Lwraft - A Raft based Directory Service

Jin Fang fangj@vmware.com Sung Ruo sruo@vmware.com

April 24, 2017

Abstract

In the cloud infrastructure management space, it is essential to maintain mission critical meta data in a highly consistent and fault tolerant data store. Lwraft, a directory service, evolved from VMware Directory Service, has implemented Raft protocol for data replication among multiple directory instances. Lwraft inherits all Raft properties, including tolerance of N node failure in a 2*N+1 node cluster; it also acquired all features from VMware directory service: LDAP protocol to store and fetch data with rich LDAP data semantics, extensible schema for user defined data structures, dynamically adding indices on user defined attributes, entry level access control and multi-tenancy for customer administration domain isolation. The product also offers ETCD-like key/value data fetch and watch through HTTP/REST interface. This paper describes how Lwraft has implemented Raft protocol with additional handling of LDAP entries replication. It gives a brief description on Lwraft security framework, which utilizes GSSAPI/SRP, for inter-node authentication and data encryption, and the algorithm for member changes without interruption of service. Finally, it provides a assessment on Lwraft scalability.

1 Introduction

In the cloud infrastructure management space, the mission critical meta data should be kept in a data store that shows high data consistency, fault tolerance and high service availability. The cloud meta data includes service configuration, component topology, service discovery, policy changes and service monitoring, among others. VMware Directory service has been used for VMware cloud management infrastructure, such as VSphere's Single Sign On (SSO) services.

Lightwave Raft or Lwraft, a variation of VMware directory service, adopting Raft protocol [1] for directory data replication among server instances in the cluster. As such, it acquired benefits from both Raft protocol and LDAP standard features. Lwraft also implements ETCD-like [3] client API for key/value operations to support this proprietary interface. The design of Lwraft demonstrates its superior scalability as described in this paper.

The organization of this paper is the following: section 2 provides an overview of Lwraft, and compare its capabilities with other well-known distributed data store and directory service products; section 3 presents the unique issues that must be addressed when applying Raft log for LDAP entries; section 4 discusses how Lwraft handles topology changes to ensure the safety property without service interruption; section 5 gives a scalability assessment; it focus on the write throughput since it is usually the bottleneck for committing transactions in a distributed system.

2 Lwraft Overview

Lwraft is a directory service that utilizes Raft protocol for data replication in a cluster of directory service instances. We will refer the Raft paper [1] "the Raft paper" throughout this paper.

Lwraft offers standard LDAP interfaces, as well as ETCD- like key/value operations through HTTP/Rest API. Utilizing Raft protocol for directory server replication has combined benefits: highly consistent data store with standard LDAP data abstraction, API and protocols.

Most, if not all, of Raft implementations are for key/value operations, such as ETCD [3] and Goggle's Logcabin [2]. These implementations support high data consistency and fault tolerance on key/value operations. Lwraft, however, provides additional features absented from those Raft implementations:

- User defined data types for complex applications, which allows the system validating the syntax of the user data.
- Powerful data query predicates using LDAP search filters.
- Dynamic indexing on users defined attribute types.
- Built-in referential constraint enforcement.

- Access Control List (ACL) on LDAP entries by users or groups.
- Multi-tenancy, which allows administration domains isolation within a single Lwraft instance so that each tenant can only accesses his own DIT tree.

With its design choices, Lwraft exhibits the following capabilities:

- Built-in security for inter-node authentication and message encryption without introducing PKI or certificates, thus eliminating the burden of certificates management for adding nodes into the cluster.
- Client authentication using GSSAPI/SRP or security tokens acquired from a third party (e.g. VMware Signle-Sign-On system). [1]
- High scalability for both read and write transactions which is tested for one-terabytes of data.

Lwraft overcomes a drawback of the replication schemes widely used in most of directory service products, for instance, those using data sync protocol (RFC 4533) for master slave replication, or Microsoft AD-like multi-master change-state based replication, which could rollback changes through its conflict resolution algorithm. These products wouldn't guarantee data consistency and persistency, and may incur data loss even if the client has received successful commitment response[2]. Thus, integrating Raft as the LDAP replication protocol seems a nature choice to obtain combined benefits of both Raft and LDAP.

3 Algorithm

Lwraft implemented all algorithm aspects presented by [1] with some additions to address the unique issues incurred from applying LDAP entry changes. This section gives a high level highlight of the standard Raft algorithm first, and then discusses the additions to the Raft protocol with Lwraft.

Raft protocol defines two primary operations: leader election and log replication. Raft Ping is a special case of log replication.

Lwraft implements leader election algorithm almost exactly as the one presented in paper except going through a procedure, "WaitLogCommitDone", when a node is elected as a new leader as described later in this section. The Raft replication algorithm can be characterized as a sequence of inductive operations: in a cluster with 2*N+1 nodes, to "commit" a log with index L, the leader must replicate (persist) the log to N nodes, plus persisting the log at the leader itself; to persist log with index L, the leader must has committed log right prior to L (referred to as prevLogIndex in the RPC arguments). The Raft paper also defines "committed a log" as when the log has replicated (persisted) to at least N+1 nodes [3]

Raft defines so called "apply a log" as "changing the server's state based on the log content". In most Raft implementations (such as ETCD and Logcabin), "apply a log" is simply adding, modify or delete a pair of key/value. The Raft paper, however, hasn't clarified what procedures are needed for "apply a log". In particular, it doesn't require applying logs in the same order as they are committed; this would be no problem with key/value pairs operations, as long as there is enough data storage capability because there is no dependency between different key/value pairs. Thus, the Raft paper suggests first persisting log at the leader before acquiring consensus from peers, and then persisting and applying the log at the leader.

In Lwraft, a Raft log represents changes to a LDAP entry, and applying the change is more complicated since there is dependency between LDAP entries, including referential constraints and entry level access control. As such, before replicating a Raft log, the LDAP operation must have been validated (e.g. passed all integrity checks) before it is deemed to be able to be "applied" successfully. Therefore, the inductive operations should be extended to log application: in order to apply log with index L, its prevLog must have been successfully applied. This stricter inductive operation requirement makes implementing Raft protocol to LDAP directory more challenging than handling key/value pairs.

To maintain LDAP operations dependency and improve performance, we slightly modified the original Raft algorithm: instead of persisting the Raft log at the leader before seeking consensus from other nodes, Lwraft combines Raft log creation and Raft log application into one transaction, and commit that transaction after the consensus has reached. However, protocol safety must be ensured with the following safeguards:

[1] Lwraft has the capability as an authentication data repository and supports multiple authentication protocols, including SRP and kerberos.

[2] There are other transaction commit protocols, such as two-phase or three-phase commitment protocols in the distributed database arena, but they haven't been adopted for data replication due to the possibility of service blocking or two much message delay

[3] Raft leader may or may not know what the last "commit log" is. For instance, when the leader obtained consensus and persisted the log locally, it would know the last commit log; if, however, the leader crashed, and the new elected leader only know it possesses the latest commit log, but may not know if the last persist log is committed. We will discuss this issue at the end of this section.

1. The leader must either be able to commit or abort the combined operations (raft log creation and its application) in that single transaction before seeking consensus from the majority.
2. CommitIndex must not be updated for the pending transaction until the leader has actually committed that transaction locally after consensus has reached, otherwise a follower might apply that log index via Raft Ping, resulting in applying a log entry that might not be able to commit.
3. If the leader cannot complete the commit operation due to storage failure (e.g. disk full), then the leader may not reuse the log index/term for future client request since the transaction is implicitly aborted locally.
4. The transaction with the combined operations will not be explicitly aborted once AppendEntries RPC calls have been issued unless the leader has changed role to non-leader. This prevents the log index/term be reused for future client request.

As such, Lwraft makes a modification to MDB transaction engine on the transaction commit procedure (txn_commit with write flag) with three callbacks, which are conditionally invoked before returning control to serialized callers:

1. RaftPrepareCommit - it is invoked when the leader has completed its local transaction (combined LDAP operation and its Raft log), and is ready to persist the dirty pages onto the local storage.
2. RaftPostCommit - it is invoked when the leader has obtained consensus from majority of servers (include the leader), and successfully persisted the transaction. This callback will update the local commitIndex contained in the committed log entry.
3. RaftCommitFail it is invoked when the server cannot complete the transaction commit locally due to storage error, and implicitly aborted. This callback sets the server to Follower.
4. No explicit transaction abort in RaftPrepareCommit once AppendEntries RPC calls have been issued unless the server has detected that it is no longer a leader. [4]

Same as the Raft paper, persisting a log doesn't commit that log right away at a follower; instead, the next successful log replication would commit the previous log at the follower; thus a follower must "apply" logs in the same order of the logs being committed to ensure that there is the same sequence of state transitions (applying LDAP entries).

Additional handling is needed through procedure "WaitLogCommitDone" when a node is elected as a new leader; it must ensure that the logs in the local server being replicated to all remote servers, and applied in the same order as they are replicated before accepting new client update requests. A property of Raft is that the new leader always contains committed logs due to its election algorithm; however, it may contain logs that are not committed, which would have higher index than the highest committed log. In the Raft paper, it simply indicates that such logs should be indirectly committed (described in its section 5.4.2 of the Raft paper). In Lwraft, procedure "WaitLogCommitDone" is executed at a new leader, which would monitor the progress of Raft Ping to replicate those uncommitted logs, and then applying those logs in that order before accepting new client update requests.

4 Topology Changes

Lwraft adopts a simplified algorithm than the one presented in the Raft paper section 6. Lwraft doesn't interrupt the service as the algorithm proposed in the Raft paper when adding or removing nodes in topology. In the Raft paper, it avoids server down time by using a two-stage approach when adding and removing members. Lwraft implements member change in one stage with additional restriction: a member can be added or removed at a time using provided Lwraft utilities. We will informally prove that the topology change process is safe with the way a new node is added and removed.

Lwraft implemented inter-node authentication and RPC message encryption with GSSAPI, combined with a plug-in, Secure Remote Password (SRP) protocol [7]. The identity of a node is created within a diretory entry when it is added to the cluster, and destroyed when a node leave the cluster. This facility ensures nodes are authenticated with each other when RPC connection is established without transmitting clear-text password over the wire. This facility doesn't depend on any PKI certificates, and eliminated the burden of setting up and maintaining certificates among servers for inter-node authentication.

In order to accelerate adding a new node into cluster, Lwraft transfers (makes hot copy of) the whole database files, through its RPC interface, to the new node location, avoiding doing a long peroid log catch-up. This is very useful for adding a node with a large database already.

[4] We cannot simply increment CommitIndex in such case to avoid reusing that log Index because that implies committing CommitIndex' previous log (the one just aborted), and some servers might have persisted the CommitIndex' previous log.

	Write Thoughput	Key Length	Value Lenght	Memory Footprint	Storage Size
ETCD	138/sec	62 bytes	1520 bytes	5.4 GB	1.5 GB
Lwraft	139/sec	62 bytes	1520 bytes	327 MB	1.8 GB

Table 1: Scalability Comparision

To informally approve protocol safety, we first consider the case where there are at least two nodes already in cluster before adding a new node, and then discuss the case where a node is added to cluster with only one node.

Adding a node is initiated with a few LDAP add and modify operations (via a utility) toward the directory. Since we start file copy to duplicate the database after the LDAP operations have been successfully committed, the operation must have been propagated to the majority of nodes in the old view of topology; plus the new node itself contains the new view. In other word only the minority of nodes in the new topology view may contain the old view. As described in the voting algorithm, those minority of nodes cannot get grants from any node in the majority of nodes since they have lower log index; therefore only the nodes with the new view can obtain enough votes to become leader; Now it is obvious that no more than one nodes in the set of the majority of nodes (those have the new view) can be elected as leader which is ensured by the existing property of the voting algorithm.

As with the special case where a node is added to a cluster with only one node, either both nodes contain the new view when voting process is initiated (after node adding procedure completed successfully), or the process is aborted at the newly added node. In both cases, no more than one leader can be elected. In scenario where the new node is failed to complete file copy or the whole process, the entry representing the new node might have been committed to the server at the first node; in such case, the provided utility must be used to rollback the changes from the first node (as if removing that node) to reflect the correct topology, otherwise it cannot commit any new transactions.

It is required that the node must be shutdown (enforced by the utility) before removing that node so that the node cannot involved in the election process in the future, thus no two or more leaders can be elected as the current property of the voting algorithm [5].

5 Scalability

With scalability as one of the major objectives, Lwraft relies on an enhanced transaction engine, i.e. MDB with Write-Ahead-Logging [6], to improve write throughput. The scalability was tested with up to a terabyte of data.

As described in [5], MDB shows superior read performance and scalability on machine with multiple CPUs since it doesn't use locks for transaction serialization. This section gives an assessment on how Lwraft scales with write throughput because the write performance is the major concern to support a large scale deployment.

As all distributed systems, write performance with Raft is influenced by both database write latency and Raft protocol message exchange latency. It should be indicated that Raft based service is to achieve goals of data consistency, fault-tolerance and high system availability, instead of accelerating local data access in a distributed system with servers deployed in a wide geographical sites. The inductive algorithm is actually making replication of data strictly serialized. Therefore, any latency in the network would directly impact on the write performance. In fact, Raft based service should be deployed at hosts with very low network latency.

In order to assess Lwraft write performance, we deploy server instances on VMs hosted by a single Hypervisor, ESXi6.0 instance, which would eliminate the network latency as a major factor of measuring the efficiency of Raft protocol implementation, as well as backend performance. We compared Lwraft write throughput with CoreOS/ETCD key/value set operations in a three-node setup. A LDAP schema objectclass is created for Lwraft to mimic key/value set option via LDAP Add.

It should be indicated that Lwraft schema for key/value uses more attributes than ETCDs key/value because the Lwraft LDAP entry contains extra attributes for indexed search and other purposes, e.g. subtree scope search requires a reverse name index, and access control requires a security-identifier, among other operational attributes. We used identical key/value size of 62/1520 bytes. We provisioned 250,000 key/value pairs and then calculate the average operations per second. The 250,000 pairs were added with two concurrent clients[6]. The test was conducted on three VMs, each configured with 8GB main memory, two 2.4GHz CPUs. The host machine is an 8 CPU (16 cores) running ESXi6.0 configured with an Intel750 PCIe Solid Stage Drive with a built-in PCIe SSD drive. The table below gives the test result [7] We would like to test the rate with much larger scale, but ETCD would take larger memory footprint as more data provisioned, which

[5] Though, this requires one more node, in the new topology view, to commit node removal. E.g. to remove a node from a three-node cluster, the other two nodes must be up and connected.

[6] With more than two clients, ETCD would trigger group commit, resulting higher throughput. Lwraft currently has not imlemented group commit

[7] The storage space is the data under member/snap for ETCD, data.mdb for Lwraft; Memory Footprint is the residency memory allocated with malloc or alike.

eventually being killed by the OS.

Here is a summary of this result: when adding 250,000 key/value pairs with the same key/value size of 62/1520 bytes, Lwraft is slightly better than ETCD on the average write throughput: 139 vs. 138 per second. The residency memory would increase as more data is added with ETCD, but it is constant for Lwraft. We wasn't able to test ETCD with larger scale with more than 250,000 key/value pairs which consumed 5.4 GB main memory; Lwraft was tested with one terabytes of data storage and 100 million LDAP entries[6].

6 Conclusion

Lwraft implemented Raft as the replication protocol on a cluster of LDAP servers, and offers both benefits from Raft protocol and VMware directory service, including high data consistency, fault-tolerance, high service availability, standard LDAP API, extensible schema, dynamic indexing. Lwraft also provides ECTD-like key/value operations and watches. Lwraft implementation introduced additional algorithm aspects so that Raft log application can meet LDAP dependence constraints between entries. Lwraft has a built-in inter-node authentication and RPC message encryption that eliminated the complexity of managing PKI certificates when adding or removing nodes. Lwraft scale was tested with one terabytes of data for large-scale deployment.

References

[1] Diego Ongaro and John Ousterhout, "In Search of an Understandable Consensus Algorithm (Extended Version)" May 20, 2014

[2] Diego Ongaro (Stanford University), "git://github.com/logcabin/logcabin.git", 2015

[3] Brandon Philips, "Distributed computing powered with etcd: Overview and future", CoreOS, May 2016

[4] Jin Fang AND Sung Ruo, "VMware Directory Server Scalability Enhancement", Pending VMware Radio Acceptance, 2017

[5] Howard Chu, "MDB: A Memory-Mapped Database and Backend for OpenLDAP", Symas Corp., OpenLDAP Project, 2011

[6] Jin Fang, "https://wiki.eng.vmware.com/Vmdir_Performance_Evaluation", VMware Internal, 2015

[7] Wu, Tom, "SRP-6: Improvements and Refinements to the Secure Remote Password Protocol", October 29, 2002"