.
5. G. Pardon and C. Pautasso, "Towards distributed atomic transactions over RESTful services," in *REST: From Research to Practice*. Springer, Aug 2011, vol. 1, pp. 507–524.
6. R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.
7. E. Wilde and C. Pautasso, *REST: From Research to Practice*. Springer, 2011.
8. J. Gray, "The Transaction Concept: Virtues and Limitations," in *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB '81)*, vol. 7. VLDB Endowment, Sep 1981, pp. 144–154.
9. T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *Journal of ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, Dec 1983.
10. E. A. Brewer, "Towards Robust Distributed Systems," in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*. Portland, Oregon, USA: ACM, Jul 2000, pp. 7–8.
11. S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.
12. D. J. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer*, vol. 45, no. 2, pp. 37–42, Feb 2012.
13. W. Vogels, "Eventually Consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008.
14. D. Pritchett, "BASE: An ACID Alternative," *ACM Queue*, vol. 6, no. 3, pp. 48–55, May 2008.
15. J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA, USA: MIT Press, Jan 1985.
16. G. Weikum and H.-J. Schek, "Concepts and applications of multilevel transactions and open nested transactions," in *Database transaction models for advanced applications*, A. K. Elmagarmid, Ed. San Francisco, CA, USA: Morgan Kaufmann, 1992, pp. 515–553.
17. H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SZGMOD Conference*, ser. 3, vol. 16. San Francisco, US: ACM, May 1987, pp. 249–259.
18. P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd ed., ser. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Jun 2009.
19. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design, 5th edition*. Addison-Wesley, 2011.

20. J. Eder and W. Liebhart, "Workflow transactions," in *Workflow Handbook 1997*. John Wiley & Sons, 1997, pp. 195–202.
21. H. Schuldt, G. Alonso, C. Beeri, and H. Schek, "Atomicity and isolation for transactional processes," *ACM Trans. Database Syst.*, vol. 27, no. 1, pp. 63–116, 2002. [Online]. Available: http://doi.acm.org/10.1145/507234.507236
22. B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature reviews in Software Engineering," Keele University and Durham University, EBSE Technical Report, Jul 2007.
23. M. P. Papazoglou, "Web Services and Business Transactions," *World Wide Web: Internet and Web Information Systems (WWW)*, vol. 6, no. 1, pp. 49–91, Mar 2003.
24. S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, ser. The Springer International Series in Engineering and Computer Science. Norwell, MA, USA: Kluwer Academic Publishers, 1996, vol. 345.
25. M. Little, "Rest and transactions?" 2009, InfoQ article, [accessed 3-February-2014]. [Online]. Available: http://www.infoq.com/news/2009/06/rest-ts
26. S. Jacobs, "Thoughts on a RESTful Transaction Process in HTTP," Feb 2004, [accessed 3-February-2014]. [Online]. Available: \url{http://www.seairth.com/web/resttp.html}
27. R. Fielding, "Waka: A replacement for HTTP," 2002, ApacheCon US, [accessed 3-February-2014]. [Online]. Available: http://gbiv.com/protocols/waka
28. H. F. Nielsen and D. LaLiberte, "Editing the web: Detecting the lost update problem using unreserved checkout," World Wide Web Consortium, W3C Note, May 1999.
29. T. Raman and A. Malhotra, "Identifying Application State," World Wide Web Consortium, TAG Finding, Dec 2011.
30. L. Dusseault, "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)," Internet Requests for Comments, IETF, RFC 4918, June 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7232.txt
31. L. Richardson and S. Ruby, *RESTful web services: Web services for the real world*. CA, USA: O'Reilly Media, May 2007.
32. J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice: Hypermedia and Systems Architecture*. CA, USA: O'Reilly Media, Sep 2010.
33. T. Erl, R. Balasubramanians, C. Pautasso, and B. Carlyle, *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*, 1st ed. Prentice Hall, Aug 2012.
34. S. Allamaraju, *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. O'Reilly Media and Yahoo Press, Feb 2010.
35. L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Sep 2013.
36. R. Khare and R. N. Taylor, "Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. Scotland, UK: IEEE, May 2004, pp. 428–437.
37. L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, Aug 1974.
38. L. A. H. da Silva Maciel and C. M. Hirata, "An optimistic technique for transactions control using REST architectural style," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 664–669.
39. H. F. Korth, E. Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," in *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 1990)*. Brisbane, Australia: Morgan Kaufmann, Aug 1990, pp. 95–106.
40. R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests," Internet Requests for Comments, IETF, RFC 7232, June 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7232.txt
41. A. Marinos, A. Razavi, S. Moschoyiannis, and P. Krause, "RETRO: A consistent and recoverable RESTful transaction model," in *Proceedings of the 7th IEEE International Conference on Web Services (ICWS 2009)*. San Sebastian, Spain: IEEE, Jul 2009, pp. 181–188.

42. A. Razavi, A. Marinos, S. Moschoyiannis, and P. Krause, "RESTful transactions supported by the isolation theorems," in *Proceedings of the 9th International Conference on Web Engineering (ICWE 2009)*. Springer Berlin Heidelberg, 2009, pp. 394–409.
43. L. A. H. da Silva Maciel and C. M. Hirata, "A timestamp-based two phase commit protocol for web services using rest architectural style," *Journal of Web Engineering*, vol. 9, no. 3, pp. 266–282, Sep 2010.
44. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, ser. Mcgraw-Hill Computer Science Series. McGraw-Hill College, Mar 1985.
45. G. F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, May 2011.
46. G. Pardon and G. Alonso, "Cheetah: a lightweight transaction server for plug-and-play internet data management," in *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000)*. Cairo, Egypt: Morgan Kaufmann Publishers Inc., Sep 2000, pp. 210–219.
47. C. Pautasso and E. Wilde, "Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design," in *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. Madrid, Spain: ACM, Apr 2009, pp. 911–920.
48. G. Pardon and C. Pautasso, "Atomic Distributed Transactions: a RESTful Design," in *Proceedings of the companion publication of the 23rd international conference on World wide web*. Seoul, South Korea: ACM, Apr 2014, pp. 943–948.
49. S. Kochman, P. T. Wojciechowski, and M. Kmieciak, "Batched transactions for RESTful web services," in *Current Trends in Web Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, June 2012, vol. 7059, pp. 86–98.
50. A. Razavi, A. Marinos, S. Moschoyiannis, and P. Krause, "Recovery management in RESTful interactions," in *Proceedings of the 3rd IEEE International Conference on Digital Ecosystems and Technologies (DEST'09)*. IEEE, Jun 2009, pp. 419–424.
51. L. A. H. da Silva Maciel and C. M. Hirata, "Fault-tolerant timestamp-based two-phase commit protocol for RESTful services," *Software: Practice and Experience, Special Issue on Web Technologies*, vol. 43, no. 12, pp. 1459–1488, Dec 2013.
52. ——, "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)*. TaiChung, Taiwan: ACM, Mar 2011, pp. 778–785.
53. C. Pautasso and M. Babazadeh, "The Atomic Web Browser," in *Proceedings of the 22nd international conference on World Wide Web (WWW'13)*, Rio de Janeiro, Brazi, May 2013, pp. 217–218.
54. N. Mihindukulasooriya, M. Esteban-Gutiérrez, and R. García-Castro, "Seven challenges for RESTful transaction models," in *Proceedings of the companion publication of the 23rd international conference on World wide web*, Seoul, South Korea, Apr 2014, pp. 949–952.
55. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
56. M. Lanthaler and C. Guetl, "Hydra: A Vocabulary for Hypermedia-Driven Web APIs." in *Proceedings of the 6th Workshop on Linked Data on the Web (LDOW2013)*, ser. CEUR Workshop Proceedings, vol. 996, May 2013, pp. 11–16.
57. N. Mihindukulasooriya, R. García-Castro, and A. Gómez-Pérez, "A Distributed Transaction Model for Read-Write Linked Data Applications," in *Engineering the Web in the Big Data Era*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9114, pp. 631–634.