

# Batched Transactions for RESTful Web Services

Sebastian Kochman, Paweł T. Wojciechowski, and Miłosz Kmiecik

Poznań University of Technology

**Abstract.** In this paper, we propose a new transaction processing system for RESTful Web services; we describe a system architecture and algorithms. Contrary to other approaches, Web services do not require any changes to be used with our system. The system is transparent to non-transactional clients. We achieve that by introducing an overlay network of mediators and proxy servers, and restricting transactions to be a batched set of REST/HTTP operations (or requests) on Web resources addressed by URIs. To be able to use existing Web hosts that normally do not support versioning of Web resources, transaction resources are currently modified in-place, with a simple compensation mechanism. Concurrent execution of transactions guarantees isolation.

## 1 Introduction

The *REpresentational State Transfer (REST)* [5] is an architecture style for Web-based applications. It has gained, and is still gaining, enormous popularity due to its simplicity, scalability and, interoperability – thanks to wide acceptance of the Hypertext Transfer Protocol (HTTP)<sup>1</sup>. REST offers simplicity at the expense of lacking some standards well supported in other styles. For example, SOAP-based Web Services have a WS-AtomicTransaction [12] standard for transaction processing, while REST currently lacks a similar standard (see also [13]).

Transaction processing is a broad and complex issue. We may consider transactions on different levels of abstraction and supporting different properties. Transactions usually model operations that have to be executed atomically, i.e., they should either be executed completely and successfully or not at all. Alternative approaches are, e.g. Sagas [16] that do not guarantee atomicity. Transactions are a very useful abstraction of the real world business operations. That's why transaction processing could be a valuable extension to REST.

Although there are many interesting proposals of introducing transactions to REST, no one has gained wide acceptance among the community. In most cases such systems or patterns arguably break some of the REST style principles. For example, the client-server communication is constrained by no client context being stored on the server between requests. This *statelessness* constraint – a key requirement in relation to RESTful Web services – is often a subject of discussions about interpretation. Developers of systems (including some transaction systems described later) often worked around the issue of statelessness constraint by giving the session state a resource identifier on the server side. An

---

<sup>1</sup> In this paper, we focus on REST over HTTP and URI.

interpretation proposed in [3] disallows it and claims that such a design cannot be called RESTful. But it might be REST with some exceptions, and this "with exceptions" approach is probably right for most enterprise architectures.

In this paper, we introduce *Atomic REST* – a new, lightweight transaction system. We restrict transactions to be a batched set of REST/HTTP operations (or requests) on Web resources addressed by URIs. While other proposals of transactions in REST are mostly software design patterns or libraries to be used by the client/server implementers, our approach is different: most of the transaction processing work is done by separate services – proxies and mediators – communicating using an overlay network. In particular, Web services do not require any changes to be used with our system. Moreover, the system is transparent to the clients that do not require transactions. These features enable easy integration of Atomic REST with existing RESTful Web services.

To be able to use existing Web hosts that usually do not support versioning of Web resources, transaction resources are currently modified in-place, with a (best-effort) compensation mechanism that is based on the symmetry of HTTP operations. Concurrent execution of transactions guarantees isolation. In the paper, we describe the Atomic REST's design and algorithms. To demonstrate the main concept, we develop a prototype implementation of Atomic REST; more information is available on the project page [7].

The paper has the following structure. First, we discuss related work. Then, the main idea of our transaction system is presented in Section 3. In Section 4 we describe the algorithms that we designed for Atomic REST, followed by the discussion of their properties and proofs of isolation in Section 5. Next, we briefly describe an example validation test of our experimental implementation in Section 6. Finally, we conclude.

## 2 Related Work

*Pessimistic Transactions* One of the first proposals of atomic transactions in REST is described by JBoss [10]. It is an extension of JAX RS – a popular Java API for RESTful Web services, with atomic transactions based on exclusive locks represented as resources on the server side. Contrary to their approach, we have adopted a different architecture, introducing separate services (mediators) that are responsible for the execution of transactions, and using server proxies that allow the services to remain unaware of transaction processing.

A similar approach to [10] is represented by RETRO [8] – a transaction model that defines many fine-grained resources for transaction processing, with a choice of exclusive and shared locks. We are not aware of any RETRO implementation announced yet. Some authors [15] pointed out drawbacks of this model: cluttering the business representations with transactional entities and the complexity that makes programming cumbersome.

*Optimistic Transactions.* Optimistic concurrency control [2] fits REST better than pessimistic transactions because it increases availability of a Web service by decreasing resource blocking. Below we characterize example approaches.

The most common solution for providing atomic transactions to REST is using the POST method to execute a batched set of operations. The concurrency control is optimistic since the data is cached by a client, and the consistency of the cache is checked during commit-time. The main advantage of the overloaded POST-based solution is its simplicity. On the other hand it is often criticized because it does not respect the semantics of uniform interface methods [5]: POST should create a resource, not execute any operations. Moreover, the mechanism is quite limited, e.g. contrary to Atomic REST, it does not allow transactions spanning many services.

Overloading the POST method is used, among other systems, in the cloud computing environments, such as Microsoft Windows Azure [6]. It offers structured storage in the form of tables with a REST-compliant API, enabling to perform a transaction across entities stored within the same table and partition. An application can atomically perform multiple Create/Update/Delete operations across multiple entities in a single batch request to the storage system, as long as the entities in the batch operation have the same partition key value and are in the same table [9]. Thus, the high scalability and accessibility of the service is achieved by introducing the limitation on the set of resources that may be included in one transaction.

A simple design pattern that provides transactions in REST is described in [14]. A new transaction is created by sending a POST to a factory resource. Once the transaction is created successfully, we can access it as a “gateway” to the main service, sending all possible HTTP requests to a variety of resources. The pattern is simple and seems to be effective, but is it RESTful? In the same book, the authors emphasize the difference between application state and resource state. The user transaction is, in fact, the application state, therefore it should not be maintained by the server. Exposing it as resources does not change anything. In fact, the authors admit that their proposal is not “the official RESTful or resource-oriented way to handle transactions” – it is just “the best one they could think up”. On the other hand, even if the pattern breaks the statelessness constraint of REST, it is a clean concept that can work successfully for a variety of services.

*Compensation.* An alternative (or complementary) approach to atomic transactions is the transaction compensation mechanism, which can be well suited for some applications. Operations are executed normally, and in case of a failure, the compensation procedure is called in order to revert the transaction’s changes. Let’s consider an example of a holiday trip. One would like to reserve an airline ticket, a hotel room and a bus trip to a national park. One wants only all or nothing. How can we provide such transaction semantics over several autonomous systems? This problem is solved frequently with compensation, even though it does not guarantee atomicity. However, it may provide an acceptable contract: a high probability of success and acceptable side effects in case of failure (e.g. a cancellation fee at one of the services).

A model of compensating RESTful transactions, called *jfdi*, has been made available by JBoss [11]. To our best knowledge, the model has not been

implemented yet. In terms of the interface, it is similar to JBoss's lock-based transactions that we described earlier. Although it does not provide any locks, the compensation logic is held on the server side – the transaction objects and compensation controllers (called *compensators*) are exposed as resources. It is very comfortable for the client that does not need to know how to compensate each operation on that particular service. However, similarly to pessimistic transactions, this design decision invalidates the statelessness constraint.

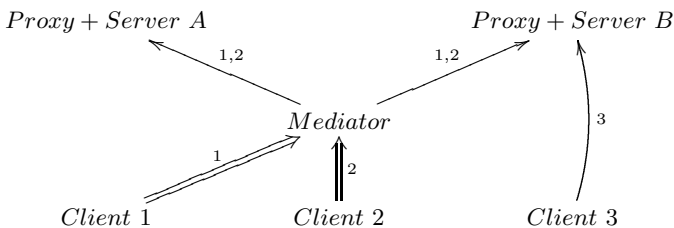
A popular compensation pattern known as Saga (described, e.g. in [16]) differs from jfdi in many respects. Saga is not intended to be a product, it is just a software pattern. In Saga, the whole compensation logic is held on the client's side. An advantage is that services do not have to be prepared anyhow. However, a given client is specific for a certain case, and so it can be hard to extend the client's code to work with other services. On the other hand, jfdi is more service-centric; it can only support services that use jfdi, but clients can be much simpler, generic and, more reusable.

This section shows that the existing transaction processing patterns are either too limited or do not produce generic, reusable clients. On the other hand, service-centric products often break the REST statelessness constraint. When designing our system, we have used the best ideas from the work described above but at the same time we tried to develop a fresh approach. In the following section we describe our system.

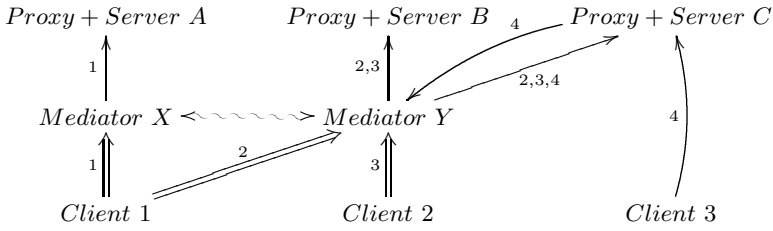
### 3 Basic Definitions

We explain the main idea of our system using two examples of interaction patterns in Figures 1 and 2. We can identify four components (ignore the arrows for a while):

- *Server* – provides a user-defined RESTful Web service, executing client requests and returning results, without knowledge of Atomic REST;
- *Client* – a user-defined client, with or without knowledge of Atomic REST;
- *Mediator* (or *Transaction Manager*) – a Web service managing transaction execution on behalf of the client;



**Fig. 1.** An example interaction pattern of Atomic REST (single mediator)



**Fig. 2.** An example interaction pattern of Atomic REST (many mediators)

- *Proxy* – a server’s *façade*, intercepting messages addressed to the server and handling Atomic REST-specific requests; the proxy enables a RESTful Web service to support transactions without any changes in its code.

A *distributed transaction* in our system (or a *transaction*, in short) is a batch of REST operations (or requests) to be applied to different resources maintained by servers. Thus, from a client view-point a transactional request does not differ much from an ordinary HTTP request. This means that clients are able to cache transaction responses, which fulfills one of the REST architectural constraints, i.e. *cacheable responses*. Execution of concurrent transactions satisfies the isolation property and a weak form of atomicity, described in Section 5.

Batching of transaction operations restricts transactions to be rather short and non-interactive (similarly to, e.g. Sinfonia [1]). Hence the time when resources are blocked by a client is reduced to a minimum. This means that our system could be deployed on the Web and platforms, such as those provided for cloud computing, in which dependencies between network nodes should be avoided and the request processing time has to be short.

A client can submit several requests to be executed by many servers, as a single transaction. For example, in Figure 1 there is a single mediator and two clients who submit their transactions 1 and 2 to the mediator for execution. The transactions request some resources on servers *A* and *B*. The mediator executes transactions sequentially, first 1, then 2. Thus, isolation is satisfied. At the same time, client 3 submits a non-transactional request to server *B* that does not conflict with the transactions and is handled by server *B* normally.

In Figure 2, there is an example with many mediators. Introducing many mediators supports privacy and load balancing. Each server gets transactions only from its trusted (single) mediator, e.g. server *A* trusts only mediator *X*. Each mediator can handle many servers. Mediators could be replicated for fault-tolerance if required. Client 1 executes transactions 1 and 2 using, respectively mediator *X* and *Y*, while client 2 uses only *Y*. At the same time client 3 submits a non-transactional request to server *C*. Since the request conflicts with the transactions, it is forwarded to *B*’s mediator *Y* as a transaction containing only a single request. At the end, all results will be returned to the clients. In order to agree upon the order of transaction execution, mediators communicate using a coordination protocol described below.

## 4 The Atomic REST Algorithm

Below we describe the algorithms that are executed by mediators, proxies and clients. For clarity, we omitted some details. The symbols used in pseudocode are as follows:

$P$	a proxy or an URI of the proxy/server, depending on the context
$t_k$	a transaction's unique identifier
$op$	a $t_k$ 's single HTTP operation (or request) to be executed
$resource$	a resource (defined by an URI) on which to execute the operation
$\langle xml \rangle$	an operation (or request) payload
$M_{t_k}$	a set of mediators coordinating the execution of a transaction $t_k$
$m_{t_k}$	a leader mediator of a transaction $t_k$
$O_{t_k}$	a set of $t_k$ 's HTTP operations (or requests), of the form $\langle op\ P/resource\ \langle xml \rangle \rangle$
$O_{t_k}^m$	a set of $t_k$ 's HTTP operations (or requests), to be submitted for execution by mediator $m$
$R_{t_k}^m$	a set of $t_k$ 's resources whose servers trust mediator $m$ (we omit $m$ when the mediator is known from the context)
$R_P$	a set of resources to be marked as "transactional" at proxy $P$
$r$	a fine-grained resource controlled by the Read/Write or Intention-to-{Read Write} locks
$res$	a result of operation (or request) execution

### 4.1 The Single Mediator's algorithm

Algorithm 1 defines the mediator's behaviour, assuming only one mediator in the system (this corresponds to our current implementation of Atomic REST). Later, we extend this algorithm to support many mediators.

When a mediator  $m_{t_k}$  receives a transaction  $t_k$  from a client, it first extracts all  $t_k$ 's resources  $R_{t_k}$  whose servers trust this mediator (lines 1-5). Then, it tries to grab fine-grained locks on these resources atomically (lines 23-26). We describe the locking mechanism used in Atomic REST below. If succeeded, the mediator requests all proxies required to execute transaction  $t_k$ , to set all their resources required by  $t_k$  into the "transactional" mode (lines 27-30). Otherwise, it enqueues  $t_k$  into  $m_{t_k}$ 's First-In-First-Out queue of transactions  $Q$  (lines 32-33).

Next, mediator  $m_{t_k}$  synchronously sends the transaction  $t_k$ 's HTTP requests (or operations)  $\langle op_i\ P_i/resource_i\ \langle xml \rangle_i \rangle$  to a corresponding proxy/server  $P_i$  for execution, and collects the results (lines 7-8).

If some operation failed, e.g. due to the "503 Service Unavailable" error, all  $t_k$ 's operations executed so far must be withdrawn. Since, our system is intended to be used with existing RESTful Web services that normally do not support multiversioning of resources, all transaction operations modify resources in-place. Thus, the only way to withdraw the operations that have already been executed by a transaction is to compensate them (lines 9-19); below we explain it.

Finally, the locks on resources are released, non-conflicting transactions are dequeued (line 20) and the composite result is returned to the client (line 21).

---

**Algorithm 1.** A single mediator  $m_{t_k}$ 's code.
 

---

```

1: receive  $\langle \text{PUT } m_{t_k} / \text{transaction} / t_k \ \{ \{ m_{t_k} \} O_{t_k} \} \rangle$ : // a transaction from a client
2: return execute-transaction( $t_k, m_{t_k}, O_{t_k}$ )
3:
4: function execute-transaction( $t_k, m_{t_k}, O_{t_k}$ ):
5:  $R_{t_k} \leftarrow \text{resources-of}(O_{t_k}, m_{t_k})$  // get resources whose servers trust this mediator
6: lock-resources( $t_k, m_{t_k}, O_{t_k}, R_{t_k}$ )
7: for all  $\langle op_i P_i / \text{resource}_i \ \langle xml \rangle_i \rangle \in O_{t_k}$ , where  $i = 1..n$  and  $n = |O_{t_k}|$  do
8:    $res_i \leftarrow \langle op_i P_i / \text{resource}_i \ \langle xml \rangle_i \rangle$ 
9:   if  $res_i = \text{error}$  then
10:     //  $op_i$  failed - depending on the error, compensate  $op_i$  or not
11:     for all  $j = 1..i - 1$  do
12:        $res_j \leftarrow \text{compensate} \ \langle op_j P_j / \text{resource}_j \ \langle xml \rangle_j \rangle$ ,
13:       where  $res_j \in \{\text{compensation-ok}, \text{compensation-failed}\}$ 
14:     end for
15:     for all  $j = i + 1..n$  do
16:        $res_j \leftarrow \text{compensation-ok}$  // since  $op_j$  ( $j > i$ ) not executed yet
17:     end for
18:   end if
19: end for
20: unlock-resources( $R_{t_k}$ )
21: return  $\{(1, res_1), \dots, (n, res_n)\}$ 
22:
23: function lock-resources( $t_k, m_{t_k}, O_{t_k}, R_{t_k}$ ):
24: // try to lock  $t_k$ 's resources  $R_{t_k}$ ; for clarity of pseudocode, semaphores are used
25: // but our implementation uses Read/Write and Intention-to- $\{\text{Read}|\text{Write}\}$  locks
26: if atomically  $\forall r_j \in R_{t_k}$  semaphore $_j$ .P( $r_j$ ) succeeded then
27:   for all  $R_{P_i} \in R_{t_k}$ , where  $i = 1..w$  do
28:     // set "transactional" mode for  $t_k$ 's resources  $R_{P_i}$  at proxy  $P_i$ 
29:      $\langle \text{PUT } P_i / \text{atomicrest} / R_{P_i} \ 1 \rangle$ 
30:   end for
31: else
32:   Q.enqueue( $t_k, m_{t_k}, O_{t_k}$ ) // enqueue  $t_k$  and wait till  $t_k$  dequeued
33:   lock-resources( $t_k, m_{t_k}, O_{t_k}, R_{t_k}$ )
34: end if
35:
36: function unlock-resources( $R_{t_k}$ ):
37: for all  $R_{P_i} \in R_{t_k}$ , where  $i = 1..w$  do
38:   // unset "transactional" mode for  $t_k$ 's resources  $R_{P_i}$  at proxy  $P_i$ 
39:    $\langle \text{PUT } P_i / \text{atomicrest} / R_{P_i} \ 0 \rangle$ 
40: end for
41: for all  $r_j \in R_{t_k}$  do
42:   semaphore $_j$ .V( $r_j$ ) // unlock  $t_k$ 's resource  $r_j$ 
43: end for
44: dequeue-trans() // dequeue non-conflicting transactions in a queue Q
45:
46: function dequeue-trans(), where  $Q = (t_1, m_{t_1}, O_{t_1}); \dots; (t_n, m_{t_n}, O_{t_n}), n = |Q|$ :
47: for all  $i = 1..l$  ( $l \leq n$ ) do
48:   Q.dequeue( $t_i, m_{t_i}, O_{t_i}$ ), where
49:    $\forall a, b \in \{1, \dots, l\}$  resources-of( $O_{t_a}, m_{t_a}$ )  $\cap$  resources-of( $O_{t_b}, m_{t_b}$ ) =  $\emptyset$ 
50: end for

```

---

**Algorithm 2.** Proxy  $P_i$ 's code.

---

```

1: receive  $\langle op P_i/resource \langle xml \rangle \rangle$ , where  $op \in \{POST, PUT, GET, DELETE\}$ :
2: if  $transaction\_mode(resource) = 0$  or (optionally)  $op = GET$  then
3:   //  $resource$  not in a “transactional” mode or uncommitted read allowed
4:    $res \leftarrow \langle op P_i/resource \langle xml \rangle \rangle$  // execute  $op$  by a server normally
5: else
6:   // some transaction locked  $resource$ , pass  $op$  as a transaction to  $P_i$ 's mediator
7:    $(1, res) \leftarrow \langle PUT m/transaction/ \langle \{m\} \{ \langle op P_i/resource \langle xml \rangle \} \} \rangle \rangle$ 
8: end if
9: return  $res$ 

```

---

*Resource locking.* To simplify the pseudocode, we have used fine-grained semaphores to block access to resources. However, in our implementation, we use multigranularity locks [2] of four types: Read, Intention-to-Read, Write and Intention-to-Write. The former two are used for GET and HEAD operations, while the latter two for PUT, POST, and DELETE.

The Intention-to-{Read|Write} locks are acquired on “ancestor” resources of the locked resources. For instance, consider a transaction containing an operation GET `http://www.service.org/books/medicine?author=foo`. In this case, the Intention-to-Read lock should be acquired both on `http://www.service.org/` and `http://www.service.org/books/` before the Read lock will be acquired on `http://www.service.org/books/medicine`.

Since all locks of a subtransaction are taken by each mediator locally as a single atomic operation, no deadlock can occur due to the locking order.

*Compensation.* To *compensate* an operation on some resource means to execute a complementary operation on this resource. Transaction compensation could be provided by a user or done automatically whenever possible. For the latter, the following table presents REST/HTTP and complementary operations that are used by Atomic REST for compensation (details omitted due to lack of space):

Request	Compensating request
GET	no compensation needed
PUT (modification)	PUT
PUT (creation)	DELETE
POST (creation)	DELETE
DELETE	PUT

## 4.2 The Proxy's Algorithm

The  $P_i$  proxy (see Algorithm 2) simply receives HTTP requests, executes them, and returns results. If any resource required by an operation  $op$  is in a “transactional” mode, i.e. there exists some transaction that can access this resource exclusively and  $op \neq GET$ , the operation cannot be directly executed by the server at  $P_i$ . In such a case, the operation is forwarded to  $P_i$ 's trusted mediator



**Algorithm 3.** Client's transactional code.

---

```

1: // get a set  $M_{t_k}$  of all mediators required to execute  $O_{t_k}$  as a transaction
2:  $M_{t_k} \leftarrow \emptyset$ 
3: for all  $P_i \in \text{servers-of}(O_{t_k})$  do
4:    $m \leftarrow \langle \text{GET } P_i / \text{atomicrest} / \text{mediator} \rangle$ 
5:    $M_{t_k} \leftarrow M_{t_k} \cup \{m\}$ 
6: end for
7: // get the transaction's unique ID  $t_k$  and use any mediator  $m \in M_{t_k}$  to execute  $t_k$ 

8:  $t_k \leftarrow \langle \text{POST } m / \text{transaction} \rangle$ , where  $m \in M_{t_k}$ 
9:  $\text{result} \leftarrow \langle \text{PUT } m / \text{transaction} / t_k \langle M_{t_k} O_{t_k} \rangle \rangle$ 

```

---

**Algorithm 4.** Mediator  $m_i$ 's code (many mediators possible).

---

```

1: receive  $\langle \text{PUT } m_i / \text{transaction} / t_k \langle M_{t_k} O_{t_k} \rangle \rangle$ :
2: //  $m_{t_k} \leftarrow m_i$ , send transaction  $t_k$  to all mediators using a total order broadcast
3:  $\text{atomic-bcast}(t_k, M_{t_k}, O_{t_k})$  to all mediators in  $M_{t_k}$ 
4: // collect transaction  $t_k$ 's partial results from all mediators and return to a client
5: while  $\exists m \in M_{t_k} \text{ result}_{t_k}[m] = \emptyset$  do
6:   ()
7: end while
8: return  $\text{result}_{t_k}$ 
9:
10: receive  $\text{atomic-bcast}(t_k, M_{t_k}, O_{t_k})$  from  $t_k$ 's leader mediator  $m_{t_k}$ :
11: // get transaction's part that can be executed by this mediator
12:  $O_{t_k}^{m_i} \leftarrow \text{get-my-part}(m_i, O_{t_k})$ 
13:  $\text{res} \leftarrow \text{execute-transaction}(t_k, m_i, O_{t_k}^{m_i})$ 
14:  $\text{send-result}(t_k, \text{res})$  to  $m_{t_k}$ 
15:
16: receive  $\text{send-result}(t_k, \text{res})$  from  $m$  // executed if  $m_i$  is a leader (i.e.  $m_i = m_{t_k}$ )
17:  $\text{result}_{t_k}[m] \leftarrow \text{res}$ 

```

---

$m$  that will execute it as a single-operation transaction (line 7). Otherwise, the operation can be executed by the server as a normal HTTP request (line 4). For efficiency, we allow uncommitted reads by non-transactional clients. If this behaviour is not acceptable, the “**or**  $op = \text{GET}$ ” must be removed (line 2).

### 4.3 The Client's Algorithm

To execute a transaction (see Algorithm 3), the client first obtains a list of mediators (lines 2-6) and then chooses one of them to get a transaction's unique ID and to pass the transaction to it for execution (lines 9-10).

### 4.4 The Many Mediators' Algorithm

Algorithm 4 extends Algorithm 1 to support many mediators. Upon delivery of a transaction  $t_k$  from a client a mediator becomes a leader and broadcasts  $t_k$  to

all mediators that are required to execute it (including itself). For this, a total order broadcast (also known as atomic broadcast) [4] is used. After getting  $t_k$ , each mediator extracts the part of  $t_k$  that refers to resources accessible by the mediator, and executes it (lines 10-14); we call each such a part a *subtransaction*. The leader collects results and returns them to the client (lines 5-8, 16-17).

## 5 Properties

Below we discuss the isolation and atomicity properties that are guaranteed by the Atomic REST algorithm (their definitions follow the ACID properties [2]).

### 5.1 Isolation

Below we prove that our algorithm satisfies isolation. In case of fatal failures (defined in Section 5.2), the property can be guaranteed *up to* the fatal failure.

**Lemma 1.** *All subtransactions executed by each mediator are isolated.*

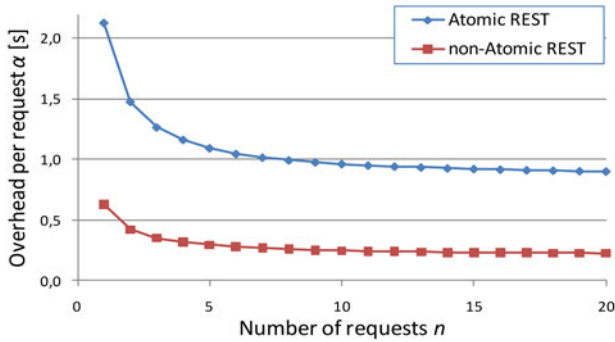
*Proof.* Each subtransaction is described by a set of resources. Since locks on the resources are taken atomically and released after a subtransaction is finished, no subtransactions sharing resources can be executed concurrently. This means that they are executed sequentially, so they are isolated. Subtransactions that do not share resources may be executed concurrently but they are isolated trivially. From this we get that all subtransactions are isolated.  $\square$

**Theorem 1.** *The Atomic REST algorithm guarantees transaction isolation.*

*Proof.* Proof by contradiction. Let us assume that two transactions  $t$  and  $t'$  are not isolated. From Lemma 1, we get that for every mediator  $m_i$  ( $m_i \in M$ ) all subtransactions of  $t$  and  $t'$  executed by  $m_i$  (if any) are isolated. Thus, if  $t$  and  $t'$  are not isolated, then there must exist two mediators  $m_1$  and  $m_2$ , such that  $m_1$  first executes a subtransaction  $subt_{m_1}$  of transaction  $t$ , then a subtransaction  $subt'_{m_1}$  of transaction  $t'$ , while  $m_2$  executes its subtransactions of transactions  $t'$  and  $t$  in the opposite order. But this is not possible since by atomic broadcast semantics, all mediators receive transactions in the same global order. Thus, if  $m_1$  executes  $subt_t^{m_1}$  followed by  $subt_{t'}^{m_1}$ , then  $m_2$  must execute  $subt_t^{m_2}$  followed by  $subt_{t'}^{m_2}$ . Moreover, non-transactional requests cannot modify resources processed by a transaction since they are guarded by a “transactional” mode that is kept during transaction execution. Hence by Lemma 1, it is possible to serialize all transactions, which satisfies isolation.  $\square$

### 5.2 Atomicity

If there are no errors, transactions are executed atomically. If there are errors, transaction atomicity is currently provided by the compensation mechanism. Since automatic compensation of transaction operations is not always possible



**Fig. 3.** Overhead induced by Atomic REST

(either due to HTTP-bound issues or application semantics), our system cannot guarantee atomicity in all cases. This is acceptable since Atomic REST is no more tolerant to failures than an average Web service that can fail at any time. Therefore, in case of some failures, some transaction operations may or may not have been executed or compensated; we call such failures *fatal*. Thus, the client should always check the results returned by the system, and in case of some error messages, execute a suitable action. For example, the client could repeat transaction operations. For instance, the PUT, DELETE and GET methods are *idempotent* methods, and so they can be repeated many times.

Implementing stronger semantics of atomicity would require significant changes to the code of Web services, such as resource multiversioning and the 2PC (or 3PC) protocol for the mediator-server communication. Multiversioning would allow transaction operations to be executed on shadow copies of resources, made public on transaction commit and rejected on transaction abort. However, we think that supporting existing Web services by Atomic REST compensates the drawback of a weaker atomicity semantics. Moreover, we intend our transactions to be a mechanism to increase expressiveness in Web programming, rather than for implementing fault-tolerant Web services.

## 6 Validation

We define *overhead value* per an individual REST/HTTP request (or operation), imposed by our transaction system, as  $\alpha = \frac{\Delta - \Delta_{min}}{n}$  (in seconds), where  $n$  is the number of requests executed by a transaction,  $\Delta$  is the elapsed time between sending the first transaction request and receiving response to the last request, and  $\Delta_{min}$  is the lower bound of transaction's total processing time, computed as  $\Delta_{min} = n * \delta$  (in seconds), where  $\delta$  is the time of processing every request by the server (in our tests we choose it to be a constant value of the sleeping time before a server can accept another client request).

In Figure 3 we show the result of an example validation test<sup>1</sup>. We can see that while  $n$  is increasing the overhead time per request asymptotically reaches a constant value. Thus, the total overhead time per transaction is linearly dependant on the number  $n$  of transaction requests. We have used a least squares method to compute the linear regression slope value, and obtained that the overhead of Atomic REST is between 3.4 and 4 times larger than the overhead of non-transactional client-server processing of the same REST/HTTP requests (denoted as non-Atomic REST in Figure 3), when compared to a purely local processing of these requests by the Web server.

## 7 Conclusion

The algorithms that we designed in this paper enabled us to develop a distributed lightweight transaction system for REST that enjoys clean design, conformance to REST constraints, support of existing Web services and transparency for non-transactional clients. Moreover, validation results of our experimental single-mediator implementation show that the overhead is acceptable.

*Acknowledgments.* This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08.

## References

1. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: A new paradigm for building scalable distributed systems. In: Proc. SOSP 2007 (2007)
2. Bernstein, P.A., Newcomer, E.: Principles of Transaction Processing. Morgan Kaufmann (2009)
3. Carlyle, B.: The REST statelessness constraint (June 2009), <http://soundadvice.id.au/blog/2009/06/13/#stateless>
4. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys 36(4), 372–421 (2004)
5. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
6. Haridas, J., Nilakantan, N., Calder, B.: Windows Azure Table. Microsoft (2009)
7. IT-SOA. Atomic REST (2011), <http://www.it-soa.eu/atomicrest>
8. Marinos, A., Razavi, A., Moschoyiannis, S., Krause, P.: RETRO: A consistent and recoverable RESTful transaction model. In: Proc. ICWS 2009 (July 2009)
9. Microsoft. Windows Azure - Team Blog, <http://blogs.msdn.com/windowsazure>
10. Musgrove, M.: (February 2009), <http://community.jboss.org/wiki/TransactionalSupportforJAXRSbasedapplications>

---

<sup>1</sup> Configuration: Intel Xeon QuadCore X3230 @2.66GHz with 4MB cache and 4GB RAM. Operating system: *openSUSE 10.3* with *Sun's JRE 1.6.0*

11. Musgrove, M.: Compensating RESTful Transactions (June 2009), <http://community.jboss.org/wiki/CompensatingRESTfulTransactions>
12. OASIS. Web Services Atomic Transaction, Version 1.2 (February 2009)
13. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In: Proc. WWW 2008 (2008)
14. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (2007)
15. Rotem-Gal-Oz, A.: Transactions are bad for REST (June 2009), <http://www.rgoarchitects.com/nblog/2009/06/15/TransactionsAreBadForREST.aspx>
16. Rotem-Gal-Oz, A., Bruno, E., Dahan, U.: Saga. In: SOA Patterns, ch. 5.4. Manning Publications Co. (June 2007)