Orestes: a REST protocol for horizontally scalable cloud database access

Felix Gessert[1]
Master Student
7gessert@informatik.uni-hamburg.de

Florian Bücklers[1]
Master Student
7bueckle@informatik.uni-hamburg.de

Norbert Ritter[1]
Professor
ritter@informatik.uni-hamburg.de

[1]Computer Science Department, University of Hamburg
Vogt-Kölln Straße 33, 22527 Hamburg, Germany

ABSTRACT
In our poster, we are going to describe a REST/HTTP database access protocol for object-oriented persistence that allows read requests to scale horizontally. Scalability is achieved by using Web Proxy Caches to serve database objects, while keeping strong ACID transaction semantics through optimistic verification mechanisms. We call this protocol *Orestes* (*O*bjects *REST*fully *e*ncapsulated in *s*tandard formats) and validate the approach by comparative benchmarks for a cloud computing scenario.

1. INTRODUCTION
The emergence of cloud computing and NoSQL databases has demonstrated a clear demand for scalable database systems with cloud capable, web-based interfaces. With Orestes we try to reconcile the desirable system properties of low latency, transparent horizontal scalability for read requests, HTTP access based on standards and independence of database and client APIs. These four properties of database access are crucial for novel application architectures. In particular applications deployed in cloud environments mostly need to interact with their services (one of which is persistence) via stateless HTTP. Also databases offered as a service need means to scale the read demands of the clients and to provide low-latency access to mitigate the effect of possible geographic application/database distribution. Our approach does not inherently scale write loads, since this aspect of scalability is left to the database that processes these writes. Horizontal read scalability on the other hand is provided by our approach on the protocol level through web caching.

Orestes is designed for object-oriented persistence and strong consistency and can therefore be backed by APIs like JDO (Java Data Objects) or JPA (Java Persistence API) on the client side and object oriented databases or relational databases with an object-relational-mapping layer on the server side. The scalable REST/HTTP protocol introduced by Orestes is therefore independent of the client side programming language and the database backend, which can be combined arbitrarily.

2. RELATED WORK
The core of Orestes is its RESTful HTTP protocol. The REST (Representational State Transfer) architectural style was introduced by Fielding [2] and is now widely regarded as a modern way of exposing services with appropriate characteristics and a clear structure. Database caching is a well-researched topic [3] with many different placement and implementation options. To our best knowledge the combination of the web caching model with database access has not yet been researched or implemented and hence is a new strategy to make databases profit from web technologies. Web-caching is the scalability model implemented by the HTTP protocol [1]. We intend to show that the same scalability model that allows web sites to scale can also scale databases.

3. CONCEPT
With Orestes we aim to achieve four key properties. Table 1 gives a summary of the mechanisms leveraged to obtain them. In this section we describe how these mechanisms work, whereas the next sections describe how they are implemented and evaluated.

Table 1. Properties achievable by Orestes

Property	Mechanism
Read scalability	Caching of database objects
Low latency	Cache deployment in client network
Loose coupling of client API and database system	Generic HTTP access protocol and resource structure including schema, transactions, querys and objects
Standard formats	Extensible HTTP content negotiation and definition of JSON formats

3.1 Protocol
The client interacting with the database uses some object oriented API (e.g. JDO) to retrieve, query and write objects in the database. Orestes is the network layer gluing both together. The HTTP interface is defined by resources with URLs identifying the database abstractions *querys*, *transactions*, *objects*, *schema* and *settings*. Each object has a schema defining its structure, a version number and a URL used to interact with it through HTTP methods, namely GET (retrieve), PUT (create/update), POST (update) and DELETE.

The typical access pattern observed in object oriented persistence is navigation: a client retrieves objects, loads referenced objects or retrieves their collections. Each step of dereferencing normally results in a network round trip. Web-caching reduces the impact of the network access.

The HTTP web-caching model allows objects to have an assigned lifetime for which they are regarded as fresh. It also provides a revalidation mechanism, which can be used to confirm the freshness of an object by asking the origin sever based on a version number (called ETag) or a

modification date. The key difficulty is the fact, that the database (the origin server) cannot invalidate objects stored in web-caches upon updates, unless the web-cache is deployed as a so called reverse proxy cache in the database network. Assigning a static caching lifetime to database objects therefore introduces the possibility of stale reads: two caches store the same object, one client changes the object, another client loads a stale version of the same object using the other cache. To deal with stale reads in Orestes, the database uses optimistic concurrency control, for instance optimistic locking. Orestes utilizes optimistic transactions since objects cached out of the control of the database system can neither be directly locked nor invalidated. Using the strengths of optimistic transactions, Orestes thus mainly addresses read-intensive applications.

The formats used to represent objects, queries, etc. are interchangeable. If for instance the database supports SQL queries, the client can issue the query in an appropriate standard media type. We define a default JSON format for all representations in the Orestes protocol. By following the REST/HTTP protocol a loose coupling of the client side persistence API and the database system is achieved – both can be combined in any combination since the middle tier is the common Orestes protocol.

3.2 Example

In this section we briefly describe how a client can access database objects and where web-caches come into play. The URL of each object consists of three parts. The domain of the server is followed by */db/* where all dynamic data of the database are located. The second part indicates the type of the object. A type is represented by a namespace and a class name. The third part represents the ID of the object. If a client intends to load an instance of a Person class with a given ID *3*, it performs an HTTP GET request for the *http://example.com/db/simulation.classes/Person/3* URL. This HTTP request will be routed through the network to the example.com server. On its way to the server, the request may pass one or more web-caches. Each of these web-caches will check its local cache for a previously received, cachable response for that request (i.e. the object). If resident in the cache, the object is directly returned without forwarding the request to the origin server. Otherwise the origin server responds to the forwarded request. The client receives a JSON document that includes all metadata (like the version number) and fields of the loaded object.

4. IMPLEMENTATION

We developed an implementation of the Orestes network layer in Java, which is easily extensible for new formats, new persistence APIs and new database systems. We created a server-side binding for two object oriented (OO) database systems, the Versant Object Database (VOD) and db4o, as well as client-side binding for JDO and a JPA port for JavaScript. The generic Java Orestes layer also contains a web interface that can be used to browse the database via the REST resource structure. We plan to use this browser interface as a practical demonstration[1] by bringing a laptop allowing interested visitors to explore a running local database instance through Orestes.

[1]No particular, further requirements

5. RESULTS

To measure the performance gain of the Orestes protocol and our implementation, we designed a benchmark scenario. The benchmark comprises a complex object model for a social networking scenario, using OO concepts like aggregation, association, generalization, etc. The configurable transactional JDO benchmark client performs a navigating access pattern by serially and randomly loading objects stored in the database (either drawn from a uniform or Zipf distribution) and writing others.

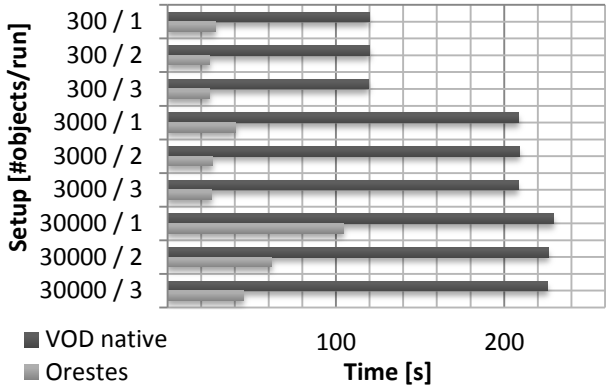

Figure 1. Average runtime of 50 Clients simultaneously loading 450 uniformly and randomly chosen objects out of a given number of objects (300, 3000, 30000) in 3 runs

Figure 1 shows the results of one of the benchmark cases we conducted: 50 client machines deployed in the Amazon EC2 cloud in Europe accessing a Versant database located in California through a web-cache in Europe. The clients perform the benchmark simultaneously, reading 450 and writing 50 objects in three consecutive runs. For the sake of simplicity we compare the average runtimes of the complete benchmark for JDO over Orestes and JDO with the native VOD TCP protocol (which uses local client caching). Even these abridged results unmistakably show the ability of Orestes to scale reads, reduce latency and disburden the database.

6. FUTURE WORK

Stale objects potentially have the price of transaction aborts. We are therefore working on the combination of an invalidation service, integration with content delivery networks and an algorithm that guarantees freshness of objects using Bloom filters for compact write-log representation.

7. REFERENCES

[1] Barish, G. and Obraczke, K. 2000. World wide web caching: Trends and techniques. *Communications Magazine, IEEE*. 38, 5 (2000), 178–184.

[2] Fielding, R.T. 2000. Architectural styles and the design of network-based software architectures. University of California.

[3] Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B.G. und Naughton, J.F. 2002. Middle-tier database caching for e-business. *Proceedings of the ACM SIGMOD* (2002), 600–611.