TEMPORAL REST – HOW TO REALLY EXPLOIT XML

Marc Kramis
Department of Computer and Information Science, University of Konstanz
marc.kramis@uni-konstanz.de

Georgios Giannakaras
Department of Computer and Information Science, University of Konstanz
georgios.giannakaras@uni-konstanz.de

ABSTRACT

The eXtensible Markup Language (XML) is more than a unified data exchange and storage format. We suggest the exploitation of XML and look at it as a fine-granular node tree, which is grown up through a sequence of user modifications. The Representational State Transfer (REST) is the perfect candidate to expose XML resources as well as their full revision and modification history to the World Wide Web. Extending the idea of XML and REST along the natural modification-driven temporal dimension breeds something, which is scalable, robust, simple, and yet extensible enough to effectively enrich striving applications such as personal information management, collaborative document authoring, distributed content management, or geographic visual analytics. In this paper, we introduce Temporal REST, i.e., an interface and protocol to access web-based XML resources as well as their full revision and modification history. We describe the underlying data model and show how it solves problems inherently arising from temporal interactions in a pragmatic and straightforward way. In addition, we provide a case study to demonstrate the power of Temporal REST due to its elegance and true simplicity. Finally, we motivate future work including the implementation of back-end services as well as front-end applications – both of which will mutually benefit from Temporal REST.

KEYWORDS

Temporal, REST, XML, Revision, Modification, History

1. INTRODUCTION

1.1 The importance of REST and XML

Twelve years after the introduction of HTTP, Roy Fielding coined the word REST [1]. REST is a set of network architecture principles, which outline how resources are defined and addressed. Practically speaking, REST defines a simple and scalable interface to exchange resources over HTTP. Each resource must be uniquely addressable through hypermedia links meeting a universal syntax. A well-defined and typically small set of HTTP operations specifies how to proceed with the obtained resource. The basic operations are POST to create a resource, GET to read a resource, PUT to update a resource, and DELETE to remove a resource. RESTful web services have appeared all over the Internet and compete with already-established protocols. The simplicity and elegance of REST makes alternatives such as the XML-based SOAP, binary CORBA [2], or DCOM [3] look like unhandy fellows. Web application frameworks such as Ruby on Rails [4] quickly adopted and favored REST. Virtually any programming language or framework nowadays has tools, e.g., Restlet for Java [5] or Astoria for .NET [6], to facilitate RESTful application development. However, Fielding did not provide a detailed description on how to use REST for a specific application. It is left to the developer of each application to specify how exactly the interface should look like and how the resources should be accessed.

In the wake of the unprecedented growth of the Internet, the need for a unified resource-encoding format culminated in the standardisation of XML. Since then, XML has started to conquer the world as a universal data exchange and storage format. The human-readability of XML along with its rich toolset consisting of

XPath, XSLT, XQuery, among others, lead to a quick adoption of XML for protocols such as SOAP, which allows to access web-based objects, BPEL [7], which allows the modeling of high-level business logic, or Atom [8], which is a protocol to feed news. Even the shady side of XML, i.e., its sheer verbosity and excessive demand for processing power could not really impair its success. Rather, more and more traditional relational database systems such as Microsoft SQL Server [9], Oracle [10], or IBM DB2 [11] have started to natively store XML data types for improved performance and interoperability. Other database systems, e.g., X-Hive [12], no longer support the traditional relational model but focus on native XML storage. In contrast to traditional (object-) relational databases, XML has a convenient feature: It supports a data-before-schema approach, which does not require the specification of a schema before the storage of any data. Finally, the Efficient XML Interchange Working Group [13] has a strong intention to speed-up the XML processing to reduce its size through a binary encoding.

1.2 Problem statement and contribution

While there exists a variety of solutions to access XML resources over the Web, there is – to our knowledge – no generic and unified solution to conveniently access all of:

1. The **current revision** of the XML resource or any subset thereof;
2. The **full revision history** of the XML resource or any subset thereof;
3. The **full modification history** of the XML resource or any subset thereof.

Our approach exploits XML by tightly integrating it with REST. We want to put aside the antiquated view of XML as a simple data exchange and storage format and discover what it really is: a fine-grained tree of nodes, which evolves over time through user modifications. If we let ourselves to view XML as a growing tree of nodes, we realise that we can access single nodes or whole sub-trees, i.e., XML fragments, within a temporal dimension in a unified, scalable and robust way.

We want to query the XML the way it was stored at any past point in time. Note that a point-in-time references a revision, i.e., a state of some resource. In addition and in stark contrast to all widely used interfaces and protocols, we want to randomly query for user modifications between any two past points in time. Only if we consider the whole life cycle of an XML resource including the past revisions and the (transaction-based) modification history, we will get a complete idea of its true power. We suggest Temporal REST as an interface with its related protocol message exchanges to generically implement our idea to exploit web-based XML resources. According to the Pareto principle [14], our proposal is simple enough for the average web application developer and at the same time it is extensible enough to be used with complex setups.

1.3 Related work

We identified three categories of related work. First, the systems without any temporal support. Second, the systems with support for access to past revisions. Third, the systems with support for access to past revisions and some kind of modification observation.

Almost all current file and database systems belong to the first category. Note that modern file and database systems actually do perform some kind of journaling or transaction logging to support crash recovery or transactional behavior but they only use it for internal purposes and do not provide a public interface.

The second category contains an increasing number of systems. However, all of these systems mainly suffer from the fact that the modification history has to be extracted on the application layer by comparing two different revisions. This extraction is based on expensive (binary) delta calculations such as Xdelta [15]. In addition, the deltas do not immediately reflect modifications on fine-granular tree-based data structures such as XML. Concurrent versions systems such as CVS [16], Mercurial [17], and WebDAV-based [18] Delta-V [19] serve as good examples for the second category. All three systems provide access to the current and all past revisions of XML (and other) resources but they do not work at the fine granularity of XML and they do not provide an interface to query the modification history of the XML at its natural granularity, i.e., the node or sub-tree level. In the file system world, Hammer [20] and ZFS [21] are recent additions. While

Hammer supports access to all past revisions, ZFS only keeps a subset of the past revisions, i.e., user-demanded snapshots.

The third category is the youngest and smallest one. Systems in the third category allow to stream modification events to other systems for backup or other purposes. They do not aim at providing random access to any past modification as we do with Temporal REST. Apple has only recently started to make the local file- and directory-level modifications visible to the applications through FSEvents [22]. Apple's Time Machine [23] is an excellent example application to consume these events to perform an incremental backup. Note that FSEvents does only remember that a given file or directory changed. It does not remember the fine-granular changes of, say, the XML node tree stored within a file. The Content Repository API for Java Technology Specification [24] comes closest to our idea as it optionally supports the concepts of versioning, activities, and observation. Versioning works similar to concurrent versions systems. Activities group modifications and make them accessible for later re-use. Observation encapsulates modifications into events to propagate them to all interested parties. However, all three concepts are kept separately and optional – a fact which complicates the every-day use.

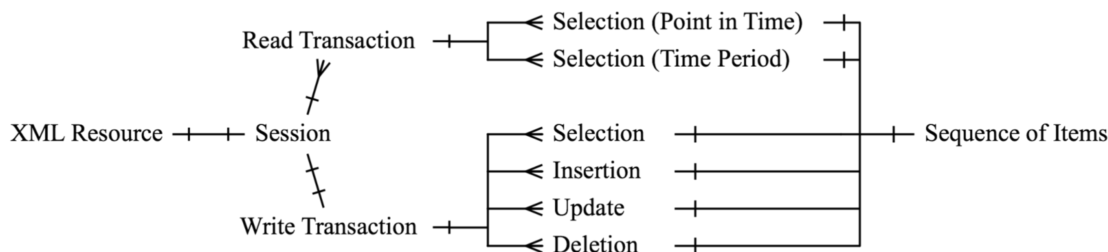
The rest of this paper is organised as follows. Section 2 describes the data model beneath Temporal REST. Section 3 provides a case study to clarify the intended use. Finally, Section 4 concludes our work and motivates future work.

2. DATA MODEL

2.1 Session- and transaction-based access

Our data model encapsulates each web-based XML resource within a single session, i.e., each session is responsible to coordinate the access to a single XML document together with its related revision and modification history. A session allows multiple concurrent read and one single write transaction at any time. Read transactions can access all past modifications and revisions of the XML document up to the last successfully committed revision. A write transaction creates a new revision of the XML document upon commit or drops all changes upon abort. Starting with one, each revision is assigned a positive number in increasing order. In addition, each revision is at least tagged with a time stamp, an author, and a commit comment. Supplementary meta-data can be added as required. While a read transaction only allows select operations, the write transaction additionally allows insert, update, and delete operations. A read transaction can sequentially execute multiple select operations. The write transaction can sequentially execute any operation until the write transaction is either committed or aborted. The isolation is clearly given with this model, i.e., only the single write transaction will see dirty data. The response of an operation always consists of a sequence of items as described with the data model of XPath 2.0 and XQuery 1.0. The relationship of the involved entities is shown in Figure 1.

Figure 1. XML resource access. The relationships are depicted according to Barker's notation [25].



For our initial version of Temporal REST, we have chosen to only support a single write transaction for each session at any time. While this might look too restrictive or cause bottlenecks due to the serialisation of concurrently issued write operations, we favor simplicity from both the usability and implementation perspective. Other widely deployed systems such as the full-text framework Lucene [26] and the file system

ZFS also support only a single write transaction and they are still perfectly useful for real-world use-cases. If the serialisation of multiple concurrently issued write transactions still causes a bottleneck, a XML resource can be split up into smaller ones, i.e., to allow more independent write transactions. Finally, an application can implement a sophisticated access and locking model, which includes two or three phase commits spanning multiple sessions. As undo and redo operations are inherently supported by Temporal REST (the past revisions are always available), an already committed revision can quickly be reverted to a past one.

Security with respect to confidentiality and integrity is cared for – if required – with the transport layer security protocol on top of which HTTP works. Authentication and authorisation are handled with the readily available mechanisms of HTTP. A clear separation of concerns tremendously facilitates the specification and implementation of Temporal REST.

2.2 Identification of XML fragments

There are two fundamental ways to access nodes and sub-trees, i.e., XML fragments, within an XML resource. First, the traditional axis-navigation or query-based access. Second, the ID-based random access. Temporal REST supports both and complements them with a temporal expression as described later.

XML IDs enable the user to tag the XML document and to quickly access the XML fragment by providing this XML ID. However, most XML nodes are not tagged with such a XML ID and remain inaccessible from the XML ID perspective. We suggest the tagging of at least all element nodes with a system-generated REST ID. Text nodes or attributes are accessible through their parent node. Other XML nodes such as comments or processing instructions may be tagged by the system on demand. One advantage of having the system to do the REST ID assignment is that the REST ID remains stable throughout revisions and modifications, i.e., a node or its modifications can be accessed irrespective of the revision or position in the tree. Another advantage is the guarantee of the existence of an ID. The system can make the REST IDs visible by tagging the serialised XML with REST ID attributes bound to the namespace of Temporal REST. Consequently, the user may choose to use this information to quickly access the nodes later on in an ID-based random-access fashion.

Each insertion operation assigns unique immutable REST IDs to all new element nodes. This assignment is made by the back-end that stores the XML and does not affect any existing user-assigned XML ID. REST IDs are numerical and they are incrementally assigned starting at one. The document root has the REST ID zero. REST IDs do not necessarily need to be assigned in document order and they must not change once assigned to a node. In addition, we suggest not reusing REST IDs. This reduces the confusion due to reassignments in future revisions. Since deletions are less frequent than insertions with most real-world workloads [27], the loss of number space is considered to be negligible. Figure 2 shows the assignment of REST IDs.

Figure 2. Assignment of REST IDs. Any XML fragment or document can be depicted as an unranked ordered tree. The REST ID makes sure that every element node gets its own unique immutable identifier. ‘<’ and ‘>’ denote element nodes and ‘#’ denotes text nodes in the node tree. A simple XML resource storing a document serves as an example.

2.3 XML fragment modification events

Each insertion, update, or deletion of a XML node results in a modification event. Each write transaction commit groups the modification events into one revision and assigns a timestamp, an author, and a comment to the whole revision. Temporal REST communicates modifications by encapsulating the modified node within an item element. The item element contains the REST ID of the modified node as well as revision, time stamp, author, and comment information. As such, both the insertion and the deletion can be considered as a setting a node to a new value. Deletion sets the node to the empty node.

We opted for this approach for two reasons. First, we can streamline the transport of XML fragments and modifications within the original XPath 2.0 and XQuery 1.0 data model, i.e., within a sequence of items. Second, the back-end can combine the storage of the modification event and the result of the modification. Section 3 will show, how this is achieved in practice.

Read transactions can select XML fragments in any revision. In addition, read transactions can select the XML fragment modifications, which took place between any two revisions. Since the revision number may not be convenient enough, the system must support the resolution of a time stamp into the closest revision number.

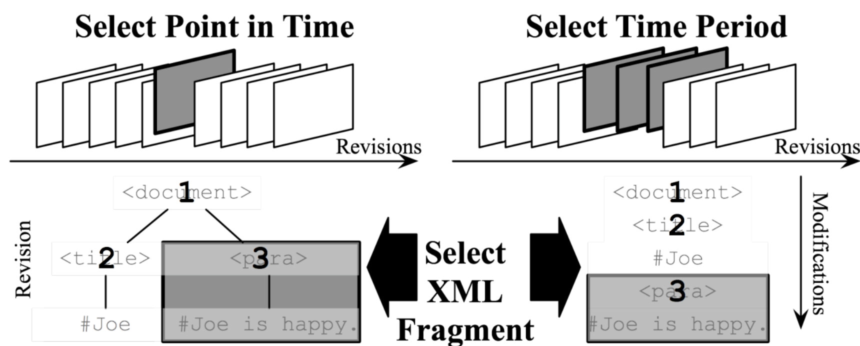
2.4 Operations on XML resources

2.4.1 Select

The select operation allows the retrieval of a sequence of items as defined with the XPath 2.0 and XQuery 1.0 data model. Each item is an atomic value, a XML node, or now also a modification event. The selection can be query-based, i.e., an XPath 2.0 or XQuery 1.0 expression, or REST ID-based. Temporal REST will restrict the execution domain of both the query and the REST ID according to the temporal expression either selecting a point in time or a time period (see Figure 3). While a query may return a sequence of multiple items, an access solely based on a REST ID will return a sequence with at most one item. If the query and REST ID approach are combined together, the query treats the node with the given REST ID as the root node of the query. The query-based approach allows to add new query languages in the future and to express complex queries including operations such as full-text search or joins. The REST ID-based approach allows to directly select an item with optimal performance since the system does not have to compile and optimise the query.

The temporal expression must be enclosed with round brackets '`(`' and '`)`' and contain a single point in time or a time period consisting of two points in time separated by a dash '`-`'. A point in time can be a revision number, an ISO date in short notation, i.e., without dashes or colons, or nothing, i.e., the last successfully committed revision. A single point in time will retrieve the XML fragments as they looked like at the given revision. The time period will retrieve the modifications between (and including) the two provided points in time in the according order. Leaving away the temporal expression automatically causes a fallback to the last successfully committed revision for backwards compatibility.

Figure 3. Selection. The left side shows the selection of an XML fragment, as it was stored at a given point in time. The right side shows the selection of the modifications on a XML fragment during a given time period.



2.4.2 Insert

A single node or a whole sub-tree can be inserted either as the first child of an existing node or as its right sibling. As such, the insert operation requires a query selecting a number of nodes or a REST ID besides the actual XML fragment to insert. During the insertion process, the back-end system will assign the REST IDs as described above. Note that the insertion of an attribute must be made with the PUT operation changing the whole node.

2.4.3 Update

A single node can be replaced with or without the replacement of its sub-tree. Again the updating operation requires a query selecting a number of nodes to update or a REST ID besides the actual XML fragment to represent the updated node or sub-tree. Restricting the effect of the update to the node (not effecting its sub-tree), allows the insertion of an attribute into an existing node without changing its whole sub-tree.

2.4.4 Delete

Whenever a node is deleted, the node and its sub-tree are purged from the system (but not from the past revisions). The deletion operation requires a query or a REST ID to select the nodes to delete.

2.5 Serialisation of XML fragments

The default behavior of the original XML data model is to serialise the whole sub-tree of a XML node returned as an item. From both the practical and safety perspective, it may be reasonable not always to return the whole sub-tree but to limit its global depth and the per-node fan-out. Whenever a serialisation truncates a XML fragment due to depth or fan-out limitations, it will tag the last serialised node with the depth or fan-out limitation to inform the user and allow to retrieve the missing nodes with a consecutive request. In addition, every element node is tagged with its REST ID. If the sequence itself contains too many items, the sequence can be paged.

3. CASE STUDY

3.1 Five main use cases

Collaborative document authoring serves as a perfect case study. Let us assume a workflow that specifies the role of the person, the activity, and the exact time this activity has to be performed during the publication process. Having different stages, the workflow involves multiple people who take on different roles such as author or reviewer who perform tasks sequentially or concurrently. If the underlying document is stored as XML, e.g., in OpenDocument [28] or DocBook [29] format, then the application layer can conveniently provide temporal functionality. At any time, the author or reviewer can effortlessly observe who has done what since the author or reviewer last looked at the document. The Temporal REST interface allows to quickly visualise the modification history or to swiftly create individual Atom news feeds for the involved people, i.e., to transform the response with XSLT into valid Atom XML. While the application still has to model and implement the workflow – a task, which is an art of its own – it is greatly simplified because it does not have to consider the design, interface, message exchange, and implementation of a specific temporal functionality: it can solely rely on Temporal REST.

The document we are working on will see a sequence of modifications as described with Table 1. Table 2 lists the HTTP request and response pairs to perform the modifications listed in Table 1 (Rows 1 to 3) alongside with a query selecting a point in time (Row 4) and a query selecting a time period (Row 5).

Row 1 of Table 2 shows the initial import of an XML document into the repository of XML resources. As a reaction to this HTTP POST request, the server-side session initiates a write transaction, inserts the XML fragment given in the request body, tags all inserted element nodes with REST IDs, commits if no error was

encountered, and responds with a sequence bound to the committed revision, i.e., 1, and containing a single item, i.e., the inserted XML fragment.

Table 1. Sequence of modifications on example document (see Figure 2 and 3).

	User intention	Required modifications	Resulting revision
1	Add title ‘Joe’ and paragraph ‘Joe is happy.’ to document	REST ID 1: Insert `<document>` as first child of REST ID 0 REST ID 2: Insert `<title>Joe</title>` as first child of REST ID 1 REST ID 3: Insert `<para>Joe is happy.</para>` as right sibling of REST ID 2	<?xml version='1'?> <document> <title>Joe</title> <para> Joe is happy. </para> </document>
2	Rewrite paragraph to ‘Mike is happy.’	Update REST ID 3 to <para>Mike is happy.</para>	<?xml version='1'?> <document> <title>Joe</title> <para> Mike is happy. </para> </document>
3	Remove title	Delete REST ID 2	<?xml version='1'?> <document> <para> Mike is happy. </para> </document>

Table 2. HTTP request and response pair examples for five main use cases.

	HTTP Request	HTTP Response
1	POST http://../document <?xml version='1'?> <document> <title>Joe</title> <para> Joe is happy. </para> </document>	<?xml version='1'?> **<rest:response xmlns:rest='REST'>** **<rest:sequence rest:revision='1'>** **<rest:item>** <document **rest:id='1'**> <title **rest:id='2'**>Joe</title> <para **rest:id='3'**> Joe is happy. </para> </document> **</rest:item>** **</rest:sequence>** **</rest:response>**
2	PUT http://../document/3 <para> Mike is happy. </para>	<?xml version='1'?> **<rest:response xmlns:rest='REST'>** **<rest:sequence rest:revision='2'>** **<rest:item>** <para **rest:id='3'**> Mike is happy. </para> **</rest:item>** **</rest:sequence>** **</rest:response>**
3	DELETE http://../document/2	<?xml version='1'?> **<rest:response xmlns:rest='REST'>** **<rest:sequence rest:revision='3'>** **<rest:item rest:id='2'/>** **</rest:sequence>** **</rest:response>**
4	GET http://../document/(1)?//para/text()	<?xml version='1'?> **<rest:response xmlns:rest='REST'>**

		<rest:sequence rest:revision='1'> <rest:item> Joe is happy. </rest:item> </rest:sequence> </rest:response>
5	GET http://../document/(2-3)	<?xml version='1'?> <rest:response xmlns:rest='REST'> <rest:sequence> <rest:item rest:revision='2'> <para rest:id='3'> Mike is happy. </para> </rest:item> <rest:item rest:revision='3' rest:item='2'/> </rest:sequence> </rest:response>

Row 2 of Table 2 replaces the XML fragment rooted at node with REST ID 3. Again, the server-side session initiates a write transaction, overwrites the existing XML fragment with the XML fragment of the request body, tags all new element nodes with REST IDs, commits if no error was encountered, and responds with a sequence bound to the committed revision, i.e., 2, and containing a single item, i.e., the updated XML fragment.

Row 3 of Table 2 removes the XML fragment rooted at node 2. The server-side session initiates a write transaction, removes the requested XML fragment, commits if no error was encountered, and responds with a sequence bound to the committed revision, i.e., 3, and containing a single empty item to mark the deletion. Note how the REST ID propagates to the item node because the item does not contain any node anymore.

Row 4 of Table 2 shows a query for a given point in time, i.e., revision 1. Here, we show an XPath 2.0 expression restricting the result to a sequence of items, each containing the text of a paragraph node. The server-side session initiates a read transaction bound to the given revision, compiles and executes the XPath 2.0 expression, and returns the result.

Row 5 of Table 2 shows a query for a time period, i.e., all modifications which took place between revision 2 and 3 (inclusive). The server-side session initiates a read transaction bound to the newer revision and retrieves all modifications between the newer revision and the older revision. As there is no further restriction, the modifications on any node of the XML resource are retrieved. Note that the two returned items are almost equivalent to the items retrieved with Rows 2 and 3. The only difference is the revision number included with the item instead of the sequence because each item may be bound to another revision.

3.2 Preliminary observations

While we have good experiences with the five main use cases, there are many others open for discussion. First of all, there are many variations on how to express a modification. E.g., an insertion of a XML fragment as the first child of some node can be expressed as a change of this (parent) node. Second, the complexity increases if not only element but also text nodes, attributes, or other nodes are tagged with a REST ID. This is not due to the load on the system, which does not change, but mainly due to the fact that text and attribute nodes must be surrounded with auxiliary metadata element nodes if modified directly and if they are not contained within a surrounding element node.

We experimented with grouping several HTTP requests into one, i.e., group several operations such as select or insert into one HTTP request. As a consequence, the request URL loses its expressiveness because it cannot transmit any information any more as this might have to be shared with all contained operations. The whole request metadata must be packed into the HTTP body. This makes it necessary to express the session context and the read transaction or write transaction boundaries within the request body. If just a single request is issued, this can be implicitly encoded in with the HTTP command and URL. A clear advantage of request grouping is the fact that several modification operations can be executed within a single transaction.

In addition, more metadata can be encoded into the request body than into the request URL. E.g., it is not convenient to encode a complex XQuery expression into the URL.

Besides theoretical reasoning, we implemented a preliminary prototype to back our estimates about performance and space requirements. We found that our simple prototype of a temporal native XML database showed performance in the same order of magnitude (approx. 30% difference) to SAX when retrieving the whole XML resource in any given revision. The same holds for XPath 2.0 expression evaluation. As soon as it comes to REST ID-based random access, our prototype clearly out-performed SAX. Note that these performance results are common when one compares any other existing native XML database with SAX. The differences start to show up when the modifications are queried. While this is only possible with our prototype, all others fail due to missing functionality. We can stream the modifications at one half of the performance of streaming a revision. In addition, our prototype is able to shrink the first revision to about one half of the size of the original XML file. Each write transaction commit then roughly adds a few kB of data, depending on the number of modified nodes. For single node modifications it can be as low as a few hundred bytes. For many nodes, it is roughly one half of the original XML fragment size.

For implementations based on (object-) relational databases, it is important to agree on a generic mapping between XML files and relational tables. This is necessary to guarantee the consistency of the interface irrespective of the underlying back-end implementation, i.e., a native XML or a relational database. As long as the data schema can be mapped to relational tables conforming to the relational normal forms, this is not a problem and actually the case for the vast majority of XML files or relational database schemas.

4. CONCLUSION

Temporal REST is a new paradigm on how to exploit web-based XML resources. Instead of solely thinking about XML as a unified resource exchange and a storage format, we promote the idea of looking at XML as a growing tree of nodes. We want to provide a generic and unified solution to conveniently access all of:

1. The **current revision** of the XML resource or any subset thereof;
2. The **full revision history** of the XML resource or any subset thereof;
3. The **full modification history** of the XML resource or any subset thereof.

We see potential applications, e.g., in the area of personal information management, collaborative document authoring, content management, or geographic visual analytics. The interdisciplinary character emerging from the fact that different sciences and businesses will develop applications on top of Temporal REST makes it especially attractive. All above-mentioned applications currently use XML and some kind of web-based interaction. The major advantage of Temporal REST lies in its expressive and convenient interface vastly reducing the design and implementation complexity formerly faced with each new application. While Temporal REST facilitates the look into the past by technical means, it will remain for every application, their users, national or international law, and, as such, our society to decide when to eventually erase past revisions. The trade-off between archiving, usability, and privacy is likely to cause enthralling discussions.

Our next three steps will comprise the implementation and further elaboration. First, we will implement two back-ends for Temporal REST; one based on an existing relational database, one based on a native XML database. Second, we will implement a Web 2.0 application for collaborative document authoring. Third, from our practical findings and input from the web application development and services community, we will release a second draft of Temporal REST detailing all use cases. In the course of the third step, we will also investigate, how Temporal REST can be integrated with existing protocols such as SOAP.

We invite the web application development and services community as well as (object-) relational and native XML database implementers to scrutinise Temporal REST, implement prototypes, and contribute new use cases and practical findings. We strongly believe in the worthiness of our idea will promote our idea towards a Request For Comment. If Web 2.0 is the web for social and collaborative interaction, Web 3.0 may become the temporal web, i.e., a global time machine.

ACKNOWLEDGEMENT

We would like to thank Prof. Dr. Marcel Waldvogel, Stephan Pietzko, and Thierry Kramis for their valuable input on Temporal REST.

REFERENCES

[1] R. T. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D University of California Irvine

[2] Object Management Group, Inc. 2004. *Common Object Request Broker Architecture: Core Specification*. http://www.omg.org/docs/formal/04-03-12.pdf

[3] Microsoft Developer Network. 1996. *DCOM Technical Overview*. http://msdn.microsoft.com/en-us/library/ms809340.aspx

[4] D. Heinemeier Hansson. *Web development that doesn't hurt*. http://www.rubyonrails.com/

[5] Noelios Consulting. *Lightweight REST framework for Java*. http://www.restlet.org

[6] Microsoft Corporation. 2008. *Project Astoria Team Blog*. http://blogs.msdn.com/astoriateam/

[7] IBM et al. 2007. *Business Process Execution Language for Web Services version 1.1*. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[8] M. Nottingham; R. Sayre. December 2005. *The Atom Syndication Format*. RFC 4287

[9] Microsoft. 2008. *Microsoft SQL Server*. http://www.microsoft.com/sqlserver/2008/en/us/default.aspx

[10] Oracle. *Get Better Results With Oracle*. http://www.oracle.com/index.html

[11] IBM. *DB2 Product Family*. http://www-306.ibm.com/software/data/db2/

[12] X-Hive Corporation. *Improve operational performance by managing your complex documentation with X-Hive*. http://www.x-hive.com/

[13] W3C. 2008. *Efficient XML Enterchange Working Group*. http://www.w3.org/XML/EXI/

[14] Pinnicle Management Associates, LLC. *Pareto Principle*. http://www.pinnicle.com/Articles/Pareto_Principle/pareto_principle.html

[15] J. MacDonald. 2008. *Xdelta*. http://xdelta.org

[16] D. Grune. 1986. *Concurrent Versions System, a method for independent cooperation*. IR 113, Vrije Universiteit, Amsterdam, pp. 9

[17] M. Mackall. 2006. *Towards a better SCM: Revlog and Mercurial*. Ottawa Linux Symposium

[18] Y. Goland et al. 1999. *HTTP Extensions for Distributed Authoring – WEBDAV*. RFC 2518.

[19] G. Clemm et al. 2002. *Versioning Extensions to WebDAV*. RFC 3253.

[20] M. Dillon. 2008. *The Hammer Filesystem*. http://www.dragonflybsd.org/hammer/hammer.pdf

[21] Sun Microsystems, Inc. *ZFS: the last word in file systems*. http://www.sun.com/2004-0914/feature/

[22] ArsTechnica. 2007. *FSEvents*. http://arstechnica.com/reviews/os/mac-os-x-5.ars/7#fsevents

[23] Apple Inc. Time Machine. *A giant leap backward*. http://www.apple.com/macosx/features/timemachine.html

[24] Java Community Process. 2007. *The Content Repository API for Java Technology Specification 2.0 (JSR 283)*. http://jcp.org/en/jsr/detail?id=283

[25] R. Barker. 1990. *CASE Method: Entity Relationship Modelling*. Reading, MA: Addison-Wesley Professional.

[26] Apache Software Foundation. *Lucene*. http://lucene.apache.org/

[27] T. Böhme; E. Rahm. 2001. *XMach-1: A Benchmark for XML Data Management*. 9. GI-Fachtagung, Springer Verlag, p 264-273.

[28] ODF Alliance. *Open Document Format*. http://www.odfalliance.org

[29] OASIS. *DocBook Technical Committee Document Repository*. http://www.oasis-open.org/docbook/