SODA: A Lease-Based Consistent Distributed File System

Fabio Kon [*] *Arnaldo Mandel*

Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo – Brazil
E-mail: {kon,am}@ime.usp.br

Resumo

Apresentamos um novo modelo para a análise da carga gerada pelo protocolo dos *leases*, um protocolo que garante a consistência de informações cacheadas em sistemas distribuídos. Através do modelo, comparamos a carga produzida por este protocolo com a produzida pelo protocolo adotado pelo sistema de arquivos do SPRITE – que também garante a consistência das informações cacheadas. Mostramos a superioridade do protocolo dos *leases* sob uma larga gama de valores para os parâmetros do nosso modelo.

Em seguida, descrevemos o SODA (Sistema para Operação Distribuída de Arquivos), que utiliza uma extensão do protocolo NFS com a inclusão de *leases*. Apresentamos detalhes de uma implementação do SODA sobre o sistema operacional LINUX. Este exemplo mostra que o SODA pode ser implementado sem muito esforço em qualquer sistema utilizando o código do NFS como ponto de partida.

Finalmente, apresentamos resultados de testes coletados no SODA e os comparamos aos resultados obtidos em um simulador do protocolo do SPRITE.

Abstract

We present a new model for the analysis of the load produced by the lease protocol, a protocol wich assures the consistency of cached information in distributed systems. Using this model, we compare the load produced by this protocol and that produced by the protocol adopted by the SPRITE distributed file system – which also guarantees the consistency of cached information. We show the superiority of the lease protocol under a large range of our model parameter values.

We then describe the SODA consistent distributed file system wich uses an extension of the NFS protocol by addition of leases. Details are shown of an implementation of SODA in the LINUX operating system. The example shows that starting up with the NFS code it should not be hard to implement SODA in other systems.

Finally, we present some SODA performance evaluation results and compare them with results obtained in a SPRITE protocol simulator.

[*] During this research the first author received a Master's scholarship from CNPq. This work was also supported by FAPESP (process # 93/0603-1).

1 Introduction

Contemporary Distributed File Systems make extensive use of file caching both on client and server sides. Caching of files in the server physical memory avoids a significant number of accesses to disk. On the other hand, file caching on clients purports to decrease the use of the network. Consequently, the network load is lowered and a faster file service is provided.

However, the use of client caching introduces the problem of maintaining the consistency among the several copies of a file which is accessed by more than one client. When one client updates a block of a file stored in its local cache, it would be interesting that the system could guarantee that subsequent accesses to the same block made by other clients could perceive the recent modifications. The greater is the network scale, the harder is to reach this goal.

Existent file systems applies different sort of policies regarding the semantics of file sharing. SUN's NFS [SUN90] does not offer any type of guarantee that shared files will be seen consistently by different clients. When a file is updated by one client, this modifications may not be noticed by other clients during a period of up to 6 seconds. When a file is created or deleted, this fact can take up to 60 seconds to be perceived by other clients. If one needs a coherent sharing of information throughout the distributed system, some other mechanism – like message passing – must be used.

The ANDREW File System [Sat90], on the other hand, applies what is called the *session semantics*. Under the session semantics the updates made to a file by one client can only be perceived by clients that open this file after the moment when the first client has closed it.

The SPRITE Network Operating System [NWO88] presented a solution to the problem of maintaining strict coherency among the copies of a file in several client caches. SPRITE disables the client cache when a file is concurrently shared by more than one client and at least one of these clients has the file open for updates. This kind of situation is called *concurrent write sharing*.

The problem with the SPRITE approach is that it requires that each time a file is opened or closed the client must notify the server of this fact, thus increasing the network load. Besides, when a file is concurrently shared with updates, every client query must be treated directly by the server through the network.

In many real networks that is not a problem since concurrent write sharing rarely occurs. However, when file sharing is more frequent a better protocol is required. In this paper we will discuss *leases*, a mechanism to assure consistency on a distributed system.

Section 2 describes the lease protocol. A new analytical model of the behavior of the protocol is presented in section 3. Finally, section 4 describes SODA, our implementation of the protocol in the LINUX operating system.

2 Leases

The lease mechanism was first proposed by Gray [GC89]. The designers of the ECHO [MBH$^+$93] distributed file system, however, claim to have developed the lease concept simultaneously and independently of Gray. In fact, ECHO was the first implementation of leases used by a large number of users. Due to problems involving the project where ECHO was inserted, it ceased to be used in the summer of 1992.

2.1 The Protocol

A lease is a contract that assures the right of property to some good during a fixed period of time. Let us see how this concept is applied to distributed file systems.

After sending a read request to a server, the client receives not only the data requested but also a lease which is a guarantee that the server will not update that data without the permission of the client possessing the lease. Every lease is valid for a limited period determined by the server. Indeed, what the server sends to the client, besides the file data, is the instant when the lease will be expired.

If a client application requests a read from a file that is locally cached, the client operating system must be sure that the lease he has for that file is still valid. If the lease is valid, the application can receive the requested data without any contact with the server.

On the other side, if the lease is not valid anymore, the client must send a message to the server to check whether the local version of the data is the most recent one. Should the cached data be out of date, the new data must be fetched from the server.

When the server receives an update request for a file, it cannot confirm the update immediately. Before committing the update, the server must gain the agreement of all clients that possess a valid lease for this file. The server can commit the update and return from the client request only after all the clients which have a valid lease for this file have agreed with the update or after the expiration of the leases of the clients which have not replied.

When a client receives a request for update agreement from his server, it marks its lease as expired. If, after that, this client needs to read the same file again, the new version must be fetched from the server.

Leases may be used not only to maintain the consistency of the file contents but also to maintain the consistency of meta-data like file attributes and directory and location information. When this sort of information is cached, leases can be used to control its coherency.

We must notice that this mechanism assures the consistency of the cached data only if a write-trough police is adopted, i.e., the write requests are not cached, they are sent directly to the server and the thread that requested the write is blocked until its completion.

It is possible to maintain the consistency using leases even with write-behind[1] but, in this case, the protocol becomes more complex.

In order to use leases and write-behind, one must use two types of leases. A *read lease* would be similar to the one just described. However, a *delayed-write lease* would provide a client the possibility of writing to its cached data and updating the server asynchronously.

Before giving a delayed-write lease for any client, the server must be sure that no other client has a lease for the same file.

[1] When write-behind is adopted, the write requests made by the client applications are cached. The requests to the server are delayed and the application thread is not blocked.

On the other hand, before providing a client with a read lease, the server must check if any client has a delayed-write lease. If such a client exists, the server must ask the client to flush its dirty data and invalidate its lease. Only after receiving all dirty data from the client or after the lease expiration time the server may send the requested read lease.

Sometimes, the management of these leases may produce a significant overhead resulting in poor performance. In this cases, the best solution is simply disabling some part of the client cache.

From now on, we will only treat the case where write-through is adopted. In order to get a good performance under this policy it is important that most of the temporary files be stored locally and not in remote servers.

2.2 Fault Tolerance

One of the main advantages of the lease protocol is its fault tolerance. Differently from the ANDREW and SPRITE protocols, the lease protocol is fault tolerant[2] on his own. If each lease is valid for a period shorter than the time required by the server to reboot, no extra mechanism is required to provide fault tolerance.

When a server crashes in ANDREW or SPRITE systems, a lot of important information about the state of the system is lost. Under the lease protocol, the only information lost is that about the clients which have valid leases. But if the time required by the server reboot process is longer than a lease lifetime, then no relevant information is lost.

On the other hand, if the server receives a write request while the network is partitioned, all the server must do is to follow the protocol, it delays the write until every lease owned by an inaccessible client has expired.

In the client side, if the communication with a server is lost, no special action must be taken. The client just uses its cache data while its leases are valid. When the leases expire, it must keep asking for new leases until the server replies.

In addition, the cache availability in a lease based system is better than in the SPRITE system. In the latter, if a client which has a file opened for update crashes, then no other client will have permission to access the file until the server becomes aware that the first client has crashed. This may take a long time. In the lease case, this problem does not exist.

2.3 Lease Term

The major factor in the performance of a lease based system is the extent of the period while the lease is valid, i.e., its term.

If the leases are valid for a short period, the necessity to revalidate them is greater. On the opposite side, if the leases last a long period, the necessity to invalidate them when the updates occur is greater.

Too long leases tend to be a bad choice in the presence of client crashes and network partitions. In these cases, the updates must wait a longer period to be committed. Besides, if the lease term is greater than the time required by the server to reboot, then some mechanism to make the lease information survive server crashes is needed.

[2] We are considering just non-Byzantine faults here.

3 An Analytical Model

Gray [GC89] has presented a simple analytical model for measuring the server load and the file service delay associated with the lease protocol. We have extended and modified his model in order to more accurately represent the extra network load produced by the consistency maintenance.

Our model counts the number of messages used to maintain the consistency of a single file provided by a single server. Figure 1 presents our model parameters.

N	number of clients accessing the file
R	per client read rate
W	per client write rate
t	lease term

Table 1: Model parameters

We suppose that N clients request reads and writes following a Poisson distribution with per client rates R and W respectively.

Let us first measure the portion of time in which a client possesses a valid lease for a specific file. The figure 1 shows some periods where the client has a valid lease – labeled **L** – and periods where it does not have a valid lease – labeled **T**.

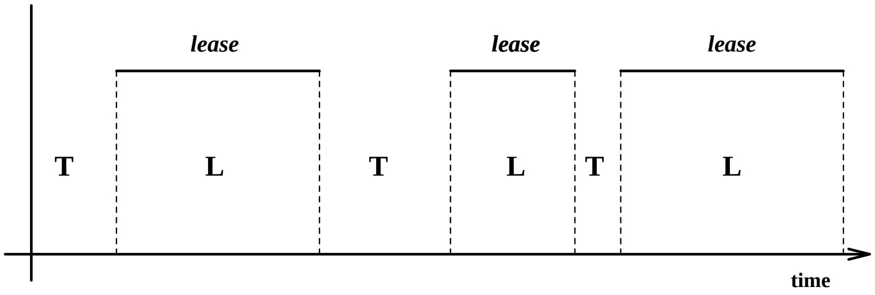


Figure 1: Leases in one client

A known result of the Theory of Reliability[3] assures that the relative portion of time in which the client has a valid lease is, on average,

$$\frac{E(L)}{E(L)+E(T)}$$

where $E()$ denotes the expected period extent.

A **T**-period starts when the lease expires and ends when the next read is made. Poisson processes does not have memory, i.e., the future process behavior does not depend on the

[3] See [BP81], section 7.2

past. So, the expected time until the next time is always the same, the inverse of the read rate:

$$E(T) = \frac{1}{R}.$$

The **L**-periods start when the client receives a lease and end in the next write or after t units of time. In order to estimate $E(L)$ we may imagine that during the **L**-periods a superposition of two Poisson processes occurs. The first, with rate NW represents that the lease may be canceled by a write requested by any of the N clients. The second represents normal lease expiration after t units of time, hence has rate $\frac{1}{t}$.

The resulting process rate is the sum of the above rates. Therefore, the expected value for the **L**-period extent is the inverse of this rate:

$$E(L) = \frac{1}{NW + \frac{1}{t}} \quad .$$

So, the portion of time in which a client possesses a valid lease is

$$\frac{\frac{1}{NW+\frac{1}{t}}}{\frac{1}{NW+\frac{1}{t}} + \frac{1}{R}} = \frac{Rt}{1 + Rt + NWt} \quad .$$

If we suppose that the N clients access the file independently, then the expected number of clients sharing the file at a given moment is

$$S = \frac{NRt}{1 + Rt + NWt} \quad . \qquad (1)$$

3.1 The Cost of Leases in the Reads

While a lease is valid, a client serves $RE(L)$ read requests through its cache excluding the read which produced the lease request. So, the cost (2 messages) of giving a lease is amortized within $1 + R\,E(L)$ reads. Therefore, the rate of messages related to lease concessions to the N clients – or the cost of leases in the reads – is

$$C_R = \frac{2NR}{1 + R\,E(L)} = \frac{2NR}{1 + R\frac{1}{NW+\frac{1}{t}}} = \frac{2NR(1 + NWt)}{1 + Rt + NWt} \quad . \qquad (2)$$

3.2 The Cost of Leases in the Writes

When the server receives a write request, it must invalidate the leases of the clients which still have a valid lease for the file. Since it does not have to invalidate the lease of the client which requested the write, it has to invalidate

$$S - \frac{S}{N} = \frac{S(N-1)}{N}$$

client leases[4].

If the network where the lease protocol is implemented supports multicast than the server must send one invalidation message and wait for $\frac{S(N-1)}{N}$ replies each time a write is requested.

Should the network not support multicast, the server must send $\frac{S(N-1)}{N}$ messages and wait for the same number of replies. Therefore, the cost of leases in the writes is

$$C_W = \begin{cases} (1 + S\frac{N-1}{N})NW = NW + S(N-1)W & \text{(multicast case)} \\ \\ 2S\frac{N-1}{N}NW = 2S(N-1)W & \text{(no multicast case)} \end{cases} . \tag{3}$$

3.3 The Lease Protocol Total Cost

From 2 and 3, we see that the total cost of the lease protocol is

$$C_{total} = \begin{cases} \frac{2NR(1+NWt)}{1+Rt+NWt} + NW + S(N-1)W & \text{(multicast case)} \\ \\ \frac{2NR(1+NWt)}{1+Rt+NWt} + 2S(N-1)W & \text{(no multicast case)} \end{cases} . \tag{4}$$

3.4 Comparing with the SPRITE Protocol

In order to compare the load produced by the SPRITE protocol and that produced by the lease protocol, we will consider the case of concurrent write sharing of a file. When a file is not concurrent write shared, both protocols tends to present a good performance.

Under concurrent write sharing the SPRITE clients must contact the server each time a read is requested. The client sends a message for the server and the server send it one reply. The writes are sent directly to the server as in our lease model, so we will not consider them here. So, let us consider that the total traffic related to consistency maintenance is $2NR$.

Therefore, formula 4 assures that, under concurrent write sharing, the lease protocol generates a lower load than SPRITE protocol if and only if

$$\frac{2NR(1+NWt)}{1+Rt+NWt} + NW + S(N-1)W < 2NR \tag{5}$$

if the network supports multicast and if and only if

$$\frac{2NR(1+NWt)}{1+Rt+NWt} + 2S(N-1)W < 2NR \tag{6}$$

if the network does not support multicast.

Applying the S value given by (1) to (6) we get the following condition for the leases superiority in the case with no multicast:

$$\frac{R}{W} > (N-1) \tag{7}$$

[4] We are subtracting from S the probability of the client which had requested the write having a valid lease for the file.

So, we can see that when the read rate is sufficiently larger than the write rate – when (7) is satisfied – the lease protocol is a good choice. Although, when the write rate is so large that (7) is not fulfilled the best solution is to disable the cache as SPRITE does.

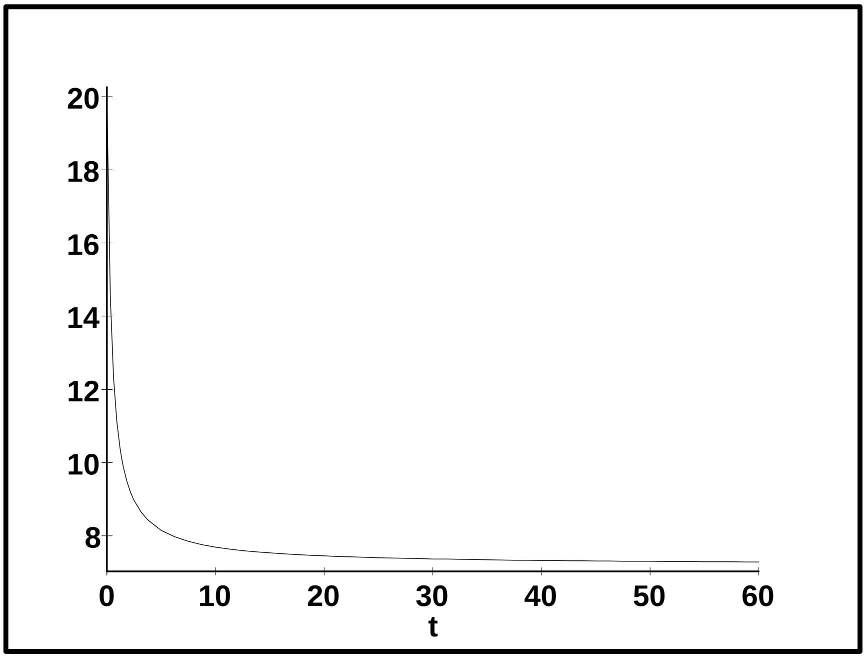
In the multicast case, it follows from (1) and (5) that the lease protocol is a better choice if and only if

$$R > \frac{NWt + \sqrt{N^2W^2t^2 + 8\,(NW^2t^2 + Wt)}}{4t} = \frac{NW}{4}\left(1 + \sqrt{1 + \frac{8}{N}\left(1 + \frac{1}{NWt}\right)}\right) \quad .$$

As long as some regular writing is going on (say, at least one write every two lease periods, so $NWt > 1/2$), and N is not too small either (take $N \geq 6$), the condition above is satisfied if $R/W > 0.8N$. Therefore, multicast should improve the odds of leases being better than SPRITE. However, due to the lack of an appropriate testbed for this version of the protocol, the remaining analysis considers only the case where no multicast is available.

3.5 Model Estimates

Figure 2 shows how the number of messages produced by the lease protocol depends on the lease term under the no multicast case. The graph was made considering 5 clients requesting, on average, one write in each 10 seconds and two reads per second.



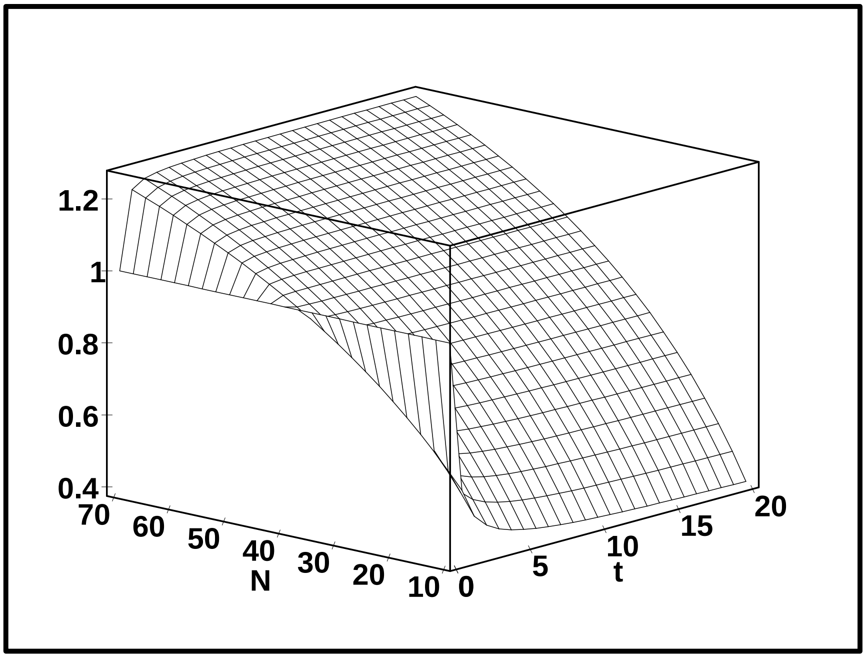
$N = 5, R = 2$ e $W = 0,1$

Figure 2: Number of messages produced by the lease protocol

We may see that there is no optimal value for the lease term. The longer the lease duration is, the lower is the load produced by the protocol. Therefore, long leases are better choices. The only limitation to the lease extension are the disadvantages of too long leases described in section 2.3.

On the other hand, we can see from figure 2, that 60 second leases do not provide any significant gain compared to 20 second leases. So, in this example, adopting 20 second leases would be a good choice.

Figure 3 presents the ratio between the load produced by the lease protocol and that produced by the SPRITE protocol. When the value in the vertical axes is below 1, the load produced by the lease protocol is lower than the load produced by the SPRITE protocol.



$R = 2$ e $W = 0,05$

Figure 3: Scalability

We can see that, while $N < \frac{R+W}{W}$ ($N < 41$ in this example), increasing the lease term effects a lowering of the load produced by the lease protocol. When $N > 41$, the lease load is greater than the SPRITE load and longer leases produce a higher load. This is the point where the cache must be disabled.

However, our model assumes that all the N clients are writing to the file. But, since it is not a common situation to have tens of clients writing concurrently to the same file, we may consider the scalability of the lease protocol as being good.

4 The SODA Distributed File System

In order to study the behavior of a lease based system on a real network, we developed the first version of the SODA consistent distributed file system [Kon94] during the first semester of 1994. SODA was implemented on the LINUX operating system and its code was written using the LINUX NFS 2.0 code. Since the SODA protocol is an extension of the NFS protocol, it can be implemented in any other system using the NFS code as a starting point.

4.1 Implementation

Three main modification to the LINUX NFS were made:

1. The LINUX NFS does not implement client caching. So we had to create the data structures and the functions wich are responsible for the cache maintenance inside the client kernel.

2. NFS servers are stateless but the lease protocol requires that the server store information about the clients which have valid leases. Therefore, we had to create data structures and functions to manage this information on the server side – which is the daemon process *nfsd*.

3. Differently from NFS, the lease protocol requires that the server send messages to the clients and wait for their replies. In order to carry this, we had to introduce a new daemon process called *sodad* in the client side. This process receives the lease invalidation requests from the servers, makes a local system call to invalidate the leases and replies to the appropriate server.

Figure 4 shows how our system works. In this example, the server attends a read request from client 1 and, while the client 1 lease is still valid, a write request from client 2.

In the beginning, the process *p1* executes a `read()` system call in order to read some bytes of a certain file `(1)`. Its kernel checks its local tables and finds that this file is managed by a remote server. Since it does not have a local copy of the requested bytes in its cache, the kernel sends a read request to the appropriate server using a RPC `(2)`. This RPC is received by *nfsd* which forward the request to its local kernel `(3)` which accesses the local disk if necessary.

After receiving the bytes from the kernel, *nfsd* returns the RPC sending the client not only the requested bytes but also a new lease for this file `(4)`. Then, the client 1 kernel copies the bytes just received to its local cache, updates its lease table and returns the system call with the requested bytes `(5)`.

If, while this lease is still valid, a process *p2* in other machine requests a write to the same file `(6)`, then the following occurs.

After receiving the write request, the client 2 kernel finds that the file is remote located and forwards the write request to its server using a RPC `(7)`. The server *nfsd* receives the write request and looks for leases for the same file in its lease table finding that client

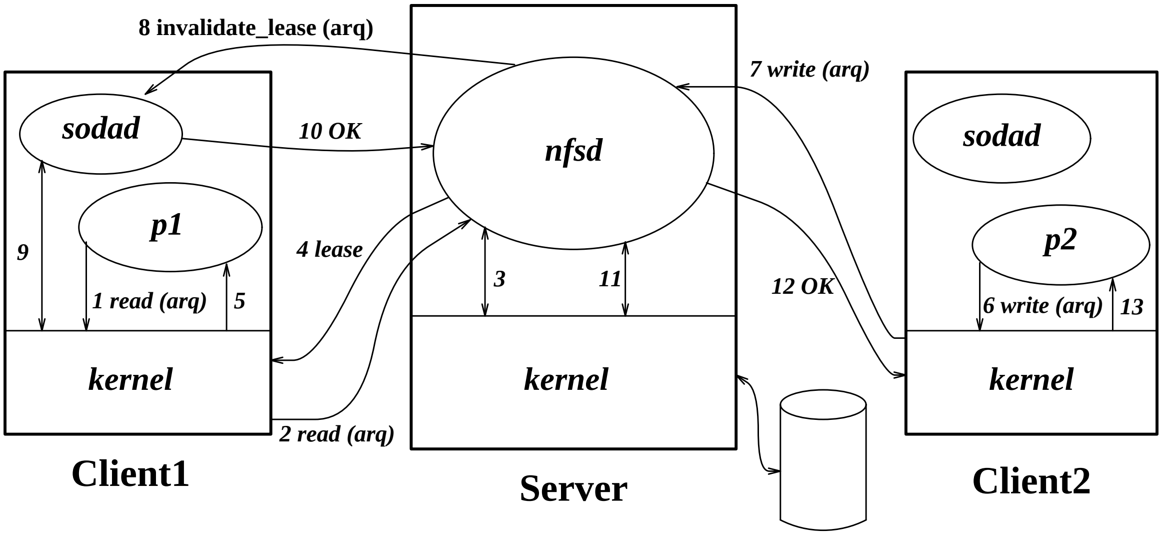


Figure 4: A read and a write request

1 possesses a valid lease. At this moment, it sends a lease invalidation message to the *sodad* process at client 1 (8).

When *sodad* receives the invalidation request, it executes the `invalidate_lease()` system call (9). This system call marks the lease as expired in the kernel lease table. Upon completion of the system call, the client replies to the server corroborating the lease invalidation (10).

Only after receiving client 1 response, the server can call the local `write()` system call to commit client 2 request (11) and then the RPC can return (12) with the result of the request.

Finally, client2 kernel receives the result of its remote write request and returns the same result to process *p2* (13).

4.2 Performance Results

In order to evaluate our system, we made some tests using three 486 and one 386-based PCs all of them running LINUX 1.0.9 and our current version of SODA. This machines were connected to a 10Mbit Ethernet network shared by a lot of workstations distributed across our Institute. The tests were made during low network load periods.

In this environment, the read requests attended by the client cache could be completed at least 13 times faster than a read attended by the server through the network.

On the other side, a read attended by the server in a SODA system is, on average, 20% slower than one in a standard LINUX NFS system. This overhead, caused by the cache maintenance procedures, is small enough to let the SODA system provide a faster service under many different conditions making extensive use of client caching.

The influence of the lease term can be seen in figure 5. This figure shows the server load produced by three clients accessing 10 files, each one with rates $R = 2$ and $W = 0.01$. The graph shape is similar to that predicted by the analytical model.

In order to compare the protocol adopted by SODA and the protocol used by SPRITE

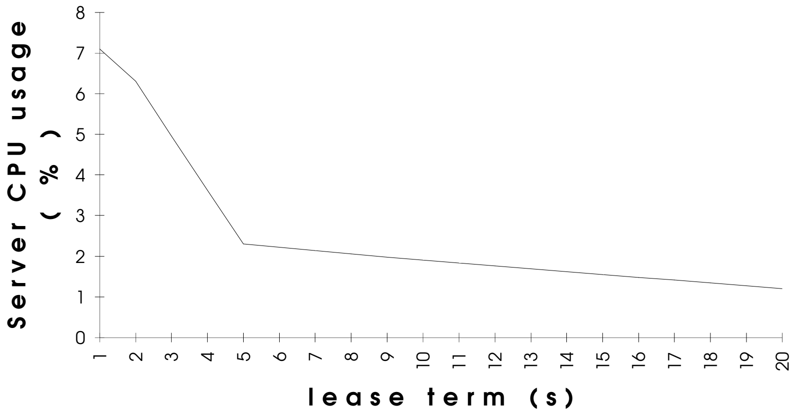


Figure 5: CPU load × lease term

under concurrent write sharing, we modified our client kernel to check the file version number with the server each time a file is read as SPRITE does. We will call this system simulated SPRITE, or just, sSPRITE.

Figure 6 presents the average time to read 1Kbyte of data both in SODA and in sSPRITE. The test was done with 3 clients and the read rate was fixed on one read per second. The write rate varies from 0 to 0.7 writes per second.

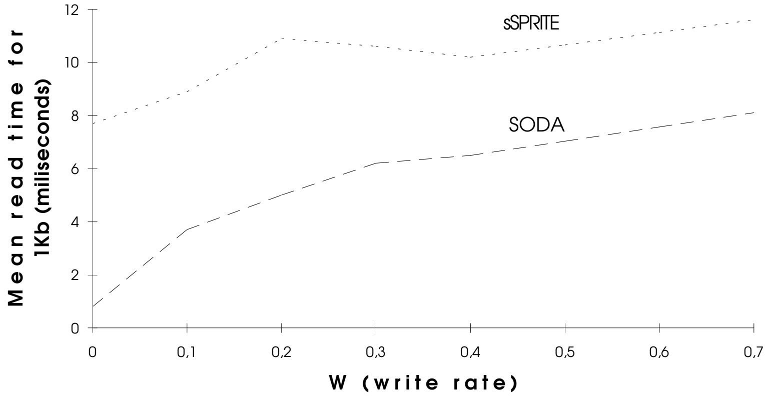


Figure 6: Elapsed time to read 1Kbyte

We may see that when W is relatively low, SODA is many times faster than sSPRITE and both tend to have the same read delays when W grows.

In the opposite side, the write times are, by definition, lower in sSPRITE. This happens because sSPRITE never needs to invalidate client leases as SODA does. Figure 7 shows the overhead associated with the lease invalidations.

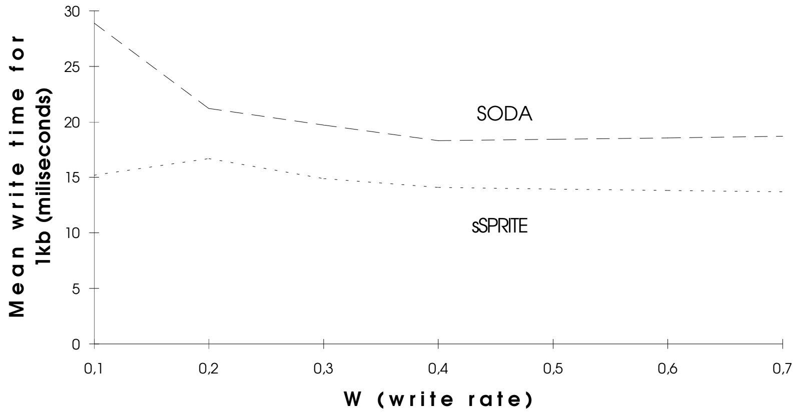


Figure 7: Elapsed time to write 1Kbyte

4.3 Future Work

There are two main topics in this work that can be improved. The analytical model presented in section 3 emulates SPRITE behavior only under concurrent write sharing. In order to model SPRITE protocol in any situation, we would need to consider in our model the `open` and `close` client requests which determine the SPRITE behavior regarding client caching. That would enlarge the model complexity but would present a ultimate comparison between lease and SPRITE protocols.

Our current implementation of SODA does not use any mechanism to ensure the consistency of cached meta-data like directory and file attribute information. We have inherited this problem from LINUX NFS. Extending the lease mechanism already implemented to the meta-data is the main modification needed to make SODA a good consistent distributed file system for LINUX.

After the implementation of the meta-data coherency mechanisms, new extensive tests should be made in order to evaluate SODA's performance more precisely.

5 Conclusion

Among the main distributed file systems, SPRITE – and its descendents [HO93, RO91] – is one of that wich offer the fastest service providing the same consistency of a centralized system. However, SPRITE does not do very well under concurrent write sharing for it completely disables client caching in this situation.

Using a new analytical model, we showed that the lease protocol, first proposed by Gray, produces a lighter server load making use of client caching even under concurrent write sharing.

We implemented the lease protocol in the LINUX operating systems and made some performance evaluations comparing SODA – our lease based system – with a simulated SPRITE. Our tests showed that SODA provides a faster service than a similar system based on SPRITE protocol under a large range of parameter values.

The SODA binaries and source code can be obtained by anonymous FTP at the site ftp.ime.usp.br, directory /pub/linux/soda.

Acknowledgment

The authors gratefully acknowledge the help provided by Dilma Menezes da Silva throughout the development of this research. We are also grateful to Vanderlei da Costa Bueno, Antonio Galves, and Isaac Meilijson for their ideas on the design of the analytical model.

References

[BP81] Richard E. Barlow and Frank Proschan. *Statistical Theory of Reliability and Life Test - Probabilistic Models.* TO BEGIN WITH, Silver Spring, MD, 1981.

[GC89] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 202–210, December 1989.

[HO93] John H. Hartman and John K. Ousterhout. The zebra striped network file system. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[Kon94] Fabio Kon. Sistemas de Arquivos Distribuídos. Master's thesis, Universidade de São Paulo, Instituto de Matemática e Estatística, Departamento de Ciência da Computação, November 1994. Available by anonymous FTP at ftp.ime.usp.br, file pub/articles/kon-master.ps.gz.

[MBH+93] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. Technical Report #103, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, June 1993.

[NWO88] Michael N. Nelson, Brent B. Welch, and John Ousterhout. Caching in the Sprite Network Operating System. *ACM Transactions on Computer Systems*, 6(1):135–54, February 1988.

[RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[Sat90] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, pages 9–21, May 1990.

[SUN90] SUN Microsystems, Inc. *SunOS System & Network Administration.* 1990.