Replication Mechanisms for a Distributed Time Series Storage and Retrieval Service

Mugurel Ionut Andreica, Iosif Charles Legrand, Ramiro Voicu

Replication Mechanisms for a Distributed Time Series Storage and Retrieval Service

Mugurel Ionuţ Andreica
Politehnica University of Bucharest
Splaiul Independenţei 313, sector 6,
Bucharest, Romania
mugurel.andreica@cs.pub.ro

Iosif Charles Legrand, Ramiro Voicu
California Institute of Technology
1200 East California Boulevard, Pasadena,
California, USA
{iosif.legrand, ramiro.voicu}@cern.ch

ABSTRACT
In this paper we present the prototype architecture of a distributed service which stores and retrieves time series data, together with replication mechanisms employed in order to provide both reliability and load balancing. The entries of each time series are stored locally on the machines running the instances of the service. Each entry is eventually fully replicated on every service instance. Our replication mechanisms depend on whether there is only one service instance receiving each entry of a time series from a client or there may be multiple such instances.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed databases.*

General Terms
Algorithms, Performance, Design, Reliability.

Keywords
Replication, Time series, Distributed service.

1. INTRODUCTION
Many data sets can be modeled as time series (i.e. in which every piece of data has an associated time stamp and their ordering according to time stamps is significant in some way), like monitoring data, packets sent on a communication channel, etc. Providing storage of and efficient access to such data sets is an important task. In this paper we present the architecture of a distributed service for fully replicated storage and retrieval of time series data, together with data replication mechanisms.

2. SERVICE ARCHITECTURE
Our time series storage and retrieval service consists of multiple service instances, running on multiple machines. Each instance runs independently of the other instances and provides the same functionality as all the others. Each instance stores the entries of each time series on its local hard-drive(s). The entries of each time series (data and index) are stored in one or more binary files (with random access capabilities). A file stores data related to a single time series (data or index entries). Each time series has a unique identifier (*tsid*). General information about each time series (identifier, set of file names and numbers, number of data entries, number of index entries at each level, current index entries at each indexing level, etc.) is stored by each instance in main memory (RAM). Periodically, the general information is check-pointed on the local hard drive(s), in order to be able to stop the instance and restart it later (possibly on a different machine), if necessary.

ICAC'11, June 14-18, 2011, Karlsruhe, Germany.
ACM 978-1-4503-0607-2/11/06.

Each entry of a time series consists of a pair *(timestamp, value)*. It is not required for the values of all the entries of a time series to have the same size (thus, the *value* field may be anything), but, in case the size of the *value* field may be variable, the size of the field must be encoded explicitly. On the other hand, all the timestamps have the same size. The (data and index) entries of a time series are numbered with consecutive numbers, starting from *0* (this number is the *position* of an entry). The positions of the data entries correspond to the order in which the entries are added to the time series (and, as we will see later, they may be different on different instances) and not to the order of their timestamps (the same holds for index entries). The numbering is relative to a time series and continues over multiple data or index files (each file stores entries numbered consecutively).

Each instance provides to the clients the same API, through which, at any time, a client may *create* a new time series, may *insert* one or more consecutive entries at the end of an existing time series (not necessarily in increasing order of timestamps) or may *query* a time series. Common time series queries are similarity search queries (e.g. [1, 3]). We currently consider only two simpler types of queries:
1) retrieve all the entries of a given time series whose timestamps are between t_1 and t_2 (inclusive), and
2) retrieve all the entries of a given time series between positions p_1 and p_2 (inclusive)

Both types of queries may take extra options, like:
a) return anything only if the number of entries between t_1 and t_2 (p_1 and p_2) is at least equal to *vmin*
b) return the entries sorted according to timestamp or position

Periodically, the instances synchronize their data between them (thus obtaining full data replication). The service instances are interconnected in a (peer-to-peer) service overlay and each instance interacts only with its overlay neighbors. The design of such an overlay is outside the scope of this paper. The prototype service was implemented in Java and we used Java RMI for communication between a client and a service instance and between two service instances.

3. REPLICATION MECHANISMS
Let's consider a time series *tsid*. Entries of this time series are generated periodically and added to the time series storage and retrieval service by a client. Let's assume that this client always uses the same service instance for adding new entries to the time

series. In this case, each service instance may employ a technique we call *prefix replication*. Periodically, each service instance S asks from its neighbors for summary information about all the time series they store (for now, we are interested only in the time series identifier *tsid* and *num_entries(X,tsid)*=the number of entries of the series *tsid* stored by the neighbor X). After receiving all the answers (ignoring answers which may have timed out), S creates locally all the time series whose identifiers were not known to S previously. Afterwards, S considers each time series *tsid*. Let *max_num_entries(tsid)=max{num_entries(X,tsid) | X is a neighbor of S}*. While *num_entries(S,tsid)< max_num_entries(tsid)*, S selects a neighbor X such that *num_entries(X)>num_entries(S)* and asks from it all the entries between the positions p_1=*num_entries(S,tsid)* and p_2=*min{num_entries(S,tsid)+K, max_num_entries(tsid)}*. Here, K is used as an upper bound for the maximum number of entries which are requested at the same time. All the entries received from the neighbor X (in ascending order of their position) are added by S at the end of its local copy of the time series *tsid* (also updating the appropriate indices). We leave open the fact whether this mechanism should consider the time series sequentially or may split them across multiple threads (and we also leave open the mechanism for splitting over multiple threads, in case this option is chosen). Note that because all the entries are inserted into the system through only one service instance *SI*, they will be propagated to the other instances in the same order in which they were inserted at *SI*. Because of this, it is correct to assume that, if *num_entries(S,tsid)<num_entries(X,tsid)*, then the first *num_entries(S,tsid)* entries of the local copy of *tsid* stored by X are identical to the ones of the local copy of S.

Let's assume now that a client may use multiple instances for inserting the entries of a time series *tsid* (and the same entry may even be inserted by the client at multiple instances). In this case, our previous assumption that any two service instances have common prefixes of the same time series fails. After asking each neighbor for summary information regarding all the time series, including the *tmax(*,*)* and *tmin(*,*)* values defined below), a service instance S will use a sliding window (which will slide circularly) $[t_1(S,tsid),t_2(S,tsid)]$ for each time series *tsid*. Let *tmax(S,tsid)*=the maximum timestamp of an entry of the time series *tsid* stored by S (or $-\infty$ if S currently stores no entry of *tsid*) and let *tnmax(S,tsid)= max{tmax(S,tsid), max{tmax(X,tsid) | X is a neighbor of S}}*. Let *tmin(S,tsid)* and *tnmin(S,tsid)* have the same meaning, except that we replace *maximum* by *minimum* (and *tmin(S,tsid)*=$+\infty$ if S currently stores no entry of *tsid*). If *tnmax(S,tsid)>tmax(S,tsid)* then we set the sliding window to $[t_1(S,tsid)=tnmax(S,tsid)-W,\ t_2(S,tsid)=tnmax(S,tsid)]$, where W is a predefined window size (in terms of timestamp intervals), which should ensure that not too many entries are located in such a time window; otherwise, we set $t_2(S,tsid)=t_1(S,tsid)$ and then $t_1(S,tsid)=t_2(S,tsid)-W$ (if we get $t_2(S,tsid)<tnmin(S,tsid)$ then we set $t_1(S,tsid)=tnmax(S,tsid)-W$ and $t_2(S,tsid)=tnmax(S,tsid)$).

First, S queries itself and obtains the set of all of the entries of the time series *tsid* with timestamps between $t_1(S,tsid)$ and $t_2(S,tsid)$ currently stored by S, sorted according to timestamp (denoted by $CSE(S,tsid,t_1(S,tsid),t_2(S,tsid))$). Let *num* be the number of such entries. Then, for each neighbor X, S asks the same query (for the time interval $[t_1(S,tsid),t_2(S,tsid)]$), except that it instructs the neighbor to return nothing if the neighbor's result does not have more than *num'* entries (*num'=0* if we care about correct full replication; *num'=num* if we want to save bandwidth). The returned entries (if any) are merged with $CSE(S,tsid,t_1(S,tsid),t_2(S,tsid))$. If new entries not stored by S are discovered, they are added at the end of S's local copy of *tsid* and inserted into $CSE(S,tsid,t_1(S,tsid),t_2(S,tsid))$. After updating the entries in the interval $[t_1(S,tsid),\ t_2(S,tsid)]$, we modify the endpoints, by decreasing them both by W (if $t_2(S,tsid)$ becomes smaller than *tnmin(S,tsid)* then we set $t_1(S,tsid)=tnmax(S,tsid)-W$ and $t_2(S,tsid)=tnmax(S,tsid)$). The way these operations are split over potentially multiple threads (or even the order in which they are performed by the same thread, e.g. one full time series at a time, or one interval from each series at a time) or over multiple replication periods (e.g. how many times we are allowed to slide the time window during one period) is not specified. Note that, unlike the prefix replication case, when, except for periodic summaries, neighbors were queried only when we were sure that S was missing some entries, this process must be performed continuously (as a newly added entry may have any timestamp). The above process may be optimized by forcing an interval of interest, i.e. stating that we only care for the entries whose timestamps are between *TA* and *TB* (by setting *tnmax(S,tsid)= min{tnmax(S,tsid),TB}* and *tnmin(S,tsid)=max{tnmin(S,tsid),TA}*).

Fig. 1 (a-b) illustrates the two multicast-pull-based [2] replication mechanisms described in this section. Solid and dashed arrows denote *2* different entries. Numbers on the arrows denote the replication (synchronization) round at which each service instance receives the corresponding entry. Links between instances denote overlay connections. We performed a prefix replication test with *3* service instances (*A* and *C* in Geneva, *B* in Amsterdam). A client (running on the same machine as *A*) continuously inserted entries (with real values) of one time series for *4* minutes using instance *A*. Instance *B* got its data from *A*, and *C* obtained its data from *A* and *B*. *A* was able to write data at a speed of approx. *600,000* entries/sec. *C* replicated data at about *250,000* entries/sec during the first *4* minutes and at about *300,000* entries/sec after no more entries were written to *A*. *B* was able to replicate data at a speed of approx. *90,000* entries/sec during the first *4* minutes and *110,000* entries/sec afterwards. We used *K=25,000*. The replication speed difference between *B* and *C* is caused by different *RTTs* to *A*.

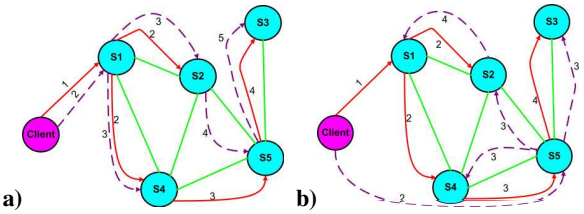

Figure 1. a) Prefix replication. b) General case replication.

4. ACKNOWLEDGEMENTS

The work presented in this paper has been partially supported by CNCS-UEFISCDI under research grants ID_1679/2008 (contract no. 736/2009) and PD_240/2010 (contract no. 33/28.07.2010).

5. REFERENCES

[1] Assent, I., Krieger, R., Afschari, F., and Seidl, T. 2008. The TS-tree: Efficient Time Series Search and Retrieval. In *Proc. 11th Intl. Conf. on Extending Database Tech.*, 252-263.

[2] Pruhs, K. and Uthaisombut, P. 2005. A Comparison of Multicast Pull Models. In *Algorithmica*, vol. 42, 289-307.

[3] Vlachos, M., Hadjieleftheriou, M., Gunopulos, D., and Keogh, E. 2003. Indexing Multi-dimensional Time-series with Support for Multiple Distance Measures. In: *Proc. 9th Intl. Conf. on Know. Discovery and Data Mining*, 216-225.