Peer-to-peer Technologies Applied to Data Warehouses

Simone Cirani, Lorenzo Melegari, and Luca Veltri
Department of Information Engineering - University of Parma (Italy)
Email: simone.cirani@tlc.unipr.it, lorenzo.melegari@gmail.com, luca.veltri@unipr.it

***Abstract*—Data mining and user data collection applications, like Facebook and Yahoo, make dealing with huge amounts of data more and more frequent. A solution to cope with this problem is to spread data over multiple network-connected physical devices. Having more devices, though, means increasing system complexity and introducing additional possible points of failure. Moreover, despite the capacity of hard drives as massive storage systems has increased extremely during years, the speed at which data can be accessed has not. In order to address this problem, over the years, distributed file systems, such as NFS and HDFS, have been designed and deployed. Such systems provide access to files stored on multiple hosts connected through a computer network in a transparent way to users. The peer-to-peer network paradigm has been introduced to overcome some limitations of the client-server architecture by adding features, such as scalability, fault-tolerance, and self-organization. In this work, we present a solution that integrates peer-to-peer network support to HDFS in order to realize a flexible, low-cost and, dynamic distributed file system.**

I. Introduction

When the amount of data to handle exceeds the physical capacity of a single machine, alternative solutions must be considered, such as splitting data on multiple machines (clusters). The most natural mechanism to share data among hosts is to use a computer network, e.g. a LAN or a larger IP network. In order to address this need, distributed file systems have been introduced. Distributed file systems are actual file systems that encapsulate all the functionalities related to data distribution over the network. The complexity of such systems is intrinsically higher compared to common file systems, where data are stored on a single host. Particularly, great attention must be paid on system reliability since multiplying the number of hosts introduces many points of failure. Therefore, issues like network availability, host failures, and the speed of data access, must be taken care of.

World-famous Facebook and Yahoo have to deal with such problems everyday. Facebook has 2 major clusters comprising a 1100-machine cluster and a 300-machine cluster, each one with 8 cores and 12 TBytes of storage. Yahoo has more than 25000 machines dedicated to distributed file system. Such examples are overkill for a distributed file system application, but many others scenarios face the same issues. It is therefore a concrete problem that needs to be approached, also by not so high budget projects.

Some well-known current implementations of distributed file systems are Network File System (NFS) [1], Hadoop Distributed File System (HDFS) [2], Amazon Simple Storage Service (Amazon S3) [3], and Google File System (GFS) [4]. One of the main goals of a NFS is data availability, that can be pursued by means of robust systems like RAID, which are usually available on dedicated (high-end) servers. Since the cost of a single high-end server tops that of many low-cost computers, current implementations of distributed file systems often make use of and scale with commercial-off-the-shell (COTS) computers. Since failure rate of COTS components is higher than that of high-end ones, these systems must explicitly implement efficient and reliable mechanisms for integrity check, redundancy, and transactionality. However, due to the intrinsic static configuration of these systems, they require proper setup and ad-hoc configuration in order to work.

On the other hand, a peer-to-peer (P2P) computer network relies primarily on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. P2P networks are typically used for connecting nodes via largely ad hoc connections. An important goal in P2P networks is that all clients (peers) provide resources, including bandwidth, storage space, and computing power for network and service maintenance. Thus, as new nodes arrive, the resource demand increases together with the total capacity of the system. This is not true in client-server architectures with a fixed set of servers, in which adding more clients could mean slower data transfer for all users. Nodes that belong to P2P networks have heterogeneous resources, and their behavior is typically unpredictable: nodes may join and leave (churn) the network at any time, either gracefully or silently, and at an unknown rate. P2P networks are designed to handle these conditions of extreme dynamicity.

P2P networks are built by establishing logical links among a number of nodes that belong to some existing network, such as the Internet, thus forming a so-called *overlay network* (or simply *overlay*). The rules that define how these logical links are built and how messages are routed in the overlay network determine the kind of P2P network. In the case no strict rule is defined for link establishment, we have unstructured P2P networks, which typically rely on flooding as routing strategy. Structured P2P networks, on the other hand, define strict rules for link establishment and message routing. Typical examples of structured P2P networks are Distributed Hash Tables (DHTs), which arrange nodes in the overlay according to a specific topology (i.e. a ring in Chord, leaves of a binary tree in Kademlia), in order to achieve some features such as logarithmic lookup procedures, with little (logarithmic) pernode state information. DHTs provide an information storage and retrieval service among a number of collaborating nodes, based on a key/value mapping, similar to an hash table. DHTs provide typical P2P network features, such as scalability, fault-tolerance, and self-organization.

In this paper, we propose a new solution for HDFS based on a P2P approach. Our goal is to exploit the benefits of P2P networks in terms of dynamicity to typically static HDFS configurations. Such design will make it possible to add or remove nodes to HDFS as needed, in order to use exactly as many nodes as needed, with no worries to under- or over-estimate the number of nodes required to perform jobs.

This paper is organized as follows: in section II we will describe the HDFS architecture, in section III we will explain how to integrate P2P support in HDFS, with particular reference to our Distributed Location Service framework as basic P2P enabling technology. In section IV we will discuss our implementation and in section V we will envisage future works to further improve our design. Finally, in section VI we will briefly discuss about other proposals for applying peer-to-peer technologies to distributed work, and in section VII we will draw our conclusions.

II. Scenario and HDFS Architecture

As common file systems, HDFS uses the concept of blocks of data. The block size is the minimum amount of data that the system can read and/or write. Stored data are broken into chunks according to the block size, which are then stored as independent units. Splitting data into blocks leads to several benefits: it simplifies the storage management as every size computation is related to multiples of blocks. Blocks are also used as base units for data replication.

In a HDFS cluster, blocks are persistently stored into one or more *workers*, each one called *datanode*. Datanodes are pure "dummy" workers, as their job is to read and write data, and periodically send reports of which data they contain.

The file system manager is another node, called *namenode*. The namenode manages the filesystem namespace, maintains its hierarchical tree and the metadata for files and directories in the tree. This information is stored persistently on the namenode's local disk. The namenode also keeps a registry of all the blocks of a given file and on which datanodes these blocks are located. The structure of the entire system is highly dependent on the namenode. Without the namenode, the entire file system would be useless since there is no way to reconstruct a file from the blocks stored on the datanodes. For this reason, it is important to make the namenode resilient to failure. This can be achieved by making regular back-ups of the persistent state of the file system metadata and writing them to multiple storage devices. This back-up jobs can be done continuously by a special node called *secondary namenode*, which has to be dedicated to these tasks only as continuous registry merging could be computing intensive.

HDFS also provides replication methods to avoid data loss. Each time a new block is written, the namenode should also locate where replicas of the block should be placed. In common scenarios, every block is replicated three times. A CRC function applied on every 512 bytes of each block is also normally used to achieve data integrity.

A HDFS cluster is typically made by several (even hundreds) datanodes (*data center*), organized in different *racks*, which are redundant in nature, and one namenode, that is indeed extremely susceptible to hardware failure.

The main operations that an application can do on a HDFS file system data are writing, reading, appending, and deleting. HDFS takes care of all the issues related to concurrent access by external clients. During the reading phase, the client first connects to a private instance of a HDFS proxy, which provides access to the namenode interface. The request is then forwarded to the namenode through a RPC call. The namenode then scans its file/block register to investigate how the requested file was split and where each block was stored. For each block, the closest datanode references, according to proximity metric of the topology of the cluster's network, are retrieved and reported to the proxy. Then the proxy creates input streams from the client to the datanodes. Input streams make it possible to read data that are stored on different datanodes in a transparent way, thus hiding the actual location of blocks, as if reading data from a single, continuous flow. HDFS manages IO failures by redirecting to available datanodes in case of unreadable blocks. HDFS files cannot be written at arbitrary positions. Writing operations on existing files are only allowed for appending or deleting. Thus, writing operations themselves are done for adding new files only. During the writing phase, the client contacts a private instance of a HDFS proxy. The proxy constructs a *create request* that is forwarded to the namenode, which then checks if the file already exists. If it does, the request is denied. If the file doesn't exist, the namenode reports the parameters required to construct an output stream to the proxy. The output stream contains two different queues: the *data queue* and the *ack queue*. The data queue contains blocks of data ready to be forwarded to the datanodes. For each block, the stream contacts the namenode to get a list of datanodes the block should be stored on. After such information is received, the block is forwarded to the first datanode of the list (*datanode pipeline*). As a block is forwarded to a datanode, it is also temporarily stored in the ack queue. When a datanode pipeline is done writing a block, it commits to the ack queue, which will then remove it. If even a single datanode in the pipeline fails to write, the ack queue removes the unsuccessful datanode from the pipeline and puts the data back in queue to successively write on the datanodes that were skipped after failure.

In order to ensure that HDFS works correctly, as sketched above, all the machines in the cluster should be pre-configured accordingly. Every datanode must know who the namenode is and where it is located in order to run successfully. The namenode must also know every datanode location so to reach it and make it active during a working session. The HDFS working environment is extremely static. Therefore, if more machines are needed to perform a job (for example because their number has been underestimated), new machines need to be added to the cluster; on the other hand, if less machines are needed (because they have been overestimated), some machines would not be used and would not be operant; in both cases the entire system should be opportunely rebalanced and configured.

III. Peer-to-peer and HDFS Integration

The HDFS distributed filesystem is focused on scaling well on commodity machines, providing good data transfer performances with high reliability. HDFS provides methods for balancing the cluster load and adding/removing datanodes to and from the system. Each machine should be pre-configured to act as a datanode, which involves knowing exactly where the namenode is and how to reach it. Also, during system startup, the namenode should know an initial list of datanodes to work properly. These constrains can limit system's functionality in those scenarios which do not guarantee complete staticity, as every datanode should be configured for accessing a well known namenode. This means that if the namenode changes because of a failure or if there is more than one namenode, each one for a specific, time-limited, working session, every datanode must be reconfigured. In such scenario, we propose to use a P2P layer underlying the HDFS system to provide a hot-plug framework for datanodes to dynamically connect and disconnect from an HDFS working session without compromising overall system's performance.

Our design relies strongly on a component, called Distributed Location Service (DLS) [5], which provides a P2P service of storage and retrieval of location information. The DLS is a DHT-based framework which we have conceived in order to enable applications to establish direct connections among the endpoints of the communication in a P2P fashion. The DLS stores mappings of key/value pairs, which associate the URI of some resource to the information needed to access it, such as its URL, its expiration time (i.e. the time after which the resource should not be considered), and an access priority value. The mapping is 1:N, rather than 1:1, due to redundancy obtained by registering more resource values for the same resource name. The DLS framework makes no assumption on the DHT algorithm and communication protocol that is going to be used, thus allowing for different implementation options as needed. The interface for accessing the DLS is transparent to applications, as it relies on two basic RPCs: *put(key,value)*, used to store a key/value mapping in the DLS, and *get(key)*, used to retrieve the set of values associated with a key in the DLS.

Our goals in the design of the architecture are:

- full exploitation of existing resources: the system should use all the available resources in terms of storage capacity, computational power, and bandwidth; the system should work with heterogeneous machines, thus dropping the constraint of using only high-end hardware;
- *load distribution*: the system should autonomously balance the associated resources;
- *upgradeability*: the system should support overall capacity changes at runtime;
- *data persistence*: data loss is not acceptable, thus the system should guarantee that stored data are always available;
- *data availability*: the system should ensure data integrity at all times;
- *data access*: stored data should be seekable and accessed in reasonable times.

Another not technical feature that we aim to provide is the possibility to cut costs for the infrastructure, both for buying machines and maintaining them. Assuming that the system works well with non high-end machines therefore has strong economic implications, thus making these solution affordable and particularly interesting even for low-budget projects. Nowadays, using COTS hardware implies that hardware replacement is often preferable and more practical than professional assistance, both for hardware problems solutions and software configurations.

In the proposed architecture, the DLS is used as an utility layer to connect cluster machines one to each other and exchange information among them. To do so, a static configuration is no longer needed. The namenode connects to the overlay, acting as a DLS peer and publishing information. When the namenode joins the overlay it publishes its status and service addresses in the DLS through successive put RPCs, thus making them accessible to other DLS peers. These information are then maintained collectively by all the peers participating in the DLS, but only the namenode can and must keep them fresh. All the information a node needs to become an active HDFS datanode are therefore available in the DLS and can be retrieved through get RPCs. As a new datanode enters the system, it can read the needed namenode information from the DLS layer, discover the namenode's location and contact it for authentication. As the namonode accepts the new datanode, the datanode becomes part of the HDFS cluster and all the following operations are done using the HDFS RPC protocol, so that the DLS is not involved anymore and HDFS performances are preserved. Figure 1 shows how namenodes and datanodes interact with the P2P substrate.

The underlying DLS layer has been designed to also introduce benefits during namenode temporary failures. The resource access information stored in the DLS include, besides the URL needed to access the resource, also an expiration time, after which the resource should not be considered fresh, and it is actually removed from the DLS. This typically happens when a resource is not renewed after a failure. In this case, the information that needs to be renewed is the namenode access information, which, if absent, denotes the failure of the namenode. During namenode downtime, its access information are therefore no more available on the DLS. In the proposed architecture, while the namenode is unavailable, datanodes begin to poll on the DLS, waiting for the namenode or its replacement to become active. As soon as a namenode joins the DLS, it publishes its access information, so that datanodes

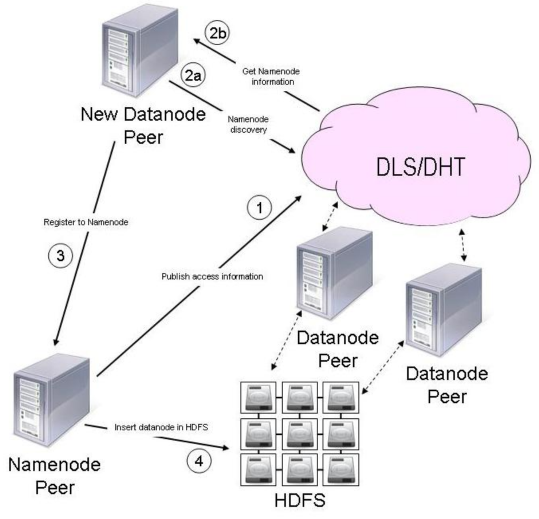

Fig. 1. DLS and HDFS integration

can contact it to register and the system returns to a fully functional status. Therefore, the system itself is responsible for its functional consistency, thus eliminating the drawbacks of assistance in the case of failures.

In the proposed work, we made the assumption that HDFS node and DLS peer both reside (and cooperate) on the same machine. However these roles are only logical and the same system behavior can be achieved when the HDFS node and peer do not coexist on the same machine, by using so-called *adapter node*s to interact with the DLS overlay. Adapter nodes are DLS edge nodes capable of interfacing with applications that do not belong to the DLS. Such nodes parse and forward location information requests into the DLS, thus acting as a proxy/gateway to the DLS. In this scenario, adapter nodes are used by HDFS nodes to get location information without participating in the DLS life cycle.

IV. Implementation

Development of the proposed work has been done, using the HDFS implementation from the *Apache Hadoop Project*[1].

In our implementation of a DLS, we have used Kademlia [6] as a DHT algorithm. This choice was taken as Kademlia outperforms other DHT algorithms in terms of robustness (i.e., churn handling, key redundancy). We modified the original Kademlia algorithm to better react to failures, by implementing a failure notification procedure that propagates in the DHT, thus allowing to remove failed nodes from a node's routing table in order to increase efficiency in routing paths. dSIP [7] has been used as a DHT management and maintenance communication protocol. dSIP has first been adapted [8] to fit the Kademlia algorithm specification, and then properly modified to handle our failure notification procedure.

On the DLS framework, an Hadoop Generic Peer was developed as a generic connection layer (both synchronous and asynchronous) between a DLS Peer and the HDFS/Hadoop architecture. On top of this generic connector, the choice has been to encapsulate the DLS functionality needed by the HDFS node in two different wrapper structures, a *DataNode Service Wrapper* and a *NameNode Service Wrapper*. These wrapper structures offer bridge functionality the generic implementation of the Hadoop Generic Peer. Usage of the Generic Peer structures makes the system independent from the underlying DHT algorithm used. System startup proceeds, as in the traditional HDFS implementation, first namenode then datanodes. However, in the proposed work, namenode initially have no information about which datanodes belong to the system. During namenode boot, the associated Peer publishes on the DLS the namenode IP address and service port, using a well known resource key. On datanode boot, no information about the namenode were previously provided. The datanode Peer does a namenode discovery on the DLS. If a namenode is available, then the datanode contacts it through the HDFS RPC protocol. When a datanode wants to leave the system, it first sends a HDFS leave request to the namenode, than it executes a leave procedure on the DLS. In case of a temporary namenode failure, the datanodes remain connected one to each other through the DLS layer. The namenode could leave the system in two ways:

- *Graceful leave :* the namenode removes service information from the DLS layer, waits for all datanode disconnection and then leaves. Datanodes will remain polling the DLS at a given rate, waiting for a new namenode to arrive.
- *Failure :* the namenode becomes suddenly unavailable, all the service information on the DLS are removed after the renew timeout has expired. While it has not, the system status is unpredictable, as all the calls to the namenode from datanodes will fail. After namenode information are removed, datanodes polls the DLS waiting for a namenode.

V. Future Work

Future research will focus on using other DHT algorithms for the DLS than Kademlia. Indeed, the relatively small number of nodes in a typical scenario makes Kademlia an overkill in terms of DHT size and churn rate. Other DHT algorithms, such as one-hop DHTs [9], appear to fit better into the architecture for our reference scenario. One-hop DHTs provide O(1) lookup procedures. This performance boost can be achieved only with O(N) state information, which make these DHTs suitable only for relatively low numbers of participating nodes. However, this is actually the kind of applications where such DHTs might offer their full benefits.

Another direction of our research will be the full distribution of the functionalities of the namenode. This feature would allow the deployment of a HDFS architecture in a purely P2P fashion. All nodes would collaborate to implement the management of the file system collectively, thus making the entire architecture more robust, since the namenode functionalities would be divided among all the peers. The advantages

[1]*http://hadoop.apache.org/hdfs*, *http://www.apache.org*

would therefore be the elimination of single point of failure and increment the performance for the system since the intermediation of the namenode bottleneck would be no longer required. This task is very challenging since it introduces several problems like:

- handling concurrent read/write data accesses;
- consistency of file system indexes;
- synchronization of data duplicates;
- consistency check.

Replacing the namenode with a distributed system comes at the price of overloading all datanodes of all the management of all the information related file system and the synchronization with the overlay participants. This approach increases the overall complexity and the must be properly designed to ensure that full functionalities are available at any time, even when nodes fail.

VI. Related Work

In literature, few works have addressed the issue of applying peer-to-peer networks to manage large amounts of distributed data. BlobSeer authors in [10] and [11] have proposed to handle storage and access of binary long objects (*blob*s), such as reading, writing, and appending, by splitting blobs into chunks and using metadata information for concurrent access management. Metadata are stored in a tree built on top of a Distributed Hash Table. In [12], the authors show how to build a distributed file system by directly storing files in a DHT. Such approach, in our vision, is not promising, as it does not address the issue of concurrency and data integrity check, which should be provided instead for data-intensive applications. Moreover, DHT reorganization, as a consequence of churn, implies that huge amounts of data be transferred from node to node. Because of this, we envisaged that a reference-based distributed system should behave better in terms of performance and P2P overlay maintenance.

VII. Conclusions

In this work we have described a possible approach to integrate peer-to-peer support into a HDFS architecture by using a Distributed Location Service as a means to connect cluster machines and allow communication among them in dynamic scenarios, thus removing the need for ad-hoc configuration of the HDFS system. The peer-to-peer layer underlying the HDFS system is meant to provide a hot-plug framework for datanodes to dynamically connect and disconnect from an HDFS working session without compromising overall system's performance.

After a description of the HDFS architecture, based on the concept of blocks of data, and the roles of datanodes and namenodes, we have outlined how peer-to-peer support could be integrated. Our solution is based on a Distributed Location Service, which is a peer-to-peer framework that allows the implementation of a service for storing and retrieving information about the location of resources. The DLS is based on DHT, in order to ensure the features of structured peer-to-peer networks for overall system performance.

Our proposed architecture has been realized using the Apache Hadoop HDFS implementation and a DLS based on a Kademlia DHT algorithm, modified on purpose in order to efficiently handle node failure events in the network.

Finally we have sketched future directions for our research, such as the use of one-hop DHTs that appear to fit better for relatively small peer-to-peer networks, and the possibility to fully distribute the functionalities of HDFS namenodes on the peer-to-peer layer, in order to make the HDFS architecture more robust.

References

[1] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) Version 4 Protocol. *RFC 3530, IETF.*, Apr. 2003.

[2] Apache Hadoop Project. *http://hadoop.apache.org*.

[3] Amazon Simple Storage Service (S3). *http://aws.amazon.com/s3*.

[4] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, Lake George - NY, USA, Oct. 2003.

[5] S. Cirani and L. Veltri. Implementation of a framework for a DHT-based Distributed Location Service. In *Proceedings of the 16th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2008)*, Split - Dubrovnik, Croatia, Sep.. 2008.

[6] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR metric. In *1st International Workshop on Peer-to-peer Systems*, 2002.

[7] D. Bryan. dSIP: A P2P Approach to SIP Registration and Resource Location. Internet-Draft *draft-bryan-p2psip-dsip-00, IETF*, Feb. 2007.

[8] S. Cirani and L. Veltri. A Kademlia-based DHT for Resource Lookup in P2PSIP. Internet-Draft *ciranip2psip-dsip-dhtkademlia-00, IETF*, Oct. 2007.

[9] L. R. Monnerat, and C. L. Amorim. D1HT: A Distributed One Hop Hash Table. In *Proceedings of the 20th IEEE Parallel and Distributed Processing Symposium (IPDPS '06)*, Rhodes Island, Greece, Apr. 2006.

[10] L. Nicolae, G. Antoniu, and L. Bougé. BlobSeer: How To Enable Efficient Versioning for Large Object Storage Under Heavy Access Concurrency. In *Proceedings of the 2009 EDBT/ICDT Workshops, pages 18-25*, Saint-Petersburg, Russia, Mar. 2009.

[11] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier. BlobSeer: Bringing High Throughput Under Heavy Concurrency to Hadoop Map-Reduce Applications. In *Proceedings of the 24th IEEE Parallel and Distributed Processing Symposium (IPDPS '10)*, Atlanta, USA, Apr. 2010.

[12] J. Pang, P. B. Gibbons, M. Kaminsky, S. Seshan, and H. Yu. Defragmeting DHT-based Distributed File System. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS'2007)*, Toronto - Ontario, Canada, Jun. 2007.