Distributed Lock Manager

Vaibhav Nachankar
M.S. Computer Science
Indiana University
vmnachan@indiana.edu

ABSTRACT

In this paper, we describe Distributed Lock Manager to provide reader with thorough insight into the functionality and evolution of distributed lock manager. Apart from the fact that at present, different solutions for implementation of DLM are available in the market, the insight presented applies to all, in general.
We also describe Chubby lock service which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance.

Keywords

Bigtable, GFS, Paxos.

1. INTRODUCTION

Immense growth in computational requirements of present day technology has aroused extreme interest in implementation of distributed computing architectures which can provide strong computational support. Areas like computational intelligence, scientific research, data ware-housing, entity recognition and like-wise fields depend on concurrent applications with high volumes of data to be processed. High degree of synchronization, resource access management, communication and concurrency support are necessary for ensuring resource availability and enhanced performance. Highly efficient and reliable locking mechanisms are required when competitive resources are distributed across a wide span. Various distributed DBMS's and file systems like Google FS, RedHat GFS, OCFS/2 and QFS use a module to handle resource access and assignment. This module is commonly known as the Distributed Locking Manager (DLM). Locking categories are mainly divided into two classes; Exclusive-mode locking and Shared-mode locking. Based on the deployment of the distributed network, there may be several cases of concurrent file access for read/write operations in file systems or parallel transactions in database systems.

2. Concurrency Control

Amongst the most important and basic properties of distributed transactions, are the ACID (Atomicity, Consistency, isolation, Durability) properties when executed in order to achieve concurrency control. Concurrency control in distributed systems can be classified into two main approaches: Pessimistic and Optimistic algorithms. Pessimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycle making use of serializable methods. Optimistic algorithms delay the synchronization of transactions until their termination. In distributed systems, the approaches can be implemented as central or distributed control points. However, there are pros and cons associated with each of these implementations.

2.1 Centralized 2-Phase Locking

A single node is designated as the command and control section which establishes scheduling of all transaction and lock tables for all resources in the system. One node acts as coordinator for all other sites. In centralized 2-PL, a unique site maintains all locking information for the whole cluster and there is only one scheduler or Lock Manager. Local transaction manager involved in the global transaction requests and releases lock from the centralized local manager using normal 2PL rules. In a global update operation in cluster containing n nodes, generally, a minimum of 2n+3 messages are required. Namely; lock request, lock grant, n update messages, n acknowledgements and unlock request. The disadvantage in this approach is single point of failure and bottle neck in locking requests.

2.2 Distributed 2-Phase Locking

On contrary to Centralized 2-PL, each site has a distributed lock management module. Each lock manager is then responsible for managing locks for data at that site. In case of replicated databases, the implementation is based on a Read-one Write-all (ROWA) protocol. ROWA means that any copy of a replicated item can be used for a read operation, but all copies must exclusively locked before an item can be updated. This approach comparatively improves the issue of formation of a bottle-neck and enhances the cluster reliability and availability. However; communication costs are increased in this case as in general, the messages involved in an update procedure will approach to 5n; namely, 'n' messages for each of lock requests, lock grants, updates, acknowledgements and unlock requests. Moreover, complexity of implementation has to be introduced to accommodate a consolidated operation of integrated resources. This requires the use of voting algorithm in which every node should agree on performing the update process otherwise no update can be performed. Generally, commonly employed algorithms include majority voting algorithms, where every node that is active should agree on update otherwise no update can be performed, nodes that are idle or in sleep mode are not considered and are left out of the process..

2.3 Time-stamping and Multi-copy Approach

These solutions are most favorable while dealing with distributed storage systems, be it databases of distributed file systems. In this, more than once copies are available for each record, at different nodes. The idea is that one of all copies is termed as the master copy while the others are slave copies. Each edit is followed by a new copy with the timestamp. In case of an update need, the read operations are always permitted taking into account the transaction. or request initiation timestamp; however, write operations are blocked if any request is found to be queued up requiring the copy to be written, with a time stamp older than the one present on the latest copy. Rest of the system keeps operating according to the timestamp precedence, as normal. This instantiation of time stamping is called Multi-copy or multi-version timestamp. However, a more basic version of time stamping is also practiced which simple serializes the resource requests according to the initiation timestamps of the requests. The literature and research about the low level locking techniques suggests that this techniques has a lower reliability compared to distributed 2PL but is better suited for substitution of centralized 2-PL solutions.

3. Distributed Locking Manager

Role of the Distributed Locking Manager (DLM) is to maintain the whole process of concurrency control and resource allocation in clusters. The lock manager often deals with only the abstract representations of the resources. The actual resources are usually disjoint from the manager. Each resource is uniquely mapped to a key. All locking services are performed using the keys. A given lock is usually in one of several possible states: (i) UNLOCKED, (ii) SHARED LOCK and (iii) EXCLUSIVE LOCK. There are several existing approaches providing these locking services. A DLM can support several levels of granularity depending on the arrangement of the resource and the request by the client. These modes are roughly divided into levels as files, tables, pages, records, and fields, in case the cluster consists of a distributed file system.

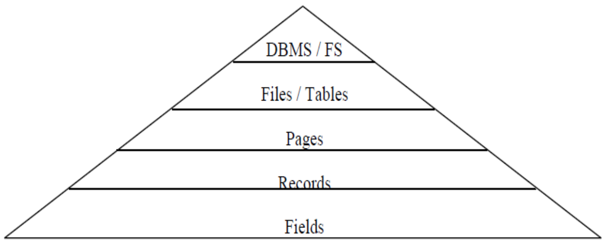


Figure 1. Generic DLM Granularity Levels in File systems/ Databases

.

3.1 Locking Modes in DLM

Distributed Locking Managers or DLMs work as an integrated part of the kernel. In order to synchronize access to devices and to regulate concurrent sharing, DLMs use the following lock modes, as a most basic classification, based on the VMS Cluster implementation. These locks may also be generalized as generic solutions for distributed DBMS, for which, these implementations are highly supported in the available literature.

3.1.1 Null Lock(NL)

Acquiring this lock on a resource indicates that a particular client is interested in that resource in future. By using this lock, the client ensures that the resource and its lock value block are preserved, thus in the low level implementations, it acquires an 'interest' timestamp on that resource. This lock is sometimes called as the Intentional Lock or IL.

3.1.2. Concurrent Read Lock(CRL)

This lock is used to indicate the desire of a read operation on the resource by the client. This lock is granted in a shared mode and multiple clients can be mapped to a single resource by using CRL. This lock is in general employed on high levels of granularity in the locking hierarchy found in a cluster.

3.1.3 Concurrent Write Lock(CWL)

This lock is acquired when the client has the intention of performing a read/write operation on the resource. This lock is allowed access in shared-mode and can be used by multiple clients over a single resource. However, to clear the ambiguity in the situation, the fact that must be emphasized is that this lock can be used only on high levels of hierarchy, not the basic ones. This lock makes way for acquiring exclusive locks on resources residing in lower granularities.

3.1.4 Protected Read Lock(PRL)

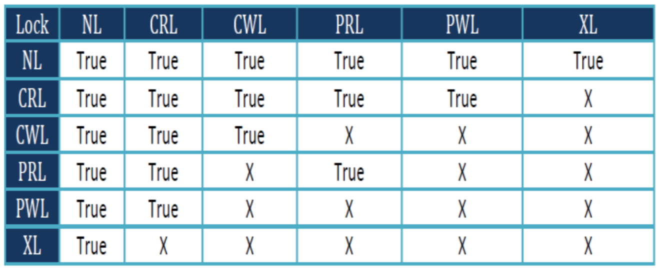
This is the most common shared lock used in traditional databases, termed as Shared lock (SHL) or PRL. This is a middle and low level lock and allows the client the access to execute read operations on resource. However, no write operation can be performed on the resource having this lock, until it has been released. This lock can be acquired by multiple clients, having intentions to make a read operation.

3.1.5 Protected Write Lock(PWL)

PWL is also a middle and low level lock and can be used to availing write access on resources. This in generally applied to the second last level in the class hierarchy of resource granularity as it allows the clients with concurrent read permissions to perform reads on the resource under consideration.

3.1.6 Exclusive Lock(XL)

XL is the lock used at lowest levels in the granularity hierarchy. This lock allows a client to gain exclusive access to resources for reading and writing operations. This lock has some special attributes accounted with it and is to be exercised with great care. The reason is that DLM has to exercise care as if this may not lead to a dead-lock situation in the cluster.



Lock	NL	CRL	CWL	PRL	PWL	XL
NL	True	True	True	True	True	True
CRL	True	True	True	True	True	X
CWL	True	True	True	X	X	X
PRL	True	True	X	True	X	X
PWL	True	True	X	X	X	X
XL	True	X	X	X	X	X

Figure 2. Cross Locking Permissibility

3.2 Implementation models

At present, a completely structured and agreed-upon model of DLM is not present. The solutions available are not exercising all the operations crucial to perfect lock management that is needed in clusters. At present, there are two major models of DLM patches under consideration by the development entities. One has been proposed by Red Hat Inc., and the other one by Oracle. The major difference among these implementations on the API level is the number of arguments to be passed while requesting locks.

3.3 Central Locking Logic

In the existing DLM patch, the Central locking logic is divided into four stages. These stages are described below along with their respective primitives. The details of remote operation or local call are hidden from the user by creating an integrated system abstraction. DLM incorporates the support to manage either of the cases in lower layers of the protocol stack.

3.3.1. Argument Verification

This stage consists of primitives like lock() and unlock() and is mainly concerned with verification of the input arguments and splitting them to be passed further into one of the four main operations:

- dlm_lock = request_lock: this is the basic level of primitive initiated to request for locking operation.
- dlm_lock+CONVERT = convert_lock: This is used to escalate or demote the level of one lock to another lock, considering the locking hierarchy and desired granularity.
- dlm_unlock = unlock_lock: This is the counter-part of the dlm_lock operation and is used to initiate the unlock operation.
- dlm_unlock+CANCEL = cancel_lock: This call is used to abort the locking request by a remote requestor.

3.3.2. Resource Assortment

This step uses the primitive xxxx_lock(). The core function of this step is to search for the resource in the cluster, and prepares to execute a lock on the resource. The resource value block obtained in this step is passed on as argument to the primitive call in next step.

3.3.3. Domain Specification

In this stage the function call from stage two is extended to primitive _xxxx_lock() and it is determined if the domain of the locking operation is local or remote. When the domain is remote, it calls send_xxxx() where as in case of local, it simply calls do_xxxx() primitive

3.3.4. Execution Module

This stage is most crucial of the whole process as it practically executes the scenarios prepared for in the previous steps. The major primitive of interest in this stage is the execution of do_xxxx() function. It manipulates the passed on resource and its value control block. In case of remote procedures, send_xxxx() is extended and evolves into do_xxxx() function to be executed on the remote node. The communication primitives employed in remote procedures for Remote and Local nodes are:

- L: send_xxxx() evolves into R: receive_xxxx()
- R: do_xxxx()
- L: receive_xxxx_reply() down converts into R: send_xxxx_reply()

NOTE: the xxxx string gives a generalized view for all the locks available in the API under consideration.

4. Chubby

This paper describes a lock service called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example,
a Chubby instance (also known as a Chubby cell) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. Most Chubby cells are confined to a single data centre or machine room, though we do run
at least one Chubby cell whose replicas are separated by thousands of kilometres.

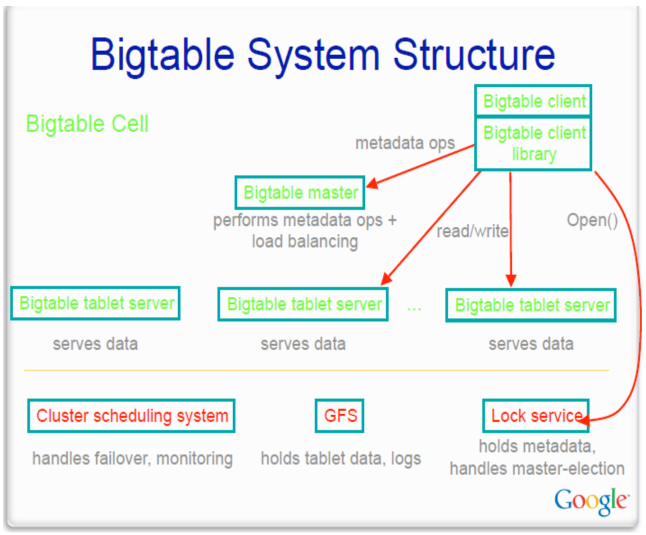


Figure 3. Bigtable System Structure

4.1 System Structure

Chubby has two main components that communicate via RPC: a server, and a library that client applications link against; see Figure 4. All communication between Chubby clients and the servers is mediated by the client library.

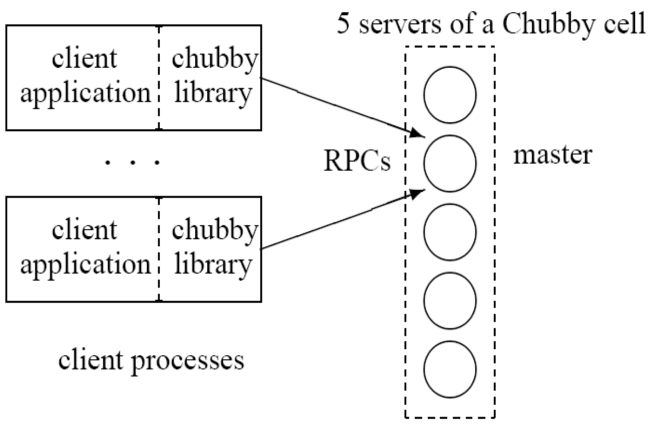


Figure 4 System Structure

A chubby cell consists of a small set of servers (replicas).
A master is elected from the replicas via a consensus protocol
Master lease can last several seconds. If a master fails, a new one will be elected when the master leases expire. Client talks to the

master via chubby library. All replicas are listed in DNS; clients discover the master by talking to any replica.
Replicas maintain copies of a simple database.Clients send read/write requests only to the master.
For a write: The master propagates it to replicas via the consensus protocol and replies after the write reaches a majority of replicas
For a read: The master satisfies the read alone.
If a replica fails and does not recover for a long time (a few hours) then a fresh machine is selected to be a new replica, replacing the failed one. It updates the DNS and obtains a recent copy of the database. The current master polls DNS periodically to discover new replicas.

4.2 File System Interface

Chubby has UNIX like file system interface. Chubby supports a strict tree of files and directories and no symbolic links, no hard links. For example:
/ls/foo/wombat/pouch
1st component (ls): lock service (common to all names)
2nd component (foo): the chubby cell (used in DNS lookup to find the cell master)
The rest: name inside the cell
It can be accessed via Chubby's specialized API / other file system interface (e.g., GFS). Support most normal operations likr create, delete, open, write etc and also support advisory reader/writer lock on a node.
Access Control List (ACL):A node has three ACL names e.g. read/write/change ACL names. An ACL name is a name to a file in the ACL directory and the file lists the authorized users.

File handle: It has check digits encoded in it; cannot be forged.
Sequence number is associated with file handle that allows
a master to tell if this handle is created by a previous master
If previous master created the handle, a newly restarted master can learn the mode information

4.3 Lock and Sequences

Locks are advisory rather than mandatory.Potential lock problems in distributed systems is that A holds a lock L, issues request W, then fails, B acquires L (because A fails), performs actions, W arrives (out-of-order) after B's actions.
Solution #1: backward compatible
Lock server will prevent other clients from getting the lock if a lock become inaccessible or the holder has failed. Lock-delay period can be specified by clients
Solution #2: sequencer
A lock holder can obtain a sequencer from Chubby. It attaches the sequencer to any requests that it sends to other servers (e.g., Bigtable). The other servers can verify the sequencer information

5. REFERENCES

[1] http://www.pittsburgh.intel-research.net/~chensm/Big_Data_reading_group/slides/shimin-chubby.ppt

[2] http://www.cs.uwaterloo.ca/~kmsalem/courses/CS848W10/presentations/Zhan-Chubby.ppt

[3] http://carfield.com.hk/document/distributed/6DeanGoogle.pdf

[4] http://shanky.org/2011/01/26/research-papers-and-videos-on-google-bigtable-gfs-chubby-and-mapreduce/

[5] *The Chubby lock service for loosely-coupled distributed systems:labs.google.com/papers/**chubby**-osdi06.**pdf***

[6] http://en.wikipedia.org/wiki/Distributed_lock_manager

[7] *khawajahashim.files.wordpress.com/.../**white-paper**-on-**distributed-locking-manager**1.pdf*