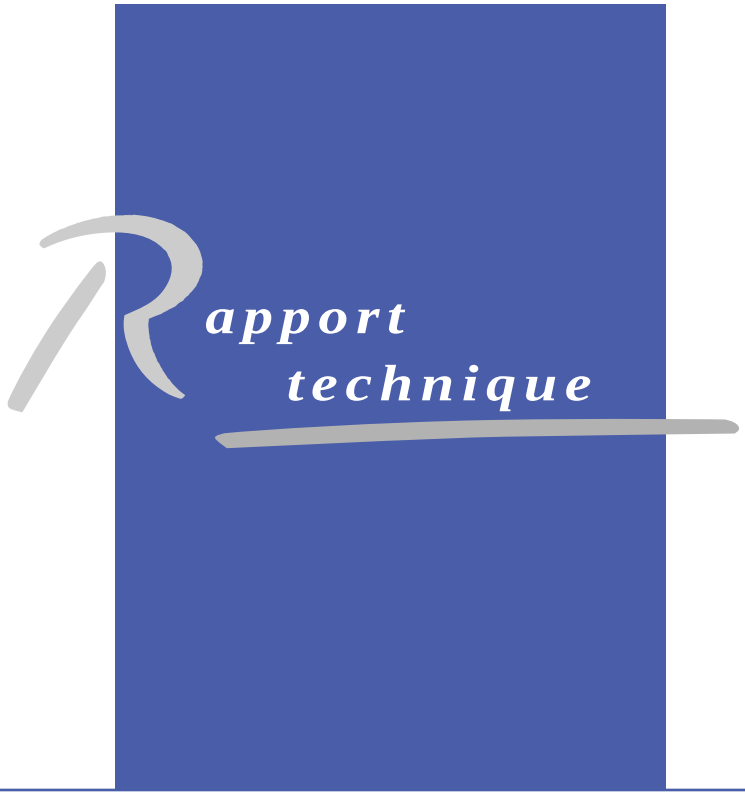
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

BlobSeer Monitoring Service

Jing Cai

N° 0368

Août 2009



ISSN 0249-0803 ISRN INRIA/RT--0368--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

BlobSeer Monitoring Service

Jing Cai *

Thème : Calcul distribué et applications à très haute performance
Équipe-Projet KerData

Rapport technique n° 0368 — Août 2009 — 19 pages

Abstract: This report introduces a monitoring service specially designed for BlobSeer, a large-scale distributed data-management platform. We describe how the large-scale distributed application is monitored and visualized using MonALISA and also reveal some technical details. A short tutorial for developers is enclosed.

Key-words: Distributed system, storage management, large-scale system, monitoring, visualization.

This work was performed at the **KerData** team of IRISA/INRIA while on a collaborative internship between INRIA and City University of Hong Kong.

* Department of Computer Science, City University of Hong Kong, China

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Un service de surveillance pour BlobSeer

Résumé : Ce rapport présente un service de surveillance spécialement conçu pour BlobSeer, une plate-forme de gestion de données réparties à large échelle. Nous décrivons comment cette application distribuée à large échelle est suivie et visualisée en utilisant MonALISA, ainsi que certains détails techniques. Un tutoriel pour les développeurs est joint en annexe.

Mots-clés : Systèmes repartis, gestion des données, système à large échelle, surveillance, visualisation.

Contents

BlobSeer[2], a large-scale data-sharing system, aims to efficiently manage the storage of large and unstructured binary data blocks which are referred to as BLOBs. This report introduces the basic concept of monitoring BlobSeer and how it could be done with the help of the MonALISA[5] monitoring framework.

1 Background Knowledge

1.1 BlobSeer

BlobSeer is a data-sharing system that manages the storage of large and unstructured data blocks called *binary large objects*, referred to as BLOBs further in this report. The blobs are striped into small chunks that have the same size, called *pages*.

BlobSeer addresses the problem of efficiently storing massive blobs in large-scale distributed environments. It provides an efficient fine-grained access to the pages belonging to each blob, as well as the possibility to modify them, in a distributed, multi-user environment.

One of the entities involved in the architecture of BlobSeer is the *client*, which initiates all blob operations: CREATE, READ, WRITE and APPEND. There can be many concurrent clients accessing the same blob or different blobs in the same time. The support for concurrent operations is enhanced by storing the pages belonging to the same blob on multiple *storage providers*.

The system has to keep track of the pages distribution across providers. Therefore, it associates some metadata to each blob. For each blob, the metadata is organized as a *distributed segment tree* [6], where each node corresponds to a version and to a page range within that version. Each leaf covers just one page, recording the information about the data provider where the page is physically stored. The metadata trees are stored on the *metadata providers*, which are processes organized as a *Distributed Hash Table*.

BlobSeer provides versioning support, so as to prevent pages from being overwritten and to be able to handle highly concurrent WRITE and APPEND operations. For each of them, only a patch composed of the range of written pages is added to the system, and a new metadata tree is created. The new metadata tree corresponds to a new version and points to the newly added pages and to the pages from the previous versions that were not overlapped by the added page range.

The system comprises two more entities: the *version manager* that deals with the serialization of the concurrent WRITE/APPEND requests and with the assignment of version numbers for each new WRITE/APPEND operation; the *provider manager*, the one that keeps track of all the storage providers in the system.

As far as this paper is concerned, an APPEND operation is only a special case of WRITE. Therefore, we disregard this aspect in the rest of the paper. Everything stated about WRITEs is also true for APPENDs, unless explicitly specified.

A typical setting of the BlobSeer system involves the deployment of a few hundreds of provider nodes, each of them storing data in the order of GB, and even tens of GB in the case of the use of the disk storage for each node. This implies that sizes within the order of TB can be easily reached for the blobs

stored in the system. Furthermore, the typical size for a page within a blob can be smaller that 1 MB, whence the need to deal with hundreds of thousands of pages belonging to just one blob.

1.2 MonALISA

BlobSeer is a storage system that deals with massive data, which are striped into a huge number of pages scattered across numerous storage providers. A monitoring tool tuned for presenting the state of a system like BlobSeer has to cope with two major challenges. On one side, it has to accommodate the immense number of pages that the system comprises once it stores several blobs. On the other side, the monitoring system has to be able to deal with a huge amount of monitoring information generated when an application accesses the nodes that make up the storage service. It is the case when multiple clients simultaneously access various parts of the stored blobs, as they generate a piece of monitoring information for each page accessed on each provider. MonALISA is suitable for this task, as it is a system designed to run in grid environments and it proved to be a scalable and reliable system.

The MonALISA (*Monitoring Agents in a Large Integrated Services Architecture*) [5] system is a JINI-based [4], scalable framework of distributed services, which provides the necessary tools for collecting and processing monitoring information.

Its architecture is based on four layers of services. It complies with the Grid Monitoring Architecture (GMA) [3] proposed by the Global Grid Forum (GGF) [1], which includes three components: consumers, producers and a directory service.

The first layer corresponds to a network of **Lookup Discovery Services** that provide discovery and notification mechanisms for all the other services.

The second layer is composed of **MonALISA services**, the components that perform the data collection tasks. Each MonALISA service is part of a group and registers itself with a set of Lookup Services, together with several describing attributes.

The interaction between clients and services is made available through transparent **Proxy services**, which represent the third layer in the MonALISA architecture. Every MonALISA service discovers the Proxy Services by using the discovery mechanism implemented into the Lookup Services layer, and permanently keeps a TCP connection with each of them.

The top-level layer is represented by the **MonALISA clients**, which offer an intuitive graphical interface of the states of the monitored systems. It allows users to subscribe to and to visualize global parameters gathered from multiple MonALISA services. It also provides detailed tracking of parameters for any individual MonALISA service or component in the entire system. The **MonALISA repository** is a "pseudo-client" for the MonALISA services, developed as a Web server. It is able to store the monitoring data and to present historical and real-time values, statistics and graphical charts for a specific group of MonALISA services. Each type of MonALISA client has to connect to the layer of Lookup services in order to request access to data gathered by one or more specified groups of MonALISA services. It is then transparently connected to the nearest and less loaded proxy service, which will forward the data that the client has subscribed to, from all the MonALISA services.

```
#include "ApMon.h"

ApMon *apm = new ApMon(ConfigFile);
        ...
apm -> sendParameter("MyParameterGroup", "NodeName",
                "MyParameter", XDR_REAL64, (char *)&value);
        ...
```

Figure 1: Instrumenting a code with the *ApMon* Library.

The MonALISA system is a well-suited choice for monitoring a distributed storage system, thanks to several features that it provides. First of all, it can monitor both a set of predefined parameters and various user-defined parameters. This is due to an application instrumentation library, called ApMon, that enables any application to send monitoring information to one or more MonALISA services.

BlobSeer is instrumented using the ApMon library, requiring each provider to report to the monitoring system each time a page is written or read, by sending a parameter and its value to a predefined MonALISA service, as described in Figure 1.

The version manager is monitored in the same way. An ApMon-based daemon parses its log file each time it is updated, in order to report the written page ranges and their associated versions. The state of the physical resources on each node is monitored through an ApMon thread that periodically sends the data to the monitoring service.

The MonALISA system also enables the user to create new data from the collected values, through the use of filters. In this way, new or aggregated values can be dynamically created within independent threads while the MonALISA service or repository receives the monitoring information. Since by default the repository can accommodate only time series in its database, all the specific data monitored from BlobSeer go through a filter that stores them into the corresponding database tables.

Another element is essential for defining a visualization tool tuned for a particular storage system. It is the possibility of having customized graphical charts, appropriate to the collected parameters. The MonALISA repository supports the integration of external graphical libraries, thus opening the way to the generation of any type of chart for any type of user-defined parameters.

2 General structure of monitoring

2.1 How the system works

As we talked in the background section, a BlobSeer system can have hundreds or thousands of data providers and lots of clients accessing the data concurrently. Monitoring the behavior of the system, as well as monitoing the clients, is a challenging task. In our approach, the version manager and each provider send monitoring data to the MonALISA system. This is done by additional monitoring infrastructures, for instance, listeners. The monitoring data is col-

lected by the *MonALISA* services and then forwarded to a *Repository*. The repository uses a database to store those received data. To process the users' requests, the repository queries the database, and then it creates and displays the visualization charts.

2.2 Types of monitoring information

The current monitoring service involves two types of monitoring information, which are the general information and BlobSeer specific information respectively.

2.2.1 General parameters.

The general type are such parameters that can be handled by the MonALISA repository module through meaningful monitoring predicates. Such parameters may include *CPU usage, network traffic, disk access, memory usage* etc. The repository subscribes to the specified parameters and stores them into its internal database. With these stored data, charts can be created simply by providing the repository with a dedicated property configuration file. This type of parameters described above are basically time series parameters, which means the x-axis of the charts typically contains time stamps. Sometimes, it is not sufficient. Therefore, we need application specific parameters.

2.2.2 Specific parameters.

The second type, namely the BlobSeer specific parameters, are such parameters that are dedicated to BlobSeer and can not be collected or shown simply by predicates and property configuration files. We thus need to create customized parameters which are subscribed to by the repository. When the monitoring data arrive at the repository, they first go through some filters that enable the creation of new aggregated parameters and the execution of some specific actions depending on the types and values of the incoming parameters. We may also discard parameters which we are not interested in.

2.3 Approaches of monitoring

Untill now, we have tackled the problem of collecting monitoring data from MonALISA and storing them in the database. Now we discuss the problem of collecting monitoring data from the application, in this case, BlobSeer. This task can be done my means of two different approaches. The first one is adding listeners to the entities that we want to monitor. The listener sends the monitoring data to the MonALISA service using ApMon. The second approach is to parse the log of the interested entity so as to obtain the monitoring information. In our case, we employed both approaches.

After the monitoring data are collected, we are ready to visualize them. As we mentioned above, the general information can be directly visualized by providing the repository with certain configuration files which are usually located at REPOSITORY_HOME/tomcat/webapps/ROOT/WEB-INF/conf. These configuration files can be used to create several different types of charts such as *real-time charts, history charts, statistics charts* and *spider charts* etc. Using these time-series based charts, we could display most of the general monitoring data.

For the second type of monitoring information, it's not so straightforward since the parameters are customized and we need to create the visualization charts from scratch. In our case, we are using the open-source chart library which is called *JFreeChart* to create the visualization figures.

3 Collecting the monitoring data

The regular monitoring data could be directly collected by the MonALISA service and repository's subscription function. Therefore here we only talk about how to collect application customized parameters.

3.1 Customized parameters

In the case of BlobSeer, the most important metrics are such parameters as how the blobs are accessed(read/write) and how the blob pages are distributed among the data providers etc. In the current implementation, we analyze two major aspects of BlobSeer parameters: blob I/O traffic and blob storage. The blob I/O traffic metric deals with how the blobs are read and written and we record every read and write operation in the database from two levels of perspectives: client level and provider level (see below). The blob storage metric deals with how each blob page is stored and it records some other basic metadata of each blob and blob page.

Due to the distributed working mechanism of BlobSeer, the read and write operations from the clients are split and distributed among data providers to achieve load balancing. Therefore, we record two levels of I/O traffic, namely client I/O and provider I/O. These two levels of I/O traffic are collected in different manners.

3.1.1 Provider level I/O

The data are collected by adding a listener to each provider and sending the monitored data using ApMon whenever a read/write operation occurs. The data sent to MonALISA service may include some basic metadata of that read/write operation, such as *blob_ id, page_ size, time stamp* etc.

3.1.2 Client level I/O.

For the I/O at the client level, we currently only deal with the WRITE operation. We use a parser to analyze the log file of the version manager and detect the client write operation by a special flag surrounded by the a pair of brackets. This process is done by the log parsing daemon which is running as long as BlobSeer is running. Whenever this daemon detects that a new blob or version is created, it inserts this new blob or version into the database. On the other hand, a more general way to handle it is to send it to the MonALISA service using ApMon and then insert into the database through the repository filters instead of communicating with the database directly. Actually this is what we have implemented.

4 Storing data into the repository

4.1 Repository structure

The MonALISA repository is mainly composed of an embedded database server and a web server. It also contains some other components like shell scripts and configuration files. But in this report, we focus on the former two major components.

4.1.1 Database

The database component is responsible for storing the data collected. When the repository subscribes to a specific predicate, it stores the corresponding data. This predicate could be a system-defined one or a customized one.

4.1.2 Web Server

The web server, which is based on a Tomcat web server, is responsible for dynamically creating and displaying the visualization charts. It uses JavaBeans to process the user requests and outputs them through a servlet. In next subsections we will discuss each of these components.

4.2 Configuring the database

The embedded *PostgreSQL* database is responsible for storing the collected monitoring data. Nevertheless, we could also use a standalone database instead of the embedded one for the purposes of having a newer version or a better performance. First we discuss the embedded *PostgreSQL* and later we will introduce how to connect to an external database.

The embedded database is located at REPOSITORY_HOME/pgsql_store. To start it, first go to the bin directory and type: pg_ctl start -D ../data. The -D switch specifies where the data folder of the database is located. We may also use a different location for the storage, say on a more stable partition of the hard disk. Use the command psql -d mon_data to get into console of the default database. Using the \dt command helps to show all the tables available in the current database. The monitor_ids table contains all of the regular monitoring parameter for every monitored node. Typically, every monitoring metric will have 2 or more resolutions which are indicated by the name of the table, such as *1y_1min* and *1y_1sec* (one value per min/sec during one year).

4.2.1 Common database commands

When using *PostgreSQL*, we sometimes need to list all of the available resources in the database. These command are usually called listing commands. They help you answer questions like "What tables are in this PostgreSQL database" or "What databases do I have within PostgeSQL". For example, \dt will show you all the tables. Other popular queries are related to permissions, indexes, views, and sequences. Following is a checklist.

Listing	Command	Argument
Tables	\dt	name
Indices	\di	name
Sequences	\ds	name
Views	\dv	name
Permissions	\dp or \z	name
System Tables	\dS	name
Large Objects	\dl	name
Types	\dT	name
Functions	\df	name
Operations	\do	name
Aggregates	\da	name
Comments	\dd	name
Databases	\l	name

4.2.2 Using a standalone database

To use a standalone database, the only thing we need to do is to modify the *Database Configuration Section* of the App.properties file which is usually located in REPOSITORY_HOME/JStoreClient/conf/.

```
lia.Monitor.jdbcDriverString = org.postgresql.Driver
lia.Monitor.ServerName = 131.254.11.254
lia.Monitor.DatabaseName = repository
lia.Monitor.DatabasePort = 5432
lia.Monitor.UserName = jcai
lia.Monitor.Pass = pg
```

Here we specify the server name or IP address, database name, database port and database user account information. You have to use this account to manipulate your database(execute commands such as *create, select, insert* or *update* table).

4.3 Configuring the repository

To enable the repository to subscribe to the desired monitoring data, we need to specify proper monitoring predicates in the *Store Configuration Section* of App.properties file. The repository is ready to receive monitoring data after it is started (or restarted) by launching the start.sh script in its home folder.

5 Creating visualization charts

In our implementation, we use JFreeChart to visualize the data we collected. JFreeChart is a free but powerful chart library for producing professional quality charts for developers and it's extremely easy to get started with it. To use latest version of JFreeChart, first download the library from the official website and unpack it. To make the repository recognize the JFreeChart library, put the $jfreechart.jar$ and $jcommon.jar$ into

```
REPOSITORY_HOME/tomcat/common/lib
```

Bear in mind that not always the latest version of JFreeChart is the best. Since the some of the regular charts coming with the repository internally use an older version of JFreeChart and since JFreeChart is itself under development, its API might change. If you come across an exception thrown out by the repository's regular charts, this might be the reason. Now we are ready to create a new chart.

In the current implementation, a new chart typically consists of three components:

- **the JavaBean** which creates the data set and the chart,
- **the servlet** which encodes the chart and outputs it and
- **the JSP web page** which finally displays the chart.

5.1 Chart Beans

Usually, each chart will have a JavaBean which actually creates it. The Bean typically consists of two major methods. One method which has a name like getDataset() does the job of communicating with the database and creating an appropriate data set for the chart. The other method which has a name like getChartViewer() creates a chart using the data set returned by the above method. In this method, different types of charts could be created and the characteristics of the chart, like *color, axis, background image* etc, could be tuned. The chart is then put in the *http* session after creation thus making it available for the image reading action. Finally, the getChartViewer method constructs an URL string and passes the image displaying job to the servlet. This URL string is what we will use in the JSP webpage for the <img> tag to display that chart. If the JavaBean needs any input parameters such as the blob_id, we need to write extra set() and get() functions. All of the chart beans are located in REPOSITORY_HOME/tomcat/common/lib/BSMonitoring.jar. Note that, we need to restart the repository to make the Java Beans/servlets take effect if they are modified or recompiled.

5.2 Chart Servlet

The chart servlet, which is called ChartViewer, simply gets the BufferedImage object from the current session, encodes it using PNG format and finally outputs it. It is a component that we may not need to touch unless we need to add more advanced features. It is currently located in the same JAR archive as the chart Java Beans: REPOSITORY_HOME/tomcat/common/lib/BSMonitoring.jar. Note that, if a new servlet is created, we need to activate it in

```
REPOSITORY_HOME/tomcat/webapps/ROOT/WEB-INF/web.xml
```

like this:

```
<servlet>
    <servlet-name>ChartViewer</servlet-name>
    <servlet-class>myServlet.ChartViewer</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet−mapping>
    <servlet−name>ChartViewer</servlet−name>
    <url−pattern>/servlet/ChartViewer</url−pattern>
 </servlet−mapping>
```

5.3 JSP webpage

The JSP webpage handles the user interface of the monitoring service and what we usually need to do is to write a JSP file which calls the chart bean.The chart bean will return an URL string which points to the servlet. This URL string will serve as the src attribute of the <img> tag and finally the servlet displays the desired chart. Note that, when creating the web page, we need to incorporate it in the master page which is used by the repository in order to achieve an uniformed user interface.

For more details, please reference the tutorial in the last section.

6 Conclusion and further work

In this report, we introduce the monitoring service which is specially designed for the BlobSeer, a large-scale distributed data management platform. We describe how the the large scale distributed application is monitored and visualized using MonALISA and also reveal some technical details.

Further, we will take into account the security problem of the BlobSeer system. We are currently dealing with the user accounting aspect. The user accounting is a recently introduced module and is still under development. In this module, we aim to collect informations about the users who do "malicious" operations to BlobSeer system. The malicious operations can be: writing data to the providers without publishing, publishing without writing any data to providers or publishing a part of the written data.

7 A tutorial on BlobSeer monitoring

This section we give a step-by-step tutorial for the newbies who would like to do more developments based on the current implementation of the BlobSeer monitoring service. In this tutorial, we will discuss the major components of the monitoring service, how to use them, and furthermore, how to create new modules using the existing infrastructure.

Before starting, please note that, from now on, we are always assuming that the current directory is REPOSITORY_HOME. Also, more detailed information can be found in the **README** file of each component.

7.1 Installation and Basic Configurations

7.1.1 MonALISA Service Installation

MonALISA is a large scale distributed monitoring service which has been used in many real monitoring applications. To start, simply install it in your preferred location, say $HOME/monalisa. The installation script install.sh will prompt you

for some information. One thing you need to notice is that you should enable the *ApMon* support since we will use *ApMon* to send the monitoring information later in this tutorial. You also have to give the correct path of java home on your system.

To start/stop/restart it, just go to the Service/CMD, and type ML_SER start/ stop/restart to proceed. In case you have given an incorrect java home path or want to change the name of your MonALISA service, simply modify the ml_env file in this directory.

7.1.2 Repository Installation

The second step is to set up the Repository. Similarly, install the repository using the install.sh shell first. Next, you may need to take some care of configuration files located in the /conf directory. Make sure you put the right value in the env.JAVA_HOME file which contains the path to the *JAVA_HOME* on your system. If you want to move the repository, modify the env.REPOSITORY_DIR accordingly. To start or stop the repository, simply launch the start.sh and stop.sh scripts. But if you want to start or shutdown the embedded database separately, use the start_pgsql.sh and stop_pgsql.sh.

7.1.3 Using a standalone database

We can use an external database in order to have a newer version or a better performance. The supported databases are *PostgreSQL* and *mySQL*. Here we will take *PostgreSQL* for an example.

Configuring the repository. First, we need to configure Database Configuration Section of the App.properties file in the ./JStoreClient/conf folder. For example:

```
lia.Monitor.jdbcDriverString = org.postgresql.Driver
lia.Monitor.ServerName = 131.254.11.254
lia.Monitor.DatabaseName = repository
lia.Monitor.DatabasePort = 5432
lia.Monitor.UserName = jcai
lia.Monitor.Pass = pg
```

Here we specify the server name or IP address, database name, database port and database user account information. You have to use this account to manipulate your database(for commands such as *create, select, insert* or *update* table).

Installing PostgreSQL. Go the the PostgreSQL website and download a latest release then unpack it. Here are the general steps for the installation. First we configure and install it.

```
./configure
gmake
su
gmake install
```

Then, we need to add a database user called postgres.

```
adduser postgres
```

We specify where the data is stored.

```
mkdir /usr/local/pgsql/data
```

Finally, we initialize it.

```
chown postgres /usr/local/pgsql/data
su −postgres
/usr/local/pgsql/bin/initdb −D /usr/local/pgsql/data
```

To connect to this database, go to the bin directory and type:

```
./psql −d postgres
```

To start or shutdown it, type:

```
./pg_ctl start/stop −D /usr/local/pgsql/data
```

The -D switch specifies the location of the data.

Allowing External Connections. The installation is not the last step. To allow our program (such as the daemon and filter) which is running outside the local machine to connect to the *PostgreSQL* database, we need to specify who and how the database is accessed. We shall add client authentication lines in the PostgreSQL_HOME/data/pg _hba.conf file like this:

```
local DATABASE USER METHOD [OPTION]
host DATABASE USER CIDR−ADDRESS METHOD [OPTION]
hostssl DATABASE USER CIDR−ADDRESS METHOD [OPTION]
hostnossl DATABASE USER CIDR−ADDRESS METHOD [OPTION]
```

If we are merely doing an experiment, we can trust all connecting clients.

```
host all all 0.0.0.0/0 trust
```

Also, we need to add authenticated listening addresses in the PostgreSQL _HOME-/data/posgresql.conf in order to allow outside connections. To allow all connections, we simply give:

```
listen_addresses = '*'
```

Finally, the listening port of the database can also be modified here.

7.2 Monitoring BlobSeer

The original version of BlobSeer in *SVN* does not support monitoring in general. To make the BlobSeer monitor-able, we need to make additional efforts. We used different approaches for monitoring different entities of BlobSeer.

7.2.1 Monitoring the Data Provider

The data provider's read/write operations are monitored by a listener. A monitoring listener is now already implemented and we may not need to change the class interface since we could extend it simply by adding more data-sending methods as follows:

```
void monitoring_listener::send_monitored_data
                          (const monitored_params_t &params)
```

These monitoring data are sent to the MonALISA service and thus can be subscribed to by any repository which is interested in them.

To enable this monitoring capability of the data provider, we can simply check out the /blobseer/contrib/monitoringBlobSeer/BSmonsupport folder, install the monitoring provider and use it instead of the original BlobSeer provider.

7.2.2 Monitoring the Version Manager

To monitor the version manager, we developed a *parser* to analyze the log file of version manager. The version manager will write a line in the log vmanager.stdout (which is usually located in /tmp/BlobSeer/vmanager/), such as:

```
[INFO 2009-Jun-03 16:15:24.639423] [~/deploy/release-0.3/vmanager/
                vmanagement.cpp:120:get_ticket] RPC success:
                allocated a new version 4 for request
                (1,4,1833659962,5797576704,2147483648) {CAV}
```

or

```
[INFO 2009-Jun-03 16:14:08.384510] [~/deploy/release-0.3/vmanager/
                vmanagement.cpp:190:create]
                RPC success: created a new blob: (1,1048576,1) {CCB}
```

The tags {CAV} and {CCB} mean *"Client Allocate Version"* and *"Client Create Blob"* respectively. These are the messages generated when the client writes on the blob or the client creates a new blob. The parser will read this log file from the beginning to the end and take corresponding actions whenever it finds one of these two tags. If no new data is available in the current log file, it sleeps for an interval whose value can be set in the script.

To use this parser daemon, we simple launch the program and provide it with the path to the log file as well as the configuration file.

```
export CLASSPATH = $PATH_TO_APMON/lib/apmon.jar:.
java vmonitor/VManagerMonitor version_manager_log config/monitor.conf
```

7.3 Displaying Regular Monitoring Information

7.3.1 Configuring the predicates

Now, we are going to use the repository to display some regular monitoring information. First we need to set proper predicates in the App.properties, which is located in the ./JStoreClient/conf folder. If we want to collect the users' *cpu usage* of **paralapeche** node which is monitored by the *BlobMonitor* MonALISA service, we can simply use a predicate like this:

```
lia.Monitor.JiniClient.Store.predicates=BlobMonitor/Monalisa/
                                paralapeche/-1/-1/cpu_usr.
```

7.3.2 Creating a time-series chart using the properties

Now, we create a history chart showing the user cpu usage over time. We need first create an empty file with a name extension of .properties in the ./conf/website_config_files directory. Let's say, it's called *usr_cpu.properties*. Inside this file, we may put some specifications about how the chart is displayed. Some example configurations are available in that folder. To make this chart take effect, we need add a link in the menu bar on the left side of the repository's web interface. To do this, simply add a line in the menu.js file which located in ./tomcat/webapps/ROOT/js as follows:

```
d.add(10111,1011,'Node CPU Usage','display?page=usr_cpu');
```

The first and second parameters may be different depending on how you organize the links. Now, open a browser and type in the url of the repository, click on the new link you have just created. If everything works out, you will see a new time-series chart about the user cpu usage.

7.4 Displaying Customized Monitoring Information

7.4.1 Configuring the subscription

Displaying the user-customized monitoring information is not so straightforward as above. As we mentioned before, the informations related to the data providers' I/O operations are collected via a listener and sent to the MonALISA service. On the other hand, the monitoring information from the version manager is sent by the log parser to the MonALISA service.

First we need to subscribe to our own customized parameters and configure the predicates as in the previous section. Next, since the monitoring data will go through the filters before being inserted into the database, we can develop multiple filters for different monitoring information or we can just extend one filter to take different actions depending on the received parameter.

```
String sParam = r.param_name[0];
if (sParam.equals("my_new_parameter_name"))
   {....do sth here and then insert into DB....}
else if (sParam.equals("my_old_parameter_name"))
    {....do sth here and then insert into DB....}
```

7.4.2 Creating a JavaBean

Now we can do some tests to write some data into the database. If everything goes fine, we have already got the data collected in the database. Now, let's create a demo JavaBean to generate a chart. Let's call this chart bean *myBarChart*. Now we create a project in the *Eclipse* or *Netbeans*, or whichever IDE you like. We need to add the **JFreeChart** library into the *Build Path* of the IDE. The following jars need to be added.

```
REPOSITORY_HOME/lib/jcommon-1.0.6.jar
REPOSITORY_HOME/lib/jfreechart-1.0.3.jar
REPOSITORY_HOME/lib/JStoreClient.jar
REPOSITORY_HOME/tomcat/common/lib/servlet-api.jar
```

We create a java package called myBean and create a java class called myBarChart under it. Import all of the classes that we may need. Some typical classes are:

```
import java.io.PrintWriter;
import org.jfree.chart.*;
import lia.Monitor.Store.Fast.DB;
import javax.servlet.http.HttpSession;
```

In the myBarChart class, we first create a method called getDataset(). The return value of this method depends on what type of chart we want to create. Here we defined some *series* and *categories* and then create a CategoryDataset.

```
String series1 = "First";
String category5 = "Category 5";
```

Then we create data set and add data into it.

```
DefaultCategoryDataset dataset = new DefaultCategoryDataset();
dataset.addValue(1.0, series1, category5);
```

The second function is called *getChartViewer()*. In this function, we first create the chart and set some characteristics of the chart. Next, we save it as BufferedImage object in the http session. Finally, we get the server name, port and context path and then concatenate them with the servlet's path which is /servlet/chartViewer. This concatenated string is actually the URL which will call our servlet and display this generated chart. Note that, the JavaBeans should be put in the directory of

```
REPOSITORY_HOME/tomcat/webapps/ROOT/WEB-INF/classes/myBean
```

or inside a JAR archive in

```
REPOSITORY_HOME/tomcat/common/lib/
```

The servlet doesn't need to be modified. It is now located in

```
REPOSITORY_HOME/tomcat/common/lib/BSMonitoring.jar
```

Note that, if a new servlet is created, we need to activate it in

```
REPOSITORY_HOME/tomcat/ webapps/ROOT/WEB-INF/web.xml
```

like this:

```
<servlet>
<servlet-name>ChartViewer</servlet-name>
<servlet-class>myServlet.ChartViewer</servlet-class>
<load-on-startup>1</load-on-startup>
```

```
</servlet>
<servlet−mapping>
<servlet−name>ChartViewer</servlet−name>
<url−pattern>/servlet/ChartViewer</url−pattern>
</servlet−mapping>
```

7.4.3 Creating the JSP webpage

Now, we create a corresponding web page to display this chart. All the webpages should be put in REPOSITORY_HOME/tomcat/webapps/ROOT/. In this JSP file we should first import the necessary libraries and then what we basically do is to call that JavaBean we have just created, like this:

```
String chartViewer=myBarChart.getChartViewer(request , response);
sOut+="<img src="+chartViewer+" border=0 usemap='#imageMap'>";
```

Note that the java bean should be included it in the JSP page.

```
<jsp:useBean id="myBarChart" scope="session" class="myBean.myBarChart"/>
```

We also have to incorporate our web page into the master page.

```
ServletContext sc = getServletContext();
final String SITE_BASE = sc.getRealPath("/");
final String BASE_PATH = SITE_BASE+"/";
final String RES_PATH = SITE_BASE+"/WEB−INF/res";
ByteArrayOutputStream baos = new ByteArrayOutputStream(100000000);
Page pMaster = new Page(baos, RES_PATH+"/masterpage/masterpage.res");
```

Done, we are now at the last step. To make this web page appear, we need add a link on the menu bar. Add a new line in the ./js/menu.js like this:

```
d.add(101223,10122,'myBarChart','myBarChart.jsp');
```

If everything works well, the new chart is completed and can be displayed.

References

[1] Global Grid Forum. `http://www.ggf.org/`.

[2] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Distributed management of massive data. an efficient fine grain data access scheme. In *International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2008)*, Toulouse, 2008. Held in conjunction with VECPAŔ08. Electronic proceedings.

[3] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture. Grid Working Draft GWD-PERF-16-3, August 2002. `http://www.gridforum.org/`.

[4] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[5] The MonALISA Website. `http://monalisa.cern.ch/`.

[6] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *5th Intl. Workshop on Peer-to-Peer Systems (IPTPS-2006)*, Santa Barbara, USA, February 2006. Electronic proceedings.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803