Distributed Monitoring for User Accounting in BlobSeer Distributed Storage System

Master Thesis

Mihaela-Camelia VLAD
mihaela-camelia.vlad@irisa.fr

Supervisors: **Alexandra Carpen-Amarie, Gabriel Antoniu, Luc Bougé**
Alexandra.Carpen-Amarie@inria.fr, Gabriel.Antoniu@inria.fr, Luc.Bouge@bretagne.ens-cachan.fr

INRIA, KerData Project-Team

February 1st, 2010

Abstract

This paper addresses the problem of using monitoring for keeping track of user activity inside a large-scale distributed storage system. Also, we are trying to determine the possible malicious actions performed by such users. In order to do this, we designed a monitoring system that uses MonALISA for the gathering of data and a distributed storage within Postgres databases. As a result, our system handles and stores large amounts of monitoring data, and it processes it in an on-line manner, in order to obtain real-time information about the users in the system and possibly block those with malicious intentions.

Keywords: Distributed system, storage management, large-scale system, malicious users detection, monitoring

Contents

1 Introduction

1.1 Context

Cloud computing [6], [18] is an emerging paradigm that is becoming increasingly popular in both industry and scientific communities. It promotes a new and innovative concept in managing hardware and software resources: instead of buying and maintaining them, users can rent virtual machines, storage space or already deployed software platforms. Cloud services have been proposed by leading industry companies, such as Amazon [15], Yahoo, IBM, Google [16] or Microsoft [17], which have strongly promoted the concept during the last years. In addition to those, the academic communities are also becoming interested in this topic. There are open-source projects, such as Nimbus [11] or Eucalyptus [10], that aim at providing a Cloud computing framework with the help of virtualization to members of scientific societies, for studies on the subject.

In this context, data management is a key issue. While users are provided with the ability to store data on remote, virtual resources, the need for mechanisms able to provide feedback about the state of the system becomes obvious. Monitoring its condition and activity can provide significant progress in areas such as efficient management of resources, performance, quality of service or security [7].

Client monitoring is important in ensuring that the terms of usage stipulated by the provider are being respected. The storage service has to be aware of its different client types, each with specific privilege levels, access rights or quality of service requirements [20]. When policy violations are detected, the system needs to enforce adaptive security rules.

These objectives could be reached by developing a system that continually analyzes the user activity and monitors the general state of the system, in order to determine abnormal activity or potential insecure events happening within.

1.2 Motivation

This work addresses how monitoring can be used in keeping track of user activity inside a cloud storage system. It discusses the useful data that should be collected by the monitoring tools in order to present the user with a detailed and meaningful image of the storage system and of the data inside. A key point is to identify how the data must be processed in order to obtain a real-time image of the integrity of data within the system and of the users creating it.

As a case study, we use the BlobSeer [8] distributed storage system as a framework for our work. BlobSeer is a large-scale data-sharing system, which aims to efficiently manage the storage of large and unstructured binary data blocks.

We also have to consider the monitoring challenges raised by a large-scale storage system, such as the large number of nodes, the fine-grained striping of the data over the storage nodes and the heavy concurrent access to data. The MonALISA system [5] is a monitoring framework designed as an ensemble of autonomous subsystems which are registered as dynamic services and cooperate in performing a wide range of information gathering and processing tasks. It is a system able to meet our challenges, and a user accounting system for BlobSeer can rely on it for gathering the required information.

Our work is based on some previous research addressing how monitoring can be used in keeping track of user activity inside BlobSeer: a 3-layered architecture has been designed in order to bring introspection capabilities within the system. Data is indexed based on the BLOB it refers and collected inside a centralized storage, where it can be consulted in order to view or extract information.

Using this work as a starting point, we have designed our own monitoring system, fine tuned for gathering and interpreting information about the users of BlobSeer. What we propose in addition to earlier work is a *distributed* architecture for storing and processing the received data, and also a module that uses the results obtained to compute a list of what we call malicious clients.

2 Background knowledge

2.1 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. It is developed inside the KerData Project-Team, at IRISA, France, and hosted at blobseer.gforge.inria.fr.

BlobSeer addresses the problem of efficiently storing massive BLOBs in large scale distributed environments. The BLOBs are striped into small chunks that have the same size, called pages. 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.

The system consists of distributed processes that communicate through remote procedure calls (RPCs). A physical node can run one or more processes and, at the same time, may play multiple roles from the ones mentioned below.

Clients. Clients may issue CREATE, WRITE, APPEND and READ requests. Their number dynamically varies in time without notifying the system. 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.

Data providers. Data providers physically store and manage the pages generated by WRITE and APPEND requests. New data providers are free to join and leave the system in a dynamic way.

The provider manager. The provider manager keeps information about the available data providers. When entering the system, each new joining provider registers with the provider manager. The provider manager tells the client to store the generated pages in the appropriate data providers according to a strategy aiming at global load balancing.

Metadata providers. Metadata providers physically store the metadata, allowing clients to find the pages corresponding to the various BLOB versions. Metadata providers may be distributed to allow an efficient concurrent access to metadata.

The version manager. The version manager is the key actor of the system. It registers update requests (APPEND and WRITE), assigning BLOB version numbers to each of them. The version manager eventually publishes these updates, guaranteeing total ordering and atomicity.

For each BLOB, the metadata is organized as a distributed segment tree [21], 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 [12].

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

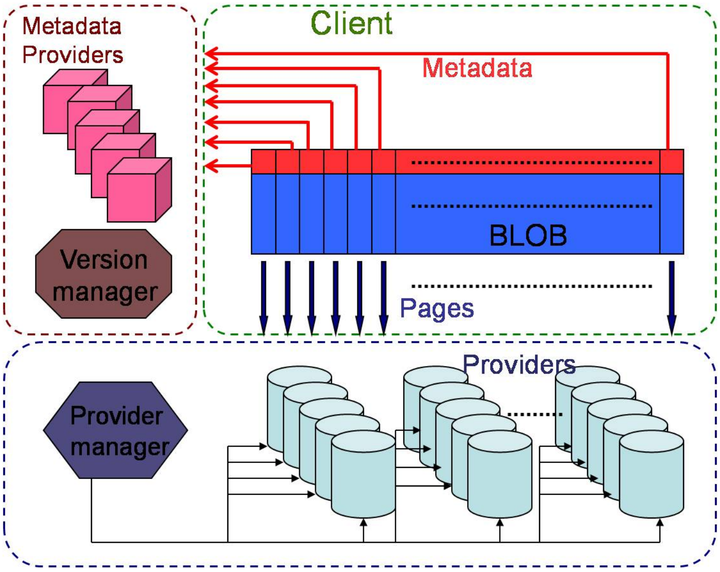

Figure 1: BlobSeer Architecture

points to the newly added pages and to the pages from the previous versions that were not overlapped by the added page range.

The interactions between the entities of BlobSeer are briefly illustrated in Figure 2.

For a WRITE request, the client contacts the provider manager to obtain a list of providers, one for each page of the BLOB segment that needs to be written. Then, the client contacts the providers in the list in parallel and requests them to store the pages. Each provider executes the request and sends an acknowledgment to the client. When the client has received all the acknowledgments, it contacts the version manager, requesting a new version number. This version number is then used by the client to generate the corresponding new metadata. After receiving the acknowledgment, the client reports the success to the version manager.

A READ request begins with the client contacting the version manager to get the version of the corresponding BLOB. If the specified version is available the client contacts the metadata provider to retrieve the metadata associated with the pages of the requested segment for the requested version. After gathering all the metadata, the client contacts (in parallel) the data providers that store the corresponding pages.

As far as this report 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 than 1 MB, whence the need to deal with hundreds of thousands of pages belonging to just one blob.

For this project, we only monitor WRITE operations and we study the impact of clients'

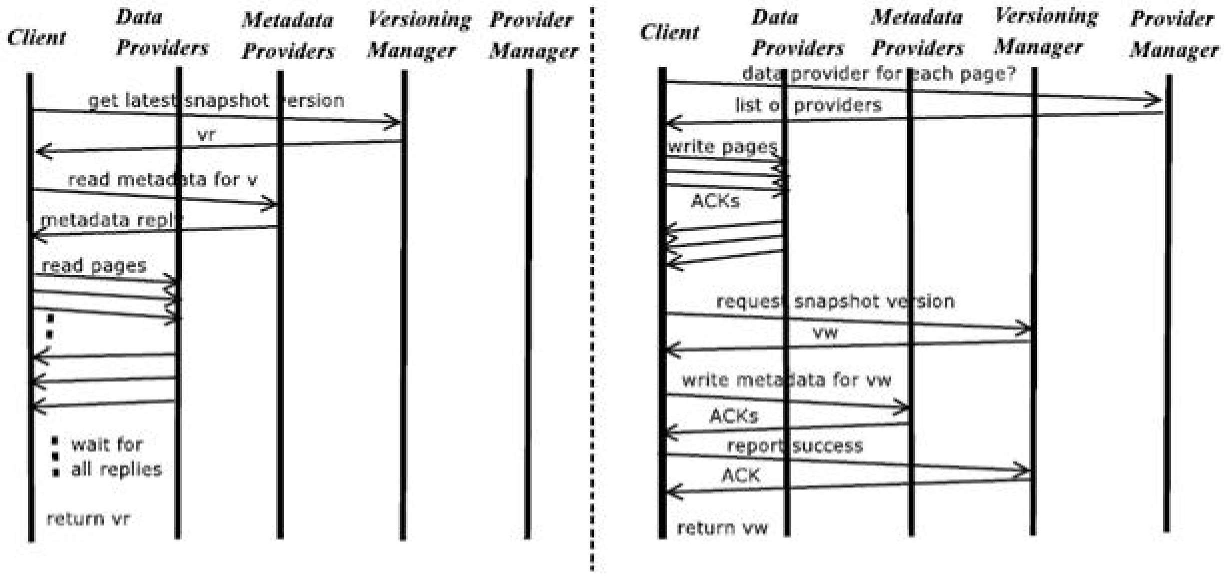

Figure 2: Internal interactions inside BlobSeer: READ(left) and WRITE(right)

behavior upon the normal functioning of the system.

2.2 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) [9] system is a JINI-based [19], scalable framework of distributed services, which provides the necessary tools for collecting and processing monitoring information. The system is designed as an ensemble of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services, and are able to collaborate and cooperate in performing a wide range of information gathering and processing tasks. These agents can analyze and process the information, in a distributed way, to provide optimization decisions in large scale distributed applications. An agent-based architecture provides the ability to invest the system with increasing degrees of intelligence, to reduce complexity and make global systems manageable in real time. The scalability of the system derives from the use of multithreaded execution engine to host a variety of loosely coupled self-describing dynamic services or

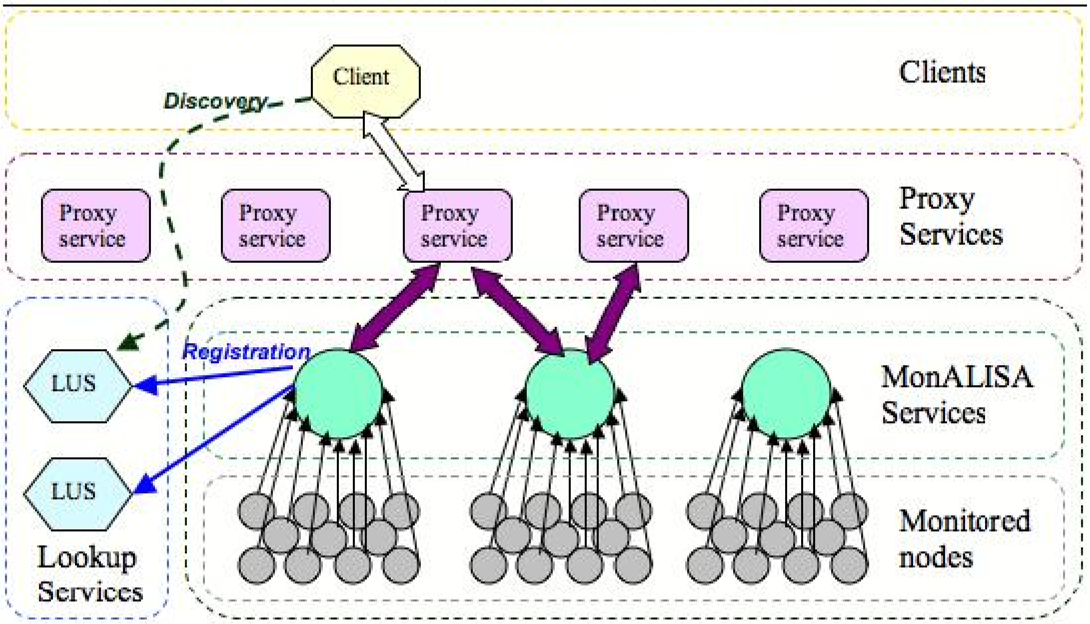

Figure 3: The MonALISA Architecture

agents and the ability of each service to register itself and then to be discovered and used by any other services, or clients that require such information. The system is designed to easily integrate existing monitoring tools and procedures and to provide this information in a dynamic, customized, self describing way to any other services or clients.

Its architecture is based on four layers of services, as presented in Figure 3. It complies with the Grid Monitoring Architecture (GMA) [14] proposed by the Global Grid Forum (GGF) [4], 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. 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.

A **Monitoring Module** within MonALISA is a dynamically loadable unit which executes a procedure (or runs a script / program or performs SNMP request) to collect a set of parameters (monitored values) by properly parsing the output of the procedure. In general a monitoring module is a simple class, which is using a certain procedure to obtain a set of parameters and report them in a standard format.

Monitoring Modules can be used for pulling data and in this case it is necessary to execute them with a predefined frequency (for example a pull module which queries a web service) or to "install" (it has to run only once) programs which are sending the monitoring results (via SNMP, UDP or TCP/IP) periodically back to the Monitoring Service. Allowing to dynamically load these modules from a (few) centralized sites when they are needed makes much easier to keep large monitoring systems updated and to provide new functionalities dynamically. Users can implement easily any new dedicated modules and use it the MonALISA framework.

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**, which enables any application to send monitoring information to one or more MonALISA services. The monitoring data is sent as UDP datagram to one or more hosts running MonALISA services. Applications can periodically report any type of information the user wants to collect, monitor or use in the MonALISA framework to trigger alarms or activate decision agents. The ApMon implementations are provided for 5 programming languages: C, C++, Java, Perl and Python.

3 Addressing security issues in BlobSeer

3.1 Security issues in BlobSeer

Part of BlobSeer's efficiency is enabling high concurrency in parallel writes, on the same or on different providers. Only metadata writes are serialized, in the case where more clients are writing the same part of a blob simultaneously. Although this approach enables performance gain, it also has a security downside: BlobSeer processes being completely independent of each other, the system as a whole has no way of knowing whether their actions are always consistent with each other, and maintain the system's integrity.

For instance, if the version manager process is killed for some reason, the provider manager has no way of knowing this, and it will still supply lists of valid providers to clients, which will write their data to them. Upon completion of the writing on providers, a client will normally try to contact the version manager in order to publish a version number for his operation. Since the version manager is unreachable, the publish operation will fail and cause the whole write to fail. Some replication mechanism of the version manager could be imagined to prevent this from happening. But in any case where the process could be unresponsive for a while, clients will unwillingly damage the system by loading up the providers with useless information.

A security issue may be the fact that BlobSeer has no authentication mechanism or any way of making distinctions between users. Every one of them accesses the blobs in the same way. Also, from the storage point of view, blob access rights are the same for all the users, and each user can access any part of a given blob. So if a user want to damage the data written into a blob by someone else, he has the possibility of doing so.

Also, BlobSeer's code is open source, and therefore anyone can access and modify it as they like. In the context of using it in a cloud environment, where BlobSeer is accessed through the client, there is no way of certifying the users. One could modify the *object_handler* component, the client's interface for accessing BlobSeer, in such a way that what is reported to the version manager is no longer consistent with what is written to the providers. Further on, we will refer to this type of malicious users and we will explain how their behavior can affect the system, and also propose a way for it to be detected in the shortest time possible.

3.2 Malicious clients

Based on how the code can be modified to create inconsistencies within writes, we have identified 3 types of such clients:

- **WriteNoPublish** (WNP) client - it is the client that writes a certain size of data to one or more providers, but it doesn't further contact the version manager to publish a version of what has been written.
- **PublishNoWrite** (PNW) client - the client that publishes a new version to the version manager (and writes it into the metadata tree) without actually writing anything to providers.

- **IncorrectWrites** (IW) client - a client who writes to providers a different number of pages than the one it reports to the version manager for versioning.

WNP client: The WNP client's damaging potential is that if many such clients are running, they will fill up the providers with useless information that nobody inside the system will ever know exists. Therefore, a lot of storage space can be wasted as the result of an attack by such clients. In an extreme case, if all the storage space is filled up, then the providers will have no more room to store information, and the running BlobSeer service will become unresponsive. Even so, this type of malicious client is the least dangerous, because it doesn't directly affect the functioning of the service's components, and if neutralized on time its effects will not be long-term.

PNW client: On the other hand, the PNW client's effects are experienced right away within the system. The metadata tree is constructed in such a way that each new version of a blob is connected by pointers to the previous version. So if a version written by a PNW client will end up in the metadata tree, and another valid version is then tied to it, a client wanting to read the latest version of a blob will end up in the following situation: BlobSeer will try to take the parts of the blob that were not covered by the last write from the previous version, the one that doesn't actually have any blob pages associated to it. At this point, the client will either read incorrect information, or some operations may fail within BlobSeer because of invalid pointers. One PNW client affects not only one version within BlobSeer, but all of the following ones too.

IW client: The IW client has the same effect as the PNW client: introducing inconsistent information within the system. This affects both the versioning system and the storage on the providers, as the information written during and after such a write cannot be properly accessed.

We implemented these clients as we have suggested earlier, by modifying the code of the *object_handler* of BlobSeer, in the following way: for IW, ensuring that the number of pages written to a provider is generated randomly, rather than the one reported. For WNP and PNW we suppressed from the WRITE operation either the part that writes to providers, or that which writes to the version manager.

3.3 Our approach: Using BlobSeer monitoring in user activity history

In order to prevent such attacks or inconsistencies, multiple security strategies can be envisioned. One solution would be that the version manager checks with the providers for the validity of a read before publishing, but that would be a huge performance overhead. The WRITE operation could also be modified to time-out if after a certain number of pages written no version is published.

In order to preserve as much of the initial performance, our approach consists of creating a monitoring system, independent of BlobSeer, which gathers information about its components and computes a history of user activity in the system. This can then be used for multiple purposes, including making some security decisions.

A system using a BlobSeer deployment 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 monitoring 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 collected by the MonALISA services and then forwarded to a distributed processing system. Our goal is to run the monitoring system in parallel with BlobSeer and monitor its clients' activity in real time. The system will maintain a list of malicious users updated periodically. This list could be fed back to BlobSeer, more specifically to the provider manager. In this way, if a malicious client tries to contact the provider manager, in order to get a list of providers for a new write, he can be denied based on his history. On condition that the information comes back to BlobSeer fast enough, this can be an efficient implementation of a security policy.

The user history is computed in the following way: once the write operation is initiated by a client on a provider (typically for one page), this provider will send to MonALISA information about the page number, page size, the hostname of the client that initiated the operation and a timestamp. At the version manager level, information about the client ID, total size of the write, page size and version number is sent. Actually, because a version number is not attributed to a write until the end, it cannot be used in the information sent in real-time by providers. In order to support a connection between the information reported by the two entities, a watermark was associated to each write operation within BlobSeer, watermark that is reported by all of the providers and by the version manager. It is randomly-generated and its only purpose is making the connection between the two. After the information reaches MonALISA, it is forwarded to a custom processing system consisting of several computing nodes. The number of pages written on the providers by a client is then compared to the size reported to the version manager (and also checked if the page sizes correspond). If they match, we consider the write to be a normal operation and if they don't, we compare the write sizes and classify it in one of the 3 malicious clients' categories.

3.4 Contribution: Architecture of the user accounting module

3.4.1 Related work

One of the first examples of an intrusion detection system is Haystack [13]. It defined a range of values that were considered normal for each feature and if a feature fell outside the normal range during a session, the score for the subject was raised. Considering the features to be independent, a probability distribution of the scores was calculated and an alarm was raised if the score was too large. Haystack also maintained a database of user groups and individual profiles. If a new user had been detected, a new profile based on restrictions related to group membership was created. One disadvantage of Haystack was that it was designed to work offline, because of the statistical analyses it required. Those could not be done on-line because this would have required high-performance systems. Our distributed monitoring addresses just the problem of distributing costly processing among nodes. Together with the filtering of data at the MonALISA level, this allows us to manipulate a large amount of information in an on-line manner.

Crosbie et al. [3] proposed to apply genetic algorithms to the problem of intrusion detection. It is a search technique used to find approximate solutions to optimization and search problems. Such a search converges to a solution from more than one direction, and it is based on probabilistic rules instead of deterministic ones. The authors applied multiple agent technology to detect some network based anomalies, using numerous agents to monitor different network based parameters. Nevertheless, the disadvantage to this approach is that it was not fast enough, the training process taking long. This makes for one prerequisite that our system does not support, because the monitoring has to be simultaneous with the functioning of BlobSeer, and it must start at the same time.

In [1], the authors put forward a distributed architecture with autonomous agents operating independently of each other in order to monitor security-related activity within a network. They suggest a scheme of escalating levels of alertness, and a way to notify other agents on other computers in a network of attacks so they can take reactive measures. A neural network is supposed to measure and determine alert threshold values. The progressive increase in alert levels is costly with respect to detection time, and for a system such as BlobSeer, a similar approach would not be fast enough. Although this kind of detection algorithm is more complex, our system requires an almost immediate detection of malicious actions, in order for its functioning not to be compromised.

3.4.2 Previous work: Bringing introspection to BlobSeer

Our work is based on previous research carried inside the KerData team related to creating an introspective BlobSeer. In [2], the authors propose a first prototype of such architecture. It comprises 3 layers aiming at identifying and generating relevant information related to the state and the behavior of the system, as depicted in Figure 4. Such information is then expected to serve as an input to a higher-level self-adaptation engine (currently not implemented yet). These data are yielded by an (1) introspection layer, which processes the raw data collected by a (2) monitoring layer. The lowest layer is represented by the (3) instrumentation code that enables BlobSeer to send monitoring data to the upper layers.

The data generated by **the instrumentation layer** are relayed by the monitoring system and finally fed to the introspection layer. The instrumentation layer is implemented as a component of the monitoring layer. The version manager and providers are equipped with ApMon listeners, in order to monitor the READ and WRITE operations.

The input for the introspective layer consists of raw data that are extracted from the running nodes of BlobSeer, collected and then stored, a set of operations realized within **the monitoring layer**. The main challenge the monitoring layer has to cope with is the large number of storage provider nodes and therefore the huge number of BLOB pages, versions and huge BLOB sizes. Furthermore, it has to deal with hundreds of clients that concurrently 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 for large-scale environments and it proved to be both scalable and reliable.

The introspection layer corresponds to a MonALISA client, the MonALISA repository. It is the location where the data is stored into a central database and made available to the introspection layer. At this level, the user can consult various graphical representations concerning the state and the activity within the system.

As a work in progress, a centralized user accounting system is being tested on top of the architecture presented above. The three types of malicious users presented earlier are computed based on the periodical querying of the repository database. Users with inconsistent activity are selected and attributed a score. A black list is computed periodically and made available to clients who request it. The downside in this approach is that the centralized database is a bottleneck in the functioning of the system, and that data stored is not structured in the most efficient way for this purpose.

3.4.3 Architecture of the user accounting module

Our work represents an extension of [2]. We have modified the architecture described above and rewritten the layers, in order to adapt it to a higher degree of distribution in the processing performed on the data. Our system has the following layout, presented in Figure 5 and has 2 main components:

- The monitoring system, which has 3 layers:
  - Instrumentation layer
  - Monitoring layer
  - Data processing layer
- User accounting module

Such information is then expected to serve as an input to a higher-level self-adaptation engine, which is to be implemented within BlobSeer, as shown.

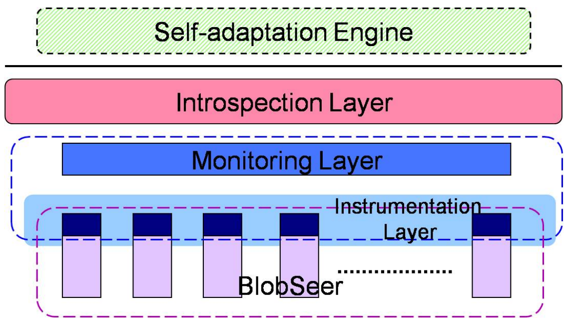

Figure 4: Architecture of the introspective BlobSeer

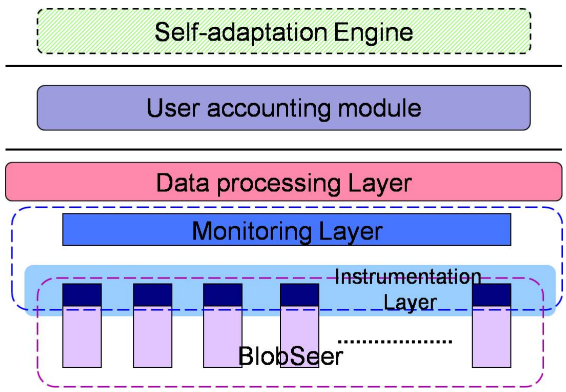

Figure 5: User accounting architecture within the introspective BlobSeer

The monitoring system is the one handling the collecting and processing of data related to client activity in BlobSeer. This information can be used in multiple ways, but our approach uses it for creating a user accounting system. The second module of our work is one that uses given information about users to compute a history for each of the clients, determining whether they performed malicious actions and how many times.

- **The instrumentation layer** is the same as the one in Figure 4. As we stated before, we equip each provider with a listener attached to the WRITE operation, so that when a client performs this operation, information about the page number, client ID and page size is sent to MonALISA. On the version manager side, similar information about a new version of a specific blob is sent using a parser that monitors the events recorded in the logs. The difference between these two types of listeners is that for one WRITE we have one version manager message to MonALISA, while on the provider side we have one for every page written. The state of the physical resources on each node is monitored through an ApMon thread that periodically sends data to the monitoring service.
- **The monitoring layer** is represented by MonALISA and the actions taken at this level about the data collected. We have implemented a filter within MonALISA, customizing these actions. Data received by providers is condensed into one data entry, and further on, both provider and version manager data are sent to a distributed storage and processing system. Information from a specific client is statically mapped to be sent each time to the same processing node, ensuring that there is one node containing the complete information about that user.
- **The data processing layer** could be considered as a distributed introspection layer, but it also integrates continuous processing of the data available here. It is a system that could itself be divided into 3 sub-layers, at each node level. First, there is one that stores all the data coming from MonALISA into some specific tables, without any kind of processing of the information. The second one updates a table periodically where it tries to condense information from the previous one, by creating a single entry into a table, with both version manager data and providers' data. The last sub-layer encapsulates a per-user history updated periodically with respect to the number of correct and incorrect operations performed.

The user accounting module is located on another node than the ones processing the data. It gathers data from all the nodes and it attributes a score to each of the clients in the history. Updates are done periodically. It also allows a client module to connect to it and request the computed list at any time.

4 Distributed monitoring system components

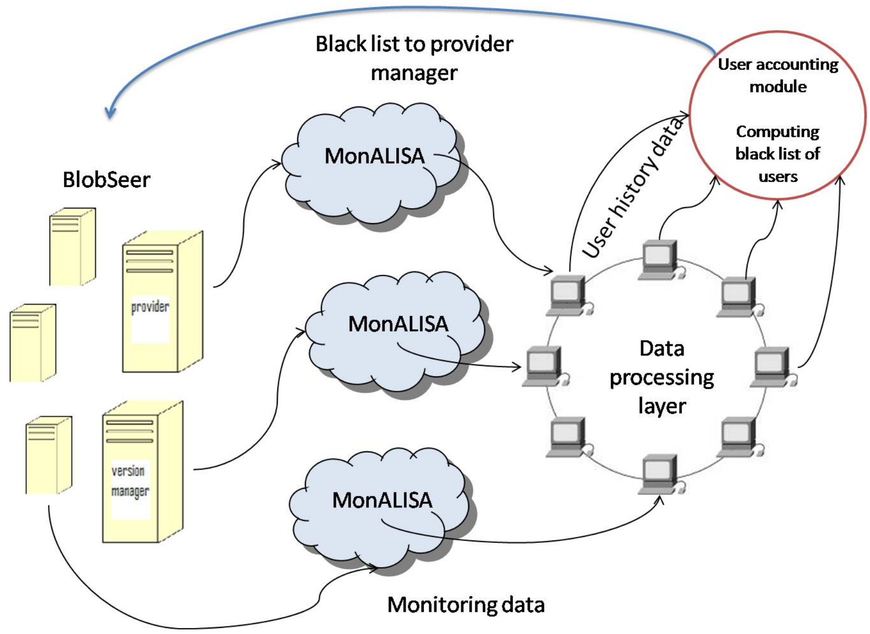

Figure 6: Distributed monitoring system infrastructure

4.1 Instrumentation layer

BlobSeer is instrumented using the ApMon library. The routines provided by the library handle the encoding of the monitoring data in the XDR representation and the building and sending of the UDP datagrams. As shown in Figure 7, the applications can use the API to send any specific parameter values to one or more MonALISA services.

4.1.1 Provider level

BlobSeer providers have the ability to have listeners attached, so in order to add ApMon support to BlobSeer, the *monitoring_listener* class was implemented. Each of the listeners' update method is called when a read or write operation is performed, and so it is reported to the monitoring system. The values sent are encoded within a string and separated by the " # " character, as shown in Figure 9.

The parameters sent are the following: client hostname, timestamp, blob id, MonALISA farm name, node name, page size, index of the current page within a write and the watermark we discussed earlier in this paper.

4.1.2 Version manager level

The version manager is also monitored. An ApMon-based daemon runs in parallel, and parses its log ?le each time it is updated, in order to report the written page ranges and their

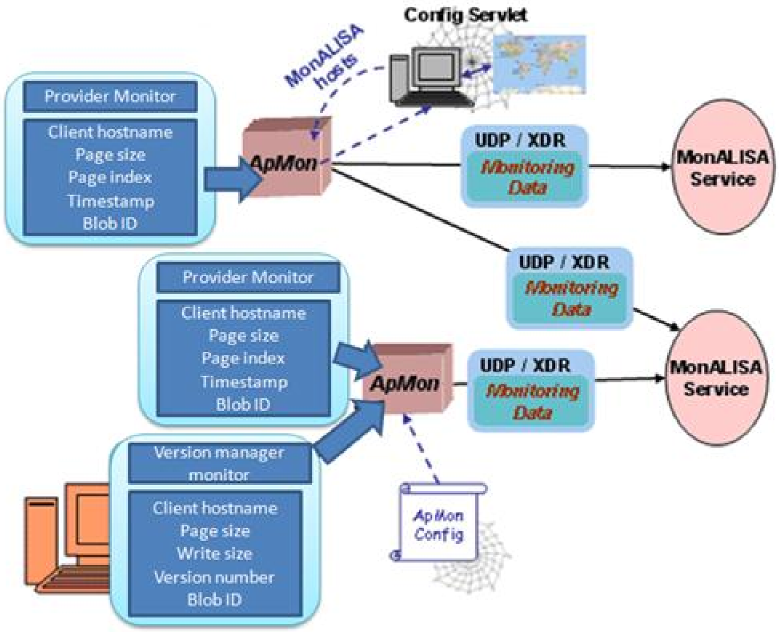

Figure 7: Collecting monitoring data from BlobSeer

```
#include "ApMon.h"

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

Figure 8: Instrumenting a code with the *ApMon* Library

associated versions. Whenever this daemon detects that a new blob or version is created, it sends this information to the MonALISA service it is configured for. In order for this approach to be consistent, we rely on a specific format of the log file.

The parameters collected from the log and sent are the following: client hostname, timestamp, blob id, watermark, blob version, page size, write size and write offset within the blob.

In order to configure ApMon, we need to set the destination hosts to which it will send data in a special file, in our case, a *monitor.conf* file, together with some other additional parameters, such as maximum message rate or the interval between datagrams. MonALISA also has to have ApMon support enabled (in the ‹FarmName ›.conf properties file), as shown in Figure 10.

4.2 Monitoring layer - The MonALISA Filter

Once the data we are interested in reaches MonALISA, it has to be processed in a custom manner, and sent further to a set of specified hosts. This is why we have chosen to imple-

```
ss<<range->id<<"#"<<range->offset<<"#"<<range->version<<"#"
    <<params.get<3>()<<"#"<<now;

buf = ss.str();

apmon->sendParameter("Blob_IO", NULL, "provider_write", XDR_STRING,
const_cast<char*>(buf.c_str()));
```

Figure 9: Sending monitored parameters on provider side

```

# the ABping module
# *ABPing{monABPing, localhost, "␣"}
^monXDRUDP{ParamTimeout=900,NodeTimeout=900,ClusterTimeout=900,
                    ListenPort=8884}%20
```

Figure 10: Enabling ApMon support on MonALISA

ment a filter at its level. A filter is a monitoring module dynamically loadable at service startup that instantiates some predefined methods when receiving data it is interested in. Our class is called *BsMonFilter*, and in order to use it as a MonALISA filter, it has the following structure and properties:

- It must extend lia.Monitor.Filters.GenericMLFilter
- It must have a constructor with a String param (the FarmName) in which you must call super(farmName). This constructor is used to dynamically instantiate the filter at runtime
- Our filter has 4 configuration parameters, that must be set in the ml.properties file of MonALISA:
    - *BSMonFilter.ConfigFile* - contains the values of the PREDICATES parameters that must be set by the user. It specifies a list of predicates to filter a desired set of results, in the following format: $F/C/N/Param_1|Param_2|...|Param_n$, where:
        - F - Farm Name
        - C - Cluster Name
        - N - Node Name
        - Param_x - Param Name
    - *BSMonFilter.nodeFile* - the path to a text file containing the list of nodes to which the monitoring data will be sent for processing. The format of the file must be "hostname port" on each line. The filter acts as a client to those nodes and will try to connect to them on the given port.
    - *BSMonFilter.Port* - the default port on which to connect to the nodes in the nodefile, in case no port is specified for one or more hosts.

◦ *BSMonFilter.filterTimeout* - the delay between sending the monitoring data to processing nodes

- The filter must override the following methods:

  ◦ *public String getName()* - returns the Filter name. It is a short name to identify data sent by the filter in the client. It is also used by MonALISA clients to inform the Service that they are interested in the data processed by this filter. It must be unique because all the filters in ML are identified by their name. Our filter's name is *BsMonFilter*.

  ◦ *public monPredicate[] getFilterPred()* - returns a vector of *monPredicate(s)*. These predicates are used to filter only the interested results that they want to receive from the entire data flow. If it returns null, the filter will receive all the monitoring information. In our case, because we are only interested in the WRITE operation, we have set the PREDICATES property in the file specified in BSMonFilter.ConfigFile to filter parameters received from any farm and node, in the group *Blob_IO* parameter *provider_write*, and in the group VManager parameter blobWrite: $PREDICATES = */Blob\_IO/*/provider\_write; */VManager/*/blobWrite$

  ◦ *public void notifyResult(Object o)* - This method is called every time a Result matches a predicate defined above. The Filter could save this in a local buffer for future analysis, or it can take some real time decision(s)/action(s) if it is a trigger.

  ◦ *public Object expressResults()* - This method is called from time to time to let the filter process the data that it has received. It should return a Vector of *eResults* classes that will be further sent to all the registered clients, or null if no data should be sent to Clients.

  ◦ *public long getSleepTime()* - returns a time(in milliseconds) for how often *expressResults*() should be called. In our case, it returns the value configured in the *ml.properties* file.

- In the ml.properties file, the path to the directory where the filter has its .class file must be added. The parameter that must be defined is *lia.Monitor.CLASSURLs*, and if there are more filters/directories they have to be separated by commas. The trailing "/" on the end of each of the paths is essential; the class will fail to load without it. $lia.Monitor.CLASSURLs = file: \$\{path\ to\ BsMonFilter.class\ file\}/$

- Also, in the ml.properties one must specify what filters should be loaded: $lia.Monitor.ExternalFilters = processing.filters.BSMonFilter$

Our filter does the following:

- First, it loads its configuration from the required parameters. It initializes the variables specified the ml.properties file, it reads the list of nodes from the nodefile, and it initializes the parsing predicates values.

- Based on the predicates we defined, it receives only the strings of data that we are sending from the BlobSeer entities.

- Within the notifyResult method, results are parsed and are stored within special data classes until the timeout for the expressResults method expires. In case several provider pages of the same write arrive before an expressResults is called, the pages are aggregated within a same data class.
- When the timeout expires, all the data kept in the cache is sent to the configured nodes, attributing each new client a node - using an algorithm that has the same results on each MonALISA node. In this way, data coming from one client is always sent to the same node, in a serialized manner. All of the MonALISA services used within the deployment must be configured in the same way (same configuration file).

4.3 Data processing layer

The data processing layer is itself structured into 3 sub-layers, each with specific roles within the system. We use this 3-step processing to both store the data we receive from MonALISA and to process it extracting information related to the behavior of the clients within the system.

First, we handle storing all the data that we receive. This can be used further in any kind of situation where we need information about BlobSeer operations and client activity related to those. Next, we introduced an intermediate layer that summarizes the information received previously. The results are also stored into the database, in a single table on each node. The final layer summarizes the number and type of operations per each user. This is the final abstraction we use about user activity, and the information held here are used further within the user accounting module.

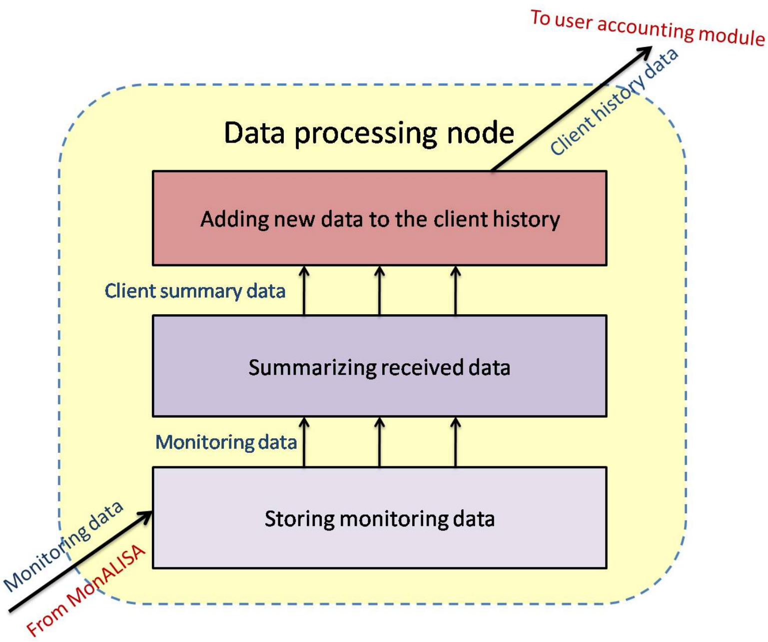

Figure 11: Architecture of a data processing node

4.3.1 Storing monitoring data

For each node, the components of the first layer, which stores all the data received, are the following:

- The database stored on each node
- A running Java Nio Server
- The process that updates entries in the database

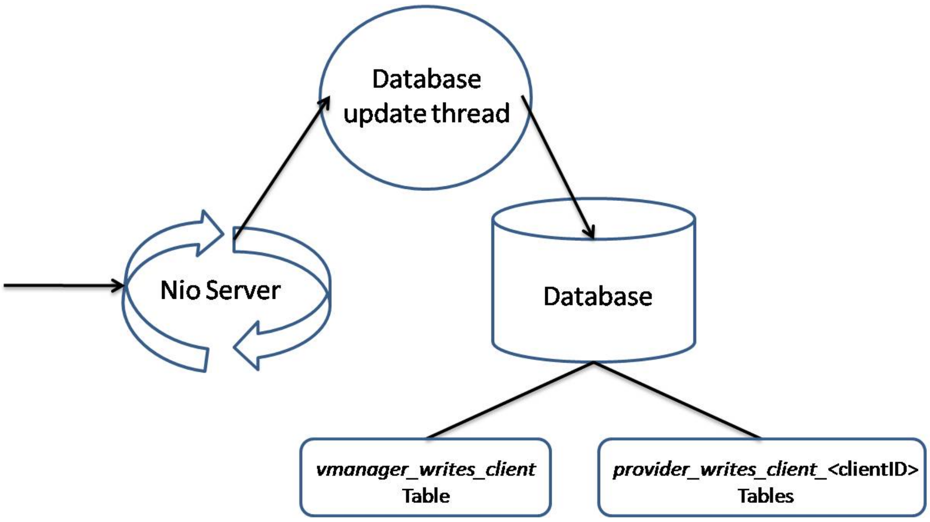

Figure 12: Data processing layer 1 - Storing data components

Database On each node, we have one running Postgres database server. The database contains the following tables:

- *vmanager_writes_client* - this table is created one per node. It contains all the data received from the version manager monitoring side. We have chosen this approach because the data sent for each client write consists in maximum one entry in the table (no entry in case of a WNP malicious client). In this way, we consider the amount of data can be of a reasonable size such as to fit without significant performance impact into one table. The structure of the table is as follows:

client_id	blob_id	version	watermark	blob_offset	write_size	page_size	timestamp
127.0.0.1	1	148	1250216125	0	1024	1024	1263136394497
127.0.0.1	1	149	1338336333	0	2048	1024	1263136554517
127.0.0.1	1	151	3234340499	0	1024	1024	1263136786556

Table 1: *vmanager_writes_client* table layout

- *summary_table* - this table is used in the next level of processing and we will discuss it there
- *provider_writes_client_<clientID>* - this table is created one per each client. It contains data received on the node about a certain client's writes on providers. We have chosen this approach because the number of pages within a write can be significant, and performing several writes by the same client has the potential to generate a large number of entries that need to be stored in the database. From the point of view of querying the database, it is more efficient to have this amount of information already grouped by some criteria (in our case, one table per client ID) rather than storing it in a single table. Querying several smaller tables is more efficient than retrieving it from one huge one. The structure of the table is as follows:

blob_id	timestamp	farm	node	watermark	page_index	page_number	page_size
1	1263136275142	Michi-IRISA	paralapeche	1372071065	0	1	1024
1	1263136534513	Michi-IRISA	paralapeche	3830431616	0	1	1024
1	1263138869310	Michi-IRISA	paralapeche	1201352078	0	1	1024
1	1263138969259	Michi-IRISA	paralapeche	227310002	0	2	1024

Table 2: *provider_writes_client_<clientID>* table layout

Nio Server The server is a Java process using features of the NIO (non-blocking I/O) library. It runs on a separate thread and it listens for connections on a user-specified port. Its implementation is based on the NIO tutorial located at http://rox-xmlrpc.sourceforge.net/niotut/ . The connection with the servers on the nodes is initiated by the MonALISA filter. Once the server accepts a connection and receives data on it, the request type is parsed and action is taken accordingly. Figure **??** shows the class diagram illustrating the interactions in which the server class takes part. There are 2 request types that we accept:

- Process monitoring data - upon receiving such a request, the server passes it to the 1st layer processing thread. Data is deserialized and put into a queue, from where the thread picks it up. This kind of request doesn't receive a response from the server.
- Gather user information - this request is served by the final processing layer, and it sends the list containing the summary information computed up to that point about each client's activity.

Processing thread The processing thread removes data from the server queue, and depending on its type, it inserts it into one of the tables described above. It also keeps a list of the updated clients, where it inserts client IDs whose information has been added into the database, and the timestamp at which the update was performed. This way, the next layer thread can only run its processing only on clients whose information has been updated, selecting the corresponding entries from the database.

4.3.2 Creating a summary of received monitoring data

The next step in obtaining the user activity information for user accounting is creating a summary of the large amount of information we expect to receive and store in the previous step. We need this because we finally want to obtain a small amount of information associated with each client, but that will characterize entirely its activity within BlobSeer.

The summary thread works as a timer task, executing its *run()* method at a specified time interval. Based on the updated clients cache collected within the previous step, we select the new entries inserted into the database for those specific client IDs. We keep a reference timestamp that we update for every timer tick. Because we run all the processing threads on the same node, the time reference will be the same, and we can compare the current timestamps with those in the table entries. Once the new information is obtained, we iterate through it and create one single entry per client write, comprising both provider data (all the pages that arrived from that write until that timestamp) and version manager data. This doesn't mean that we will only have one table entry per write, as some of the write data (such as part of the pages or version information) might arrive at a later time than other ones, and the timer interval might expire in the process. In a write-intensive scenario this kind of behavior is actually to be expected.

Within this thread, the *summary_table* information is filled. Its layout is the following:

client_id	timestamp	blob_id	watermark	provider_pages	write_size_vman	page_size
127.0.0.1	1263136281634	1	1372071065	1	0	1024
127.0.0.1	1263136401665	1	1250216125	2	2048	1024
127.0.0.1	1263136541695	1	3830431616	1	0	1024

Table 3: *summary_table* layout

A schematic of the interactions taking place at this level is depicted in Figure 13.

4.3.3 Computing client history

Our final layer abstracts into more general user activity information the data contained up to that point about a user in the system. This thread also has a timer attached, running periodically and each time updating a cache containing information about all the clients whose data is stored on the node.

The cache data associated with a client has the following entries:

- Number of IW writes
- Number of PNW writes
- Number of WNP writes
- Number of normal operations performed by the client

Before information new information is added to the cache, every operation is put on hold for a preconfigured period of time. We do this in case more information about that same

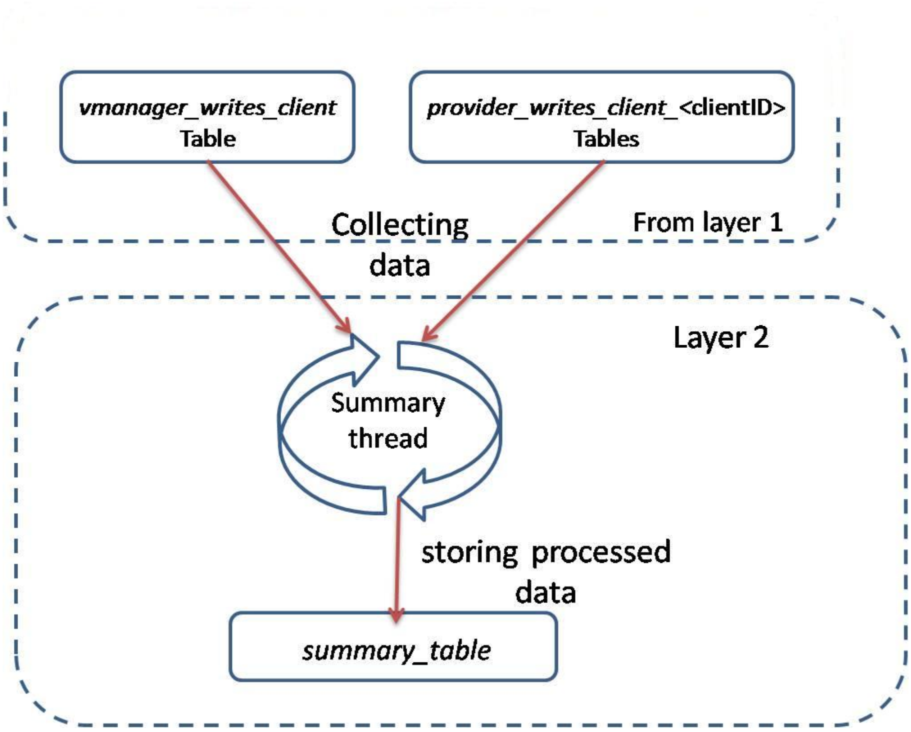

Figure 13: Data processing layer 2 - Creating a summary of the received data

operation may not arrive to the node all at once. In keeping data about a write pending for 3-5 seconds, we give all the information time to arrive within a reasonable interval and we prevent a correct operation from being classified as a different type.

The algorithm repeating for each timer tick is the following:

- The timestamp corresponding to the end of the operation are saved
- The entries with a timestamp newer than the last update time are selected from the *summary_table*
- For each new WRITE operation:
  - In case information some about this WRITE was already in our pending list, we update the pending entry with the additional information, but keeping the original expiry time.
  - If the information is not in the pending list, we update its timestamp to know when it was introduced, and we add it.
- Next, for each operation in the pending list, we check to see if it has expired. If that's the case, we check the type of the WRITE based on the size written to providers and that reported to the version manager: PNW, WNP, IW or a normal operation.
- If we already have an entry for this client in our cache, we update it. If we don't, we create a new object for the entry associated with this client ID.

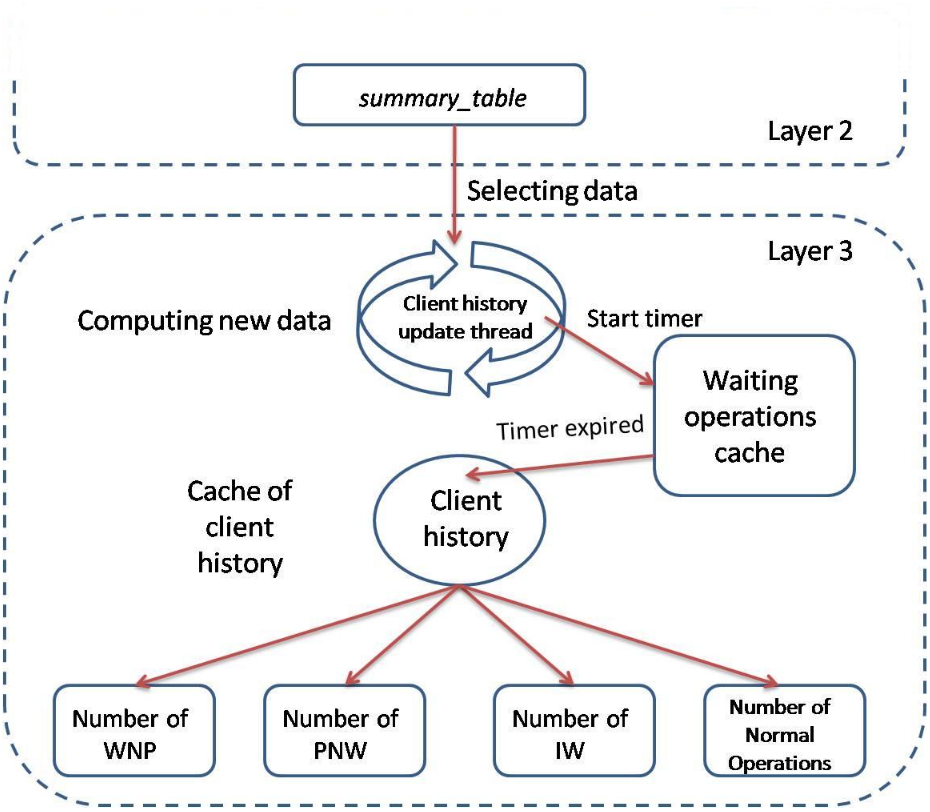

Figure 14: Data processing layer 3 - How the client history is being computed

In addition to that, it is this level that supplies information about client activity to the user accounting module. When the NIO server receives a request for collecting client activity data, it passes it to this thread, which serializes all the cache data and sends them back to the requester through the server's send method.

Figure 14 displays a summary of the processing taking place at this level.

4.4 User accounting - Computing malicious clients list

The module computing the malicious users list runs on a separate node, and uses the information about BlobSeer clients' activity gathered within the monitoring system. It consists of 2 running threads :

- Nio server thread - it is the same as for the distributed processing nodes. Its role is to listen for connections, to interpret requests and to pass them on to the other thread.
- Timer update thread - it has the role of periodically computing a list of malicious users, the black list we are interested in for our user accounting actions. This thread also receives as an argument the list of nodes that store the BlobSeer monitoring information, the same one that the MonALISA filter uses.
  - Every time the timer expires, it tries to connect to each one of them and request the data contained in each of the caches.

- The connection is initiated as blocking, because we need the results at the precise time we initiate the request, but with a timeout, in case one of the nodes dies or fails to connect for some reason.
- As new data is received, the information is added to a global cache list maintained at this level.
- After all the nodes are interrogated, we iterate through the computed global cache, we check each client's activity and, in case it contains some suspected operations, we add its ID to the black list computed.

5 Conclusion

5.1 Contribution

In this paper, we present our approach toward creating an efficient user accounting system for BlobSeer, based on monitoring. Our solution relies on a distributed architecture, and it offers an efficient mechanism for both monitoring, storing and processing the data related to user activity. The user accounting module is independent of the monitoring part, ensuring that the data received there could also be used for other purposes.

The goals of the paper were to use the set of specific data that can be collected from BlobSeer, and create a system processing them in an online manner, so as to provide an almost real-time image of the user activity in the system. We argue that our distributed processing system has no significant impact on BlobSeer performance and that it can be a viable alternative in implementing security policies within.

We proposed the use of multiple MonALISA services that share the load of collecting the data from the providers, and we implemented filters for this data at the service level. This way, data within a single WRITE is aggregated, in order to reduce in size the network traffic.

Because of the distribution algorithm at the filter level, data about one client is always sent to the same node. This is essential to the good working of the system; otherwise the client might be reported as malicious inconsistently. Our 3-layered distributed data processing system is a fast way of compressing and storing the data received.

The module computing the malicious clients is tuned for interacting with the processing system and gathering information about the clients from all the nodes.

5.2 Future work

The gathered user accounting can be the starting point towards an implementing a self-adaptation or a security mechanism for BlobSeer. A possible next step would be the implementation of the self-adaptation engine within BlobSeer, in such a way that the users determined as malicious to be notified to the provider manager. In case it receives further write requests from such users, it can block them by not returning the list of providers on which they can write data. We can also use the detection of malicious WRITEs to invalidate certain blobs or versions affected by such writes.

A more advanced mechanism concerning security within BlobSeer could involve the implementation of access rights on blobs in such a way that only part of the users are credited to access certain information. Also, we envision that users cand be offered different levels of quality of service. For example, we could allocate storage space and bandwidth differently based on a trust level that the user must gain during his activity within the system.

One of the improvements that the current system could benefit from would be a higher distribution degree of the processing that computes the black lists. The user scores could be computed in a distributed way, so that the module gathering the data has less computing to do and that the whole process would take less time.

References

[1] Joseph Barrus and Neil C. Rowe. A distributed autonomous-agent network-intrusion detection and response system. *Electronic Notes in Theoretical Computer Science*, 63:41–58, 2002.

[2] Alexandra Carpen Amarie, Jing Cai, Luc Bougé, Gabriel Antoniu, and Alexandru Costan. Monitoring the BlobSeer distributed data-management platform using the MonALISA framework. Research Report RR-7018, INRIA, 2009.

[3] Mark Crosbie, Gene Spafford, and Prof Gene Spafford. Applying genetic programming to intrusion detection. In *In Proceedings of the AAAI 1995 Fall Symposium series*, pages 1–8, 1995.

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

[5] I. Legrand, H. Newman, R. Voicu, et al. MonALISA: An agent based, dynamic service system to monitor, control and optimize grid based applications. In *Computing for High Energy Physics*, Interlaken, Switzerland, 2004.

[6] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? An architectural map of the cloud landscape. In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Philip K. Mckinley, Farshad A. Samimi, Jonathan K. Shapiro, and Chiping Tang. Service clouds: A distributed infrastructure for constructing autonomic communication services. *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 341–348, 2006.

[8] B. Nicolae, G. Antoniu, and L. Bougé. BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency. In *Data Management in Peer-to-Peer Systems*, St-Petersburg, Russia, 2009.

[9] The MonALISA Project. http://monalisa.cern.ch/.

[10] The Eucalyptus project: http://open.eucalyptus.com/.

[11] The Nimbus project: http://workspace.globus.org/.

[12] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA, 2005. ACM.

[13] S. E. Smaha. Haystack: an intrusion detection system. In *IEEE Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Orlando, FL., 1988.

[14] B. Tierney, R. Aydt, D. Gunter, et al. A grid monitoring architecture. Grid Working Draft GWD-PERF-16-3, August 2002. http://www.gridforum.org/.

[15] The Amazon Elastic Compute Cloud: http://aws.amazon.com/ec2/.

[16] Google App Engine: http://code.google.com/appengine/.

[17] Windows Azure Platform: http://www.microsoft.com/windowsazure/.

[18] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

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

[20] B. Weber, M.U. Reichert, W. Wild, and S.B. Rinderle. Balancing flexibility and security in adaptive process management systems. In *Proceedings of the OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 59–76, Berlin, October 2005. Springer Verlag.

[21] 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.