LanStore: a highly distributed reliable file storage system

Vilmos Bilicki
University of Szeged
Department of Software Engineering
6720 Szeged, Hungary
bilickiv@inf.u-szeged.hu

ABSTRACT

We need clever solutions that manage distributed network systems. LanStore is a highly reliable, fully decentralized storage system which can be constructed from already existing desktop machines. Our software utilizes the otherwise wasted storage capacity of these machines. Reliability is achieved with the help of a traditional erasure coding algorithm called the Reed-Solomon algorithm which generates *n* error correcting code items for each *m* data item. The distributed behavior is controlled by a voting- based quorum algorithm. These provide us with the capability of tolerating up to *n* simultaneously failing machines. As LanStore is intended to be used in LAN environments, instead of employing an overlay multicast solution we used an IP level multicast service. To use the bandwidth effectively, we designed a special UDP- based multicast flow control protocol. Our solution supports both IPv4 and IPv6. For the implementation platform we chose the Windows family and the .NET framework as they are the most popular platforms in offices and university departments. So far we have implemented a prototype version of this solution. We measured its performance and the results indicate that this solution can provide a throughput comparable to the currently used network file systems, its performance depending on the selected error correcting capability, the number of failing machines and the performance of the client machine. In special cases like video-on-demand with a high client number our solution can outperform the traditional single server solutions.

Keywords

distributed system, distributed storage, erasure codes, multicast

1. INTRODUCTION

In today's hectic world time is money and so is information. This is especially true nowadays with customer data from e-business and the huge amount of logistic and scientific data which may be worth their weight in gold. The amount of data is increasing sharply. The average storage capacity you get for your money is skyrocketing. Storage of several hundred GBytes is achievable for everyone. One might argue that today's storage capacity is just following the trends and there is enough cheap storage to meet the increasing demand.

.NET technologies '2005 *conference proceedings*,
ISBN 80-86943-01-1

Unfortunately, the total cost of ownership is also increasing sharply with the amount of the maintained data. In a typical company there are several file servers which provide the necessary storage capacity and there are many tape libraries for archiving the contents. If the storage need grows the company can purchase a new hard disk or a new server. To have a reliable system there is usually replication between the dedicated servers. The disk drives are organized in raid arrays, typically RAID 1+0 or RAID 5 [Che94]. This solution is not scalable enough for today's internet scale applications where there can be huge fluctuations in demand. Failsafe behavior versus effective storage capacity ratio is not optimal because of mirroring. Management is the other weak point of this system. That was why the Storage Area Network was designed. In a typical SAN there are several storage arrays that are connected via a dedicated network. The storage arrays typically contain some ten to sixty hard disks. To protect the data from hard disk failure these disks are organized into RAID 0, 1, 5 arrays. Protection from more two or more hard disk

failures is very costly because of mirroring. In larger systems it is vital to protect the data against storage array failure; hence the storage arrays are duplicated and connected by SAN switches. The servers are connected to this network via their fiber channel interfaces and provide a 2 GBit/s transfer capability. The scaling of this system is achieved by adding new hard disks to arrays, or moving the partition boundaries. The price of SAN components is high compared to typical network components and servers, and the storage usage failure toleration ratio is not so optimal.

We would like to present a much better and cheaper solution to this problem. A typical PC now has huge computing and storage capacity. It is not unusual to find more than 100 GBytes of storage capacity, over 500 MBytes of RAM and two GHz or more CPU clock frequency in a desktop PC. It seems that these parameters are constantly increasing. A typical installation of an operating system and the software required does not consume more than ten to fifteen GBytes. The rest of the storage space is unused. A typical medium-sized company has more than 20 PCs. A university or research lab usually has more than two hundred PCs. In this case the storage capacity that is wasted may be several TBytes in size. So it would great if we could utilize this untapped storage capacity.

In order to solve the above-mentioned problem we decided to design and implement LanStore with the following design assumptions:

- It is highly distributed without central server functionality.
- It has low server load. We would like to utilize the storage capacity of desktop machines; these machines are used when our software runs in background.
- It is optimized for LAN. The use of multicast and a special UDP based protocol is acceptable.
- It has effective network usage. We designed and implemented a simplified UDP-based flow control protocol.
- It is self organizing and self tuning. We used a multicast-based vote solution to implement the so-called ‘Group Intelligence’.
- There is a highly changeable environment. The desktop machines are restarted frequently compared to dedicated servers.
- It is a file-based solution. For effective caching we chose file-based storage instead of a block-based one. [Kis92]
- It has campus, research laboratory-type file system usage. Also, file write collisions are rare. [Kis92]
- It has an optimal storage consumption failure survival ratio. As a first approach we selected Reed-Solomon encoding for data redundancy.

2. OVERVIEW

In this article we would like to present our LAN-based distributed storage solution, which can work even when the node failure rate is high. In the next part we list and compare several existing solutions for distributed data storage approaches. In Section 4 we describe the main building blocks of our application. The dependence between these blocks and the design assumptions are also included here. Then Section 4.1 describes the data loss problem and the currently available solutions for it. We compare these solutions with our solution. Section 4.2 describes the network layer of our application and we show the features of our new simple multicast flow control algorithm. In Section 4.3 we present the core of our application, namely that of group intelligence. We show the goal of this layer and the solutions used. Next, Section 4.4 discusses our security layer with the features provided. Section 4.5 describes our data persistence layer. The design goals and the chosen solutions are also stated here. The implementation details are then described in Section 5. Finally, in Section 6, we present our results.

3. RELATED WORK

Distributing the contents among storage blocks is by no means a new idea. The oldest and the most popular technique is the RAID (Redundant Array of Independent Discs) technique [Che94]. It uses two basic data distributing solutions called stripes and mirroring. The first algorithm uses XOR parity data slices for correcting only one error while the second one can be used several times to achieve the necessary error correcting level, but the storage efficiency then sharply decreases. RAID is used typically for computers with several hard disks inside. The Zebra [Hart93] file system took the idea of striping from RAID, but instead of distributing the data among hard disks it distributes the data among storage servers. To effectively use the network bandwidth it uses per client striping instead of per file striping. The weak point of this solution is its single error correcting capability. Petal [Lee96] uses striping without redundancy and mirroring as a type of data distribution. One can define block level virtual disks with the aid of a low level interface. There are special server functions which translate the addresses used on a virtual disk to a physical machine

and disk addresses. It uses a heartbeat backbone to provide the so-called "liveness" property. A distributed consensus is achieved by using Leslie Lamport's Paxos [Lam98] algorithm. The goal of the Pasis [Wyl00] project was to create a solution for building a survivable data storage that was as simple as possible. Here is a thick client and thin servers. The only functionality implemented in servers is the data store which can be implemented as a simple file share, except that all this functionality is implemented on the client side. For the object name to physical location mapping, a directory server is used. In a later article [Wyl04] the authors of the Pasis framework define a new approach for handling Byzantine[Cas00]-type failures. In this solution the correction of failed storage nodes is a client task; there is no background process for consistency maintenance. This solution does not utilize the computing power of server nodes. Self*-store [Str00] is based on Pasis, its goal being to create a safe storage where, for a specified duration, there is no chance of data erasure. If the logfiles were stored in the Self*-store then the intruders would not be able to erase their footprints. OceanStore [Rhe03] defines a global scale storage system on a multicast overlay network. They use Tapestry[Zha01] for object naming and locating. To achieve data redundancy they use both erasure codes and mirroring. There are several defined classes of storage nodes with different responsibilities. For example the inner ring members have the task of data redundancy handling, but this solution is unsuitable in a laboratory where the storage nodes are desktop machines and they cannot tolerate a heavy processor load from a background process. FAB [Fro03] defines a storage system with a block level interface. The clients use SCSI commands for data manipulation whose implementation uses the thin client and thick server paradigm. This solution is unsuitable in an office or laboratory, however

4. ARCHITECTURE

Before going into detail let us see the high level workings of LanStore. As we mentioned before the main design goal was to gather the empty storage capacity into a virtual storage unit. To utilize in an equal way the storage capacity of the member nodes, we divided the files into equal fragments. In this way every storage node has the same number of stored data fragments. We would like to collect the free space from PC's in computer laboratories, classrooms, and so on.

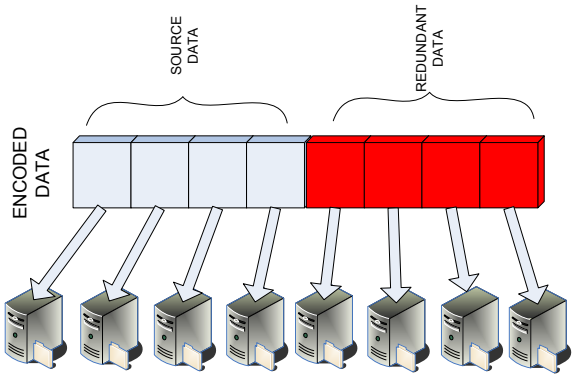

Figure 1

There is a high probability that one or more machines will be rebooted or turned off. We need data redundancy to correct the data which is stored on these machines. We will use forward error correcting codes (FEC) for error correcting. With the help of these algorithms we create *n* data fragments for *m* original data fragments. This means that we can reconstruct *n* failing data fragments. This process is shown in Figure 1. The consistency among modules is provided by a voting algorithm. If there are a critical number of working data nodes the remaining nodes may be reconstructed. Our solution is transaction based. At the end of a transaction a vote is taken and any changes are written to a permanent storage unit when the majority of nodes agree on the next common state. If there is no majority acceptance of the new state the transaction will roll back. After the changes are written into a permanent storage, a second vote is taken of the result. If there is a successful majority vote the whole task will be marked as fulfilled; if there is no successful majority result the first and the second transactions will roll back.

In our system the file is the basic data unit. We designed the file store for campus and research laboratory usage where file-based caching could be much more effective than block-based caching [Kis92]. The files are identified with the aid of the hash of their contents. With this solution we never store the same file twice. If someone tries to upload a file that already exists in our storage system, it creates a new link to the existing file. In the case of a modification, the storage uses versioning to handle the modifications. Our application is divided into independent modules. This design pattern provides an easy-to-maintain and robust code, where each module can be replaceable by another one using interfaces. The necessary functionality groups of our software provide us with natural borders among modules.

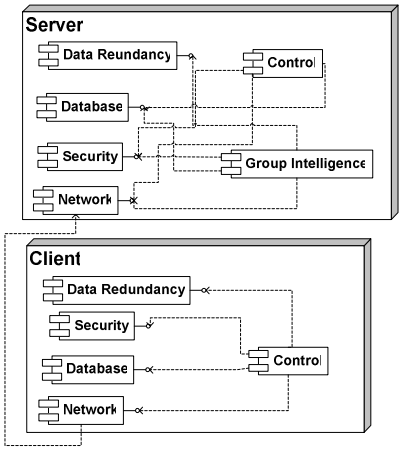

Figure 2

The modules are the following:

- Data redundancy module
- Network module
- Data persistence module
- Security module
- Group intelligence module
- Application logic module
- GUI module

Figure 2 shows the communication path between the modules. The control module is the core of our application; it uses the services provided by other modules. It is singleton, while every other module is thread safe. We may find that there are the same modules in the client and server sides, which contradicts our goal of developing an application with a fat client and thin server. During normal functioning the server does not use its Data Redundancy module. It only stores, sends the necessary data fragments and maintains its state with the help of the Group Intelligence module. We need the Data Redundancy module only for heavy data migration when every server helps a new or old server in an inconsistent state to achieve the consistent state.

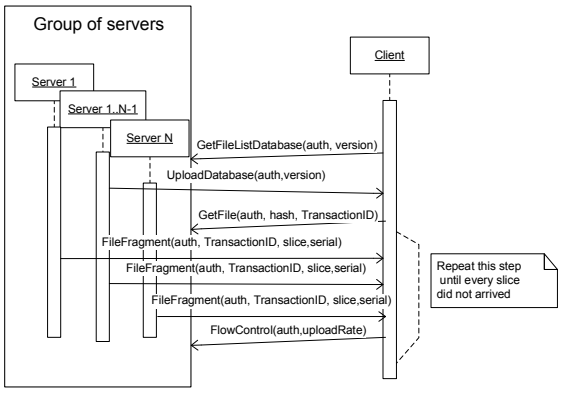

Figure 3

In Figure 3 the whole file download process is shown. First the client asks the group of servers via a multicast message for the altered data between its version and the global version of the directory/file database. We need this database on the client side to browse its contents. The designated server that was selected by the Group Intelligence module reacts and sends the recent changes. Next, the client starts a download process with the **GetFile()** multicast message. This message contains a transaction ID which is globally unique and it is generated from the hash of the file and the public key of the user. Every active server receives this message and starts uploading file fragments. During this upload process the client uses the flow control mechanism outlined in Section 4.2.

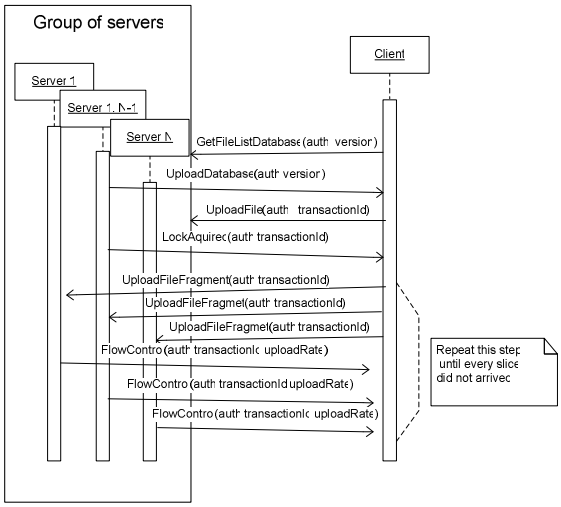

Figure 4

Figure 4 shows the file upload sequence. First the client sends a multicast message to the group of servers with the transaction ID. This step is needed to acquire a lock for the actual file. If there is no upload transaction with this ID the designated server sends it the right to modify. When the client receives this message it starts uploading file fragments to the servers. In the background a vote is taken among the servers after each slice upload. This mechanism is described in Section 4.3. There may also be a flow control between the servers and the client, which is mentioned in Section 4.2.

4.1 DATA REDUNDANCY MODULE

The task of this module is to provide the necessary data redundancy for error correction. Several approaches are available in the literature. The most popular one is that of data mirroring. This is an easy-to-use and implementable technique with low processing overheads but we pay the price on the storage consumption side. The creation of data parity blocks is another popular way, but apart from its optimal storage consumption this technique can

correct only one error at a time. This is a big drawback.

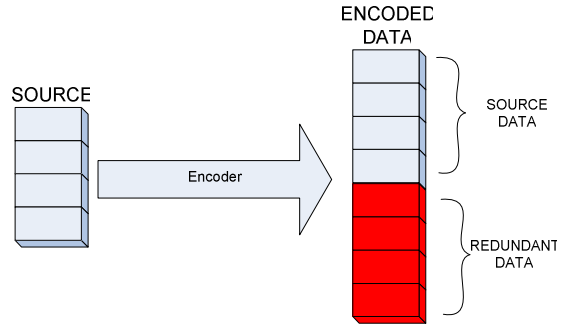

Figure 5

For our goal a special class of the forward error correcting codes FEC, the so-called erasure codes provide the best solution. Since we can detect failing data, we only have erasure errors. In the case of FEC codes one can select the required redundancy level and the algorithm generates the necessary error correcting data blocks for the existing data blocks (see Figures 5&6). If a data block fails, it can be calculated from the remaining data and error correcting blocs.

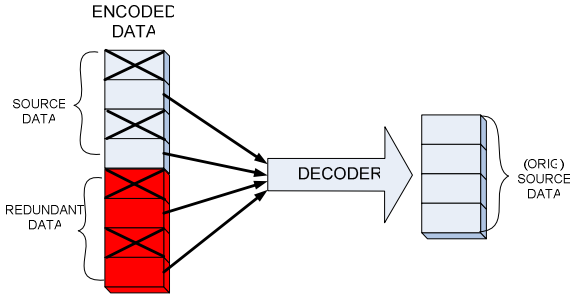

Figure 6

There are two types of FEC codes: codes with guaranteed error correcting capabilities and codes which have an error correcting capability with a given probability. We opted for the first code family because of its guaranteed error correcting capability. The price, however, is the processing overheads which depend on the selected error correction capability. This is one or two magnitudes higher than that for the second case. We chose a special case of the Bose-Chaudhuri codes called the Reed-Solomon [Riz94] code. The basic theory for this is quite straightforward: we have *n* data blocks and we need *m* data blocks to correct fewer than *m* erasure errors. To produce *m* data blocks we require a special equation system where every partial matrix is invertible. To produce such an equation system the Reed-Solomon approach makes use of the Vandermonde matrix. The Galois field is used as the space where the operations are performed. With this solution we replace the complex calculation-intensive operations by lookup tables. Here we use the Luigi Rizzo [Riz94] implementation of the Reed-Solomon code. The module divides the processed files into 64 KByte long stripes and calculates redundancy data for these slices. These stripes form the basic unit of the versioning system.

4.2 MULTICAST FLOW CONTROL

Our software is designed to run in a LAN environment. Most modern LANs are switched and there is practically a full mesh among network nodes. The key feature of such a network is that the bottleneck is on the source side or on the destination side; the network itself does not contain bottleneck nodes. TCP was designed and optimized for situations where the network is a black box and we can detect the available bandwidth only with the help of packet loss. There is an optimal windowing algorithm [Imr04], but this is not optimal when there is more knowledge and we can use a multicast protocol. We have complete knowledge of both sides of the communication channel, so it is plausible to use a flow control mechanism based on this. We designed a simple flow control mechanism that is capable of handling both multicast and unicast traffic. UDP here was used as a base and we added a simple signaling mechanism. Prior to each data manipulation process a transaction identifier is created by the client from the hash of the manipulated file and the public key of the client, this ID being unique to the whole system. At the same time only one client manipulates a file.

Our multicast flow control mechanism has two working modes, both modes utilizing the error correcting capability of our solution. In this way we can strike a balance between processor occupation and network transfer capability. The download mode operates during data transfer from a group of servers to a client. The upload mode operates during the data transfer from a client to a group of servers. In the following we will describe these modes.

```
Download mode:
1. Receive(fragment, stripeId, from)
2. IF(stripe is not yet processed)
3.         StoreFragmentInQueue()
4.         CheckQueue()
5. ELSE
6.         Drop(fragment)
7. END IF
8. IF(the Queue occupation is over 20%)
9.         SendFlowControlInformation()
10. END IF
```

Figure 7

```
CheckQueue function:
1. IF(there are more than N data fragments for the same stripe)
2.      IF(we have every data fragments)
3.              SendAlertToControler()
4.              SetTheProcessedFlag(stripeId)
5.      ELSE
6.              StartErrorCorretion(stripeId)
7.      END IF
8. END IF
```

Figure 8

In the download mode the client receives the file segments from servers and then stores these fragments in the input queue. If there are sufficient fragments for error correction (Figure 8, line 6) the client immediately starts the error correcting process. When it finishes the error correction, an alert is sent to the controller and it sets the processed bit for the processed stripe (Figure 8, lines 3&4). Further fragments for the processed stripes are dropped. With this solution we can avoid the situation where bottleneck nodes slow down the data transfer rate, and we can tolerate transparently the failure of nodes below a critical number.

In the upload mode our task is similar, namely that of tolerating the node failures and avoiding the situation where several slow nodes decrease the speed of the whole upload process. In this case after the first control packets the client starts sending the data fragments to different nodes as unicast UDP packets. When a storage node notices that the free space of its input queues is below 80%, it sends a control packet to uploading clients with a preferable transfer rate. The client has the responsibility of deciding whether it will accept the request or continue the upload with a higher speed. The decision of the client is based on responses from other storage nodes. It selects a speed which is acceptable for more than a critical number of storage nodes. The rest of the nodes will be corrected with the help of the Consistency process which is a part of the group intelligence.

4.3 GROUP INTELLIGENCE MODULE

In a distributed system this module plays a very important role. Its main task is to provide consistency, meaning a consistent state and consistent databases. In an ideal system where there are no failures this is not a hard task, but such difficulties arise when we have a real system. In the real world there is no algorithm that provides guaranteed consistency. To be able to handle this situation we define the following model of reality:

- The participants in the group management protocol can reboot or switch off at any time.
- The recorded data can never be overwritten.
- The messages must be delivered without delay or they will be lost.

With these constraints this module has:

- A voting-based algorithm for sequence upload verification
- A voting-based algorithm for file modifying finalization
- A voting-based algorithm for designated node selection
- Management of the correcting process of failed nodes

The voting algorithm is based on one by Leslie Lamports called Paxos [Lam98]. Every server node maintains a history database [Figure 9] that contains the successfully finished instructions. A data modification or upload is a sequence of stripe uploads which are a sequence of data fragment uploads. After every stripe upload a vote is taken of its success. If it was successful this fact is placed in the history database. After every data modification transaction (sequence of stripe uploads) a vote is taken of the success of the transaction. The success of a transaction really means that every sequence upload vote was successful. If a transaction was successful then every node erases the temporality signaling flag of the modified file. After this is carried out the new version of the file is the latest version.

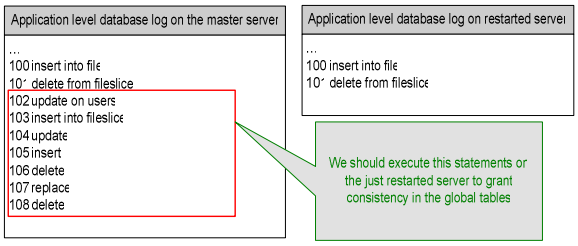

Figure 9

A designated node is important when the group of storage nodes sends messages to the client. This happens when a client asks for the new file list and about the success of file modification. The load of the processor, the occupation of the memory and the stability of the node are the properties which are important during the designated storage node election process. The designated nodes are changed after a few dozen transactions.

The correction of failed nodes is handled collectively; each consistent storage node is responsible for a stripe. The sequence of tasks needed to correct it is calculated using the data difference between the local

history table and the globally accepted one. To calculate the required data fragment these nodes act as clients. With this method we can achieve a relatively fast self-correcting capability of the group without imposing a high load on any given node. There are so-called synchronization points where a part of every history table in the system is the same. After reaching several such points the old records are deleted from the history table.

4.4 SECURITY

The security module has the task of providing data integrity, user and node authentication and access control. We store the digital certificates of nodes and users in the central database; the MD5 hash and the windows SID is stored here too. We use the existing Kerberos infrastructure for authentication when it is available. When there is no such infrastructure then we provide a simple asymmetric encryption-based authentication infrastructure. The data integrity of messages is guarded by digitally signing them with the sender's private key.

4.5 DATA STORAGE

The data storage module is responsible for data persistence and it has to maintain the history of conducted processes. The stored data can be divided into two main groups, the information which must be globally consistent and the information which has local importance (Figure 10). The Group Intelligence module maintains the consistency of globally important information.

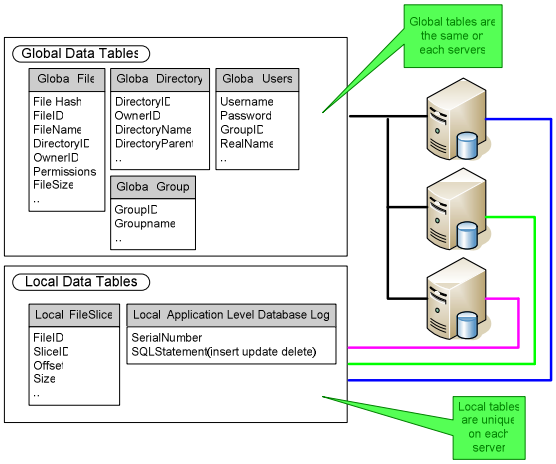

Figure 10

We store the following information:

- Metadata about data such as file name, path and access control lists.
- The data which is needed for the correct working of our system like users, nodes and certificates.
- The file fragments which have to be stored.
- A history of the processed instructions.

Every data type has its own behavior and therefore we selected different solutions for persistence. Meta data, infrastructure data, and histories are stored in a lightweight relation database. The size of this database never exceeds some 10 Mbytes. The fragments can be several hundred MBytes. We tested the handling of large objects in the current databases. We may conclude that the conventional file system has a speed about ten times faster for file fragments than current database solutions.

We implemented a version handling file storage. We store every version of a file. Between versions only the difference is stored. The basic unit of the difference handling is the file slice which was mentioned in the Redundancy module.

The goal of the history table was described in the Group Intelligence module.

5. IMPLEMENTATION

We selected the Windows platform because of its widespread usage in offices and university laboratories. Because it is well integrated in the Windows platform, .NET framework and the C# language was selected. For example it was very easy to check the infrastructure and the computing power of the hosting PC for leader election with the help of the Windows Management Instrumentation service. Another reason for using the .NET platform and managed code against the unmanaged C or C++ code was the short development cycle. Five graduate students have been working for a year on the software which is now in the alpha state. It has currently more than 20,000 lines of code. Figure 11 shows the detailed architecture. On the client side there are two threads: the Network module and the Client integration module. The network module has the task of capturing incoming packets and storing it in a synchronized queue. We designed this module to be as simple as possible to be able to capture every packet. The Client integration node consumes the packets from the common synchronized queue with the assistance of helper classes. If the queue is empty then the thread will go in the wait state. In this state the network module can wake it up with a pulse signal. In the case of file upload the GUI uses asynchronous method calls for each storage server. In this way outgoing traffic is handled in parallel. As the network module does not inspect the contents

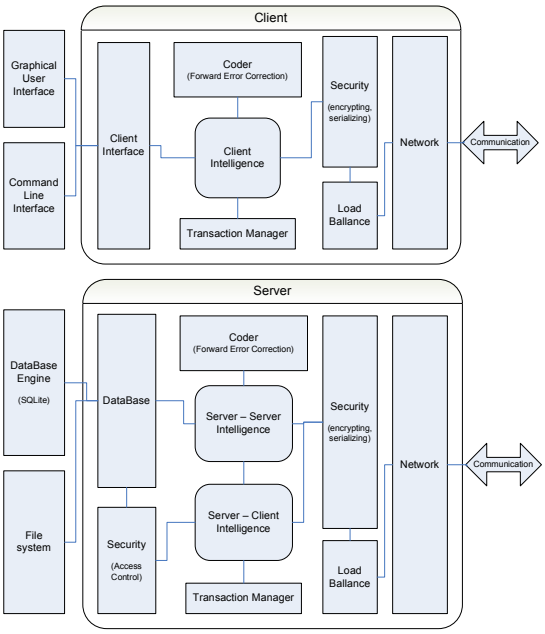

Figure 11

of the package and the packages could be encrypted with only one thread, the original client integration thread for handling the incoming will decode the packets and, if needed, wake up the appropriate sender thread for handling the output traffic.

The server side has a similar architecture, but instead of a GUI there is a database engine and a Server-Server intelligence module. These four threads are always running: the Network the Server-Server the Server-Client and the Hello thread. The first three threads work the same way here as on the client side. However, there are two queues; one for Server-Server and one for Server-Client module. The Network module makes a decision based on the type of the destination address of the incoming packet for selecting the appropriate queue. The Hello thread has the simple task of periodically sending hello packets. These packets act as keep-alive packets.

Owing to its speed, small size and easy-to-deploy capabilities, SQLLite was selected as the database engine. It has no transaction handling capabilities. When one tries more than one writing process simultaneously, it throws an exception. To avoid this, we used the .NET frameworks ReaderLock solution to achieve a serial access of this resource.

As we said earlier, we used the FEC encoder implemented by Luigi Rizzo [Riz94]. We use it as a native code.

6. EVALUATION

The raw encoding capacity with Reed-Solomon encoding was first measured. The results are shown in Table 1. We may conclude that the currently used processors produce a usable throughput for 64/32 (64 nodes, and out of these 32 contain error correcting information).

CPU Clock Frequency (GHz)	N	K	Throughput (MBit/s)
1	64	32	40
2	64	32	80
3	64	32	120
3	200	100	38.4

Table 1

To test the performance we used a laboratory with sixteen PC's, each having P4 3 Ghz processors, 1 GByte of RAM and a 100 MBit/s network adapter, while for debugging we used virtual PC's. We measured the throughput in different scenarios. Even in a larger configuration when there were 16 servers and we used a 16/8 redundancy scheme, the 100 MBit/s network bandwidth was the bottleneck. The processor utilization was only 20% on the client side, and less than 1% on the server side.

The above-mentioned measurements give a picture only about the raw coding capacity of a typical PC. Although this process is the most time-consuming part of the whole transaction, the remaining task could add significant delays. To be able to compare our solution with already exiting systems we tested our framework in different scenarios. One of the most accepted methods of file system testing is the Andrew benchmark [How88] which was created to measure the efficiency of the Andrew file system. This benchmark contains the following measurements:

- MakeDir
- Copy
- ScanDir
- ReadAll
- Compile

It measures the time needed for these tasks. Among these popular tasks the size of the manipulated files is important. The article [Cro98] estimates the distribution of file sizes of the UNIX file system as a Pareto distribution with parameters a=1.05 and k=3800. In another paper [Dou99] it was demonstrated that the windows file system file length distribution could be modeled with the help of a lognormal distribution and a tail with a two-step lognormal distribution. As a simple, but appropriate solution we chose the Pareto distribution to model the file size distribution of user homes.

Currently our system is accessible only through the GUI provided. We do not provide an API, so we cannot use the original Andrew benchmark script. In these circumstances we did the following and then took measurements: we created an application which

generates files with the length of Pareto [Cro98] distribution the depth of its directory path follows linear distribution. Each character inside the files is generated with a linear random distribution. We uploaded and downloaded the generated file/directory set with the help of the GUI. We used the Windows SMB file share as a comparison partner. A test network was set up with 10 PC's, each having P4 3 Ghz processors, 1 GByte of RAM and 100 MBit/s network adapter connected via a HP4108 switch as server nodes and a similar PC as a client node. The redundancy ratio was set to 7/3, so for every seven original data items three error correction items were generated. The following tasks were measured on the LanStore and on a Windows share which was one of the server nodes:

1. The delay of directory creation (a), and deletion (b) in seconds, with 615 randomly generated directories, with depth and name space of a random linear distribution. We executed this task on LanStore and on a Windows share system.
2. The delay of file upload (c) and download (d) in seconds and the throughput in MByte/second with 200 randomly generated files with the size distribution of Pareto(a=1.05, k= 3800) and with random hierarchy. The aggregate size of these files was 4.08 Mbyte.

We obtained the following results:

	Lanstore		Windows file share	
	Delay	Throughput	Delay	Throughput
a	353	-	5.3	-
b	116	-	3.8	-
c	213	0,02	3.5	1,25
d	53	0,08	6.1	0,7

Table 2

From these results we may conclude that for small files our system is about two magnitudes slower than the currently used network file systems. The reasons for this lie in the distributed nature of our system. In the current implementation every operation is handled in separated transactions and after every transaction a vote is taken of the success or failure of the transaction. As we have seen with small files or with administrative tasks like a directory tree manipulation, these overheads can take a longer time than the whole file upload. We can correct this behavior by batch processing the operations. When we upload a directory we can then assign a transaction for the whole process instead of managing every single operation as a transaction.

To test the framework as a video archive, we had to measure with different file size distribution. The video files are in most cases larger than normal files, so we used the value of 3,800,000 for k. With this value we generated 75 files with an aggregated size of 1.03 GBytes and the directory hierarchy was randomly generated. The test bench was the same as in the previous measure. We got the following results for file upload (e) and file download (f):

	Lanstore		Windows file share	
	Delay	Throughput	Delay	Throughput
e	262	4.02	144	7,32
f	240	4.39	104	8,5

Table 3

We can see that with larger files our solution provides a delay and throughput comparable to traditional network file systems. With batch processing this result can be further improved. In the case of a stabile environment we can achieve higher throughput than tradition file systems by sending the error correcting data fragments only when they are needed.

The data storage efficiency was measured as the ratio of the size of stored files and the size of data which is stored for every file. A record size in our database was about 35 bytes, which is not comparable to the stored data quantity. We may conclude that the data storage efficiency really only depends on the used error correcting level.

7. FUTURE WORK

So far the group intelligence module has only been partially implemented, but we plan to finish it later this year. We would like to implement the batch processing and client side caching to achieve a better performance for small files. To be able to modify the contents we need versioning, and we plan to implement this in early 2006. We would like to measure the performance in larger configurations with some 150-200 PC's. In the future we would like to use the LanStore as a basic building block for a wide area video-on-demand system and a long term archive for users' files. The current bottleneck is the FEC encoder; we would like to study the use of other solutions.

8. CONCLUSIONS

In this article we presented a solution for a cheap, reliable, high performance LAN based distributed storage. The solution components we used are not new but we could not find a system which is optimized for such circumstances. The measurements prove the usability of this solution even with current desktop computing capabilities. We think that in the near future with increasing processor capacity similar solutions will be widely used.

9. ACKNOWLEDGEMENTS

The author would like to thank Tibor Antal, Peter Bagrij, Tamas Horvath, Andras Maroti, and Kornel Kallai for their creative ideas and hard work, Tibor Csendes for his useful comments and suggestions, and David P. Curley for checking this article from a linguistic point of view.

10. AVAILABILITY

The source code, the binaries, the detailed benchmarks and the tool for benchmarks will be published and be freely available at the following website: http://nlab.inf.u-szeged.hu/lanstore

11. REFERENCES

[Hart93] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. Operating Systems Review – 14th ACM Symposium on Operating System Principles, 27(5):29–43, December 1993.

[Che94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. ACM Computing Surveys, 1994.

[Kis92] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. ACM Transactions on Computer Systems, 10(1):3–25, February 1992.

[Lee96] Edward K. Lee and Chandrohan A. Thekkah. Petal: distributed virtual discs. SIGPLAN Notices, 31(9):84-92,1-5 October 1996.

[Lam98] L. Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16 (2), pp. 133-169, May 1998

[Wyl00] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. IEEE Computer, 33(8):61–68, August 2000.

[Wyl04] J. J. Wylie, G. R. Goodson, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In Int. Conf. on Dependable Systems and Networks (DSN), Florence, Italy, June 2004.

[Cas00] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In Proc. of OSDI, 2000.

[Str00] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[Rhe03] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In Proceedings of the Conference on File and Storage Technologies (FAST), 2003.

[Zha01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph," Tapestry: An infrastructure for fault-tolerant widearea location and routing," UC Berkeley, Tech. Rep. UCB/CSD-01-1141, April 2001.

[Fro03] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise Storage Systems on a Shoestring. In 8th Workshop on Hot Topics in Operating Systems (HOTOSVIII), Kauai, HI, USA, May 2003.

[Imr04] Cs. Imreh, V. Bilicki. On the optimization models of congestion control. In XXVI. Operational Research Conference. Gyor, Hungary, May 2004.

[Riz94] Luigi Rizzo, Effective Erasure Codes for Reliable Computer Communication Protocols. ACM Computer Communication Review, VOL 27, pp. 24-36, 1997.

[How88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, ``Scale and performance in a distributed file system," Transactions on Computer Systems, Vol. 6, pp. 51-81, February 1988.

[Cro98] M. Crovella, M. Taqqu, and A. Bestavros. Heavy-Tailed Probability Distributions in the World Wide Web. Appears in the book: A Practical Guide To Heavy Tails: Statistical Techniques and Applications, R. Adler, R. Feldman and M. S. Taqqu, editors, Birkhauser, Boston, 1998.

[Dou99] J. R. Douceur and W. J. Bolosky. A large-scale study of filesystem contents. In ACM SIGMETRICS'99, ps 59–70, May 1999.