- BEHIND THE CLOUD - [1]
A Closer Look at the Infrastructure of Cloud and Grid Computing

Alexander Huemer
Florian Landolt
Johannes Taferl

Winter Term 2008/2009

[1]created in the "Seminar aus Informatik" course held by Univ.-Prof.Dr. Wolfgang Pree

Contents

List of Figures

Abstract

This paper discusses some aspects of cloud computing infrastructure. The first chapter outlines the role of Distributed File Systems in cloud computing, while it also treats the Andrew File System and the Google File System as examples of modern distributed file systems. As Distributed Scheduling is an interesting field of research, and could possibly gain more significance as cloud and grid computing become increasingly important, we have decided to include this topic in our paper. In the third chapter, we take a closer look at a specific programming paradigm and its corresponding framework which facilitate writing distributed programs.

Chapter 1

Distributed File Systems

1.1 Definition

A distributed file system enables a host in a computer network to gain access to files that are located on a different host in the same network. Normally this means that the participating users working on those hosts can share files. In most implementations the network related part of the file system is completely abstracted so that there is no discernible difference in interaction comparing to a local file system. The client does not have direct block level access to the storage system of the server instance, although this would indeed be possible and such implementations do exist. However, these file systems belong to the field of cluster file systems and are thus out of scope of this paper. The interaction flow with a distributed file system is transported over a network layer, normally over a classic berkeley socket connection. Since the service of a distributed file system is visible to all computers connected to the same network, access rights and restrictions are necessary. These can be achieved in the network layer with access control lists and/or capabilities. These rights and restrictions are not necessarily the same as those of the local file system on the server host. Since the data storage is not located at the client the opportunity arises to create facilities for transparent replication and fault tolerance. Therefore, if a number of m hosts exist in total, n hosts may go offline without interrupting live file system operation. In every case the following applies:

m	...number of online hosts
n	...number of failing hosts
x	...number of hosts needed for normal operation

$$m \geq 1$$
$$n \geq 1$$
$$n < m + x$$

Figure 1.1: Distributed File System Principles of Fault Tolerance

One problem that most currently existing distributed file systems share is the necessity of including a central head server which acts as an interface between the file system and the network. If this host goes down out of reasons that were planned or unplanned, the file system availability is interrupted.

1.2 Andrew File System

1.2.1 Features of AFS

AFS (The Andrew File System) can serve as a good example for a full-featured distributed (network) file system that is used in production environments in countless installations worldwide. It has its origin at the Carnegie Mellon University and was initially part of the Andrew Project, but was later developed as a standalone project. It is named after the founders of CMU, Andrew Carnegie and Andrew Mellon.
It uses a set of trusted servers to present a homogenous, location-transparent file name space to clients. This makes it possible to create a planet-spanning file system which is available even to roaming users. AFS has several benefits over traditional networked file systems, especially when it comes to security and scalability. It is not uncommon for enterprise AFS cells to exceed fifty thousand clients, which would hardly be possible with legacy file systems. AFS uses Kerberos for authentication and implements access control lists on directories for users and groups. For increased speed, each client caches files on the local file system if they are repeatedly requested. This also allows limited filesystem access in the event of a server crash or a network outage. The possibility of offline usage is a side effect of this design. IO operations on an open file are directed only to the locally cached copy. When a modified file is closed, the changed chunks are copied back to the file server in case it is available, otherwise this happens when the file server goes back on-line. Cache consistency is maintained by a mechanism called *callback*. The server registers cached files and informs the clients concerned if the file is updated by someone else. Callbacks are discarded and must be re-established after any client, server, or network failure, including a time-out. Re-establishing a callback involves a status check and does not require the re-reading of the file itself.
A consequence of the file locking strategy is that AFS does not support large shared databases or record updating within files shared between client systems. This was a deliberate design decision based on the perceived needs of the university's computing environment. It leads, for example, to the use of a single file per message in the original email system for the Andrew Project, the Andrew Message System, rather than a single file per mailbox.
A significant feature of AFS is the volume, which is a tree of files, sub-directories and AFS mount points (links to other AFS volumes). Volumes are created by administrators and linked to a specific named path in an AFS cell. Once they have been created, users of the filesystem may create directories and files as usual without having to be concerned about the physical location of the volume. A volume may have a quota assigned to it in order to limit the amount of space consumed. If required, AFS administrators can move that volume to another server and disk location without the obligation of notifying users; indeed the operation can occur while files in that volume are being used.
Replication of AFS volumes as read-only cloned copies and possible. When accessing files in a read-only volume, a client system will retrieve data from a particular read-only copy. If at some point that copy becomes unavailable, clients will look for any of the remaining copies. Again, users of that data are unaware of the location of the

read-only copy; administrators can create and relocate such copies as needed. The AFS command suite guarantees that all read-only volumes are exact copies of the original read-write volume. However, this does not imply that changes on the the original are passed on to its copies.
The file name space on an Andrew workstation is partitioned into a shared and local name space. The shared name space (usually mounted as /afs on the Unix filesystem) is identical on all workstations. The local name space is unique to each workstation. It only contains temporary files needed for workstation initialization and symbolic links to files in the shared name space.
The Andrew File System heavily influenced Version 4 of Sun Microsystems' popular Network File System (NFS). Additionally, a variant of AFS, the Distributed File System (DFS) was adopted by the Open Software Foundation in 1989 as part of their Distributed computing environment.

1.2.2 Implementations of AFS

There are three major implementations, namely Transarc (IBM), OpenAFS and Arla, although the Transarc software is losing support and is deprecated. AFS (version two) is also the predecessor of the Coda file system.
A fourth implementation exists in the Linux kernel source code since version 2.6.10. Contributed by Red Hat, which is a fairly simple implementation still in its early stages of development and therefore incomplete.

1.3 Google File System

1.3.1 Design of GFS

GFS is optimized for Google's core data storage and usage needs (primarily the search engine), which can generate enormous amounts of data that need to be retained; Google File System grew out of an earlier Google effort called "BigFiles", developed by Larry Page and Sergey Brin in the early days of Google, while it was still located in Stanford. Files are divided into chunks of 64 megabytes, which are extremely rarely overwritten, or shrunk; files are usually appended to or read. It is also designed and optimized to run on Google's computing clusters, the nodes of which consist of cheap, "commodity" computers. Consequently, precautions must be taken against the high failure rate of individual nodes and subsequent data loss. Other design decisions opt for high data throughput at the cost of high latency.
Nodes are divided into two types: one Master node and a large number of chunkservers. The latter store the data files, with each individual files broken up into 64 megabytes fixed-size chunks (hence the name), in a similar manner to clusters or sectors in regular file systems. Each chunk is assigned a unique 64-bit label, while logical mappings of files to constituent chunks are maintained. Each chunk is replicated at least three times throughout the network. Files which have a high demand rate or those in need of more redundancy, are replicated more often.
The Master server does not usually store the actual chunks, but rather all the metadata associated with the chunks, such as the tables mapping the 64-bit labels to chunk locations and the files they make up, the locations of the copies of the chunks, what

processes are reading or writing to a particular chunk, or taking a "snapshot" of the chunk pursuant to replicating it (usually at the instigation of the Master server, when, due to node failures, the number of copies of a chunk has fallen beneath the set number). The master server keeps all metadata up to date by periodically receiving updates from each chunk server, in the form of Heart-Beat messages.
Permissions for modifications are handled by a system of time-limited, expiring "leases", where the Master server grants sole permission to a process for a finite period of time to modify the chunk, during which no other process will be allowed to interfere. The modified chunkserver, which is always the primary chunk holder, then passes the changes on to the chunkservers together with the backup copies. The changes are not saved until all chunkservers acknowledge them, thus guaranteeing the completion and atomicity of the operation.
Programs access the chunks by first querying the Master server for the locations of the desired chunks; if the chunks are not being operated on (if there are no leases pending), the Master provides the locations. Then the program contacts and receives the data from the chunkserver directly (similarly to Kazaa and its supernodes).
As opposed to many filesystems, GFS is not implemented in the kernel of an operating system, but is instead provided as a userspace library.

1.4 Comparison AFS vs. GFS

Both of these distributed filesystems share the same basic idea of having a number of chunk servers that actually do the serving of block data, although the implementations are very different. The reason for this is that both implementations target completely different fields of distributed computing. Both can serve as basic examples of the particular fields, maybe even as reference implementations. The large environments in which these file systems are being run and the durability they have already shown serve as proof for a good design. Since Google FS is unfortunately not available to the public, impartial tests and comparisons are not possible. Currently it is not clear if this situation will ever change. However, a direct confrontation would be difficult anyway because the Google filesystem does not follow the given guidelines of a classic file system, since this would not have been of great benefit to Google and would have made the design unnecessarily complex.
There cannot be a *winner* in this comparison, the rivals are too different. Both do their job well and surpass their competitors in many disciplines. It is possible that we will see a variation of Google FS in the future which is available to everybody and easy to use.

Chapter 2

Distributed Scheduling

2.1 Definition

Distributed Scheduling can be shortly described as “How can we execute a set of tasks T on a set of processors P subject to some set of optimizing criteria C?”. The most common optimizing criterion is the minimization of execution time. Other criteria like “minimal cost”, “minimal communication delay” or “priority” can be configured for special applications.

The scheduling problem can be divided into two parts: algorithms, which concentrate on the policy and scheduling systems, which provide mechanisms to implement algorithms. In this document Condor is introduced as a representation for distributed scheduling systems.

2.2 Condor

“Condor is a job scheduler developed by the Condor research project. Like other full featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority, scheme, resource monitoring and resource management. Users submit their jobs to Condor, and Condor places them in a queue, chooses when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion."[13]

In this section, the most important principles of Condor will be discussed. For a detailed description visit the website of Condor project’s `http://www.cs.wisc.edu/condor`.

2.2.1 Features

Distributed Submission: There is no central submission machine. Many machines can be used to submit jobs (even from their own desktop machine) and hold their own job queue.

Job Priorities: Users can assign priorities to their jobs to take effect on the execution of them. “Nice users” request only those machines which would otherwise be idle.

User Priorities: Administrators can assign policies to users.

Job Dependence: In many cases some jobs must be finished to start others. Condor provide mechanisms to execute the jobs in a correct sequence.

Support for multiple Job Models: Condor handles serial jobs and parallel jobs. See section 2.2.3 for more details.

Class Ads: The Class Ads are mechanisms to publish specifications of the kind of machines needed or offered. Class Ads are also able to match jobs to machines and prioritize them.

Job Checkpoints and Migration: Condor can take snapshots of jobs and subsequently resume the application. This allows a paused job to continue later or to migrate to other machines.

Periodic Checkpoint: Condor can be configured to periodically take checkpoints. It reduces the loss in the event of system failure such as the machine being shut down or hardware failure.

Job Suspend and Resume: On the basis of policies, Condor can ask the operating system to suspend and later resume jobs.

Remote System Calls: Condor can use Remote System Calls to prevent problems when jobs use files and resources. With this mechanism users do not need user accounts or permissions on the machine where the job is executed. To the user the program appears to be running on the his desktop machine, where it was originally written.

Pools of Machines can Work Together: It is possible to run jobs across multiple Condor pools. Jobs can be executed across pools of machines owned by different organizations.

Authentication and Authorization: Administrators have detailed control over access permissions and Condor can perform network authentication using mechanisms such as Kerberos or public key certificates.

Heterogeneous Platforms: A single pool can contain different platforms such as Linux, Unix or Windows NT. A job running on Windows can be submitted from a machine running on Linux.

Grid Computing: Condor incorporates many Grid-based computing methodologies and protocols. It can interact with resources managed by Globus.

2.2.2 Class Ads

`Class Ads` are like advertising sections in newspapers. Sellers and buyers place their ads, so they can find their counterpart and prioritize the offers.

Sellers offer information about their products they want to sell. Buyers post information about the products they want to buy. For Condor, sellers are machine owners and buyers are users who want a job to be executed.

In Condor, Class Ads are stored in expressions of key- value- pairs. A simple example of this is shown in Figure 2.1.

The values of these expressions can also consist of mathematical- and logical expressions which can be evaluated in boolean values. In contrast to boolean variables used in programming languages, they can have three different values:

```
MemoryInMegs = 512
OpSys = "LINUX"
NetworkLatency = 7.5
```

Figure 2.1: Class Ads structure

- `TRUE`
- `FALSE`
- `UNDEFINED`

When there is a lack of information to have the expressions assessed as either `TRUE` or `FALSE`, then the expression is simply assessed as `UNDEFINED`. Figure 2.2 shows some examples of expressions and their values.

```
1 MemoryInMegs = 512
2 MemoryInBytes = MemoryInMegs * 1024 * 1024
3 CPUs = 4
4 BigMachine = (MemoryInMegs > 256) && (CPUs >= 4)
5 VeryBigMachine = (MemoryInMegs > 512) && (CPUs >= 8)
6 FastMachine = BigMachine && SpeedRating
```

Figure 2.2: Class Ads Expressions

The expression in line 4 is assessed as `TRUE`; the one in line 5 is assessed as `FALSE`; and the one in line 6 will be assessed as `UNDEFINED`, because the expression “SpeedRating” is not defined.

If there is more than one Class Ad which matches the original one, then the expression with the key “Rank” comes into effect. The Class Ad which Rank-expression judges to be the best result will be preferred. Figure 2.3 shows an example of this principle.

```
Requirements = ((Arch=="Intel" && OpSys=="LINUX")
         && Disk > DiskUsage)
Rank = (Memory * 100000) + KFlops
```

Figure 2.3: Class Ads Ranking

Finally, Class Ads can be matched with one another. Figure 2.4 shows two extracts of two matching Class Ads. One represents a queued job (user-request) and the other a machine. The `MyType` expression specifies what type of resource a Class Ad is and the `TargetType` specifies what type the counterpart wanted should be.

These Class Ads only match if both `Requirements` expressions are assessed as `TRUE`. The expressions of a Class Ad can refer not only to the one in that ad, they can also refer to expressions of the target ad. In this example the `Requirements` expression of the job refers to the `OpSys` expression of the machines ad. If there are more than one matching Class Ads, then the `Rank` expressions are employed. Else if no machine matches the requirements of a queued job, the job can not be executed.

```
MyType = "Job"                      | MyType = "Machine"
TargetType = "Machine"              | TargetType = "Job"
Requirements = ((Arch=="INTEL"      | Requirements = Start
 && OpSys=="LINUX")                 | Rank = TARGET.Department
 && Disk > DiskUsage)               |  == MY.Department
Rank = (Memory * 10000) + Kflops    | Activity = "Idle"
...                                 | Arch = "INTEL"
                                    | ...
```

Figure 2.4: Class Ads Matching

2.2.3 Universes

In Condor, runtime environments are called “Universe”. For every job, the best fitting Universe has to be selected. Condor supports six universes, namely “Vanilla”, “MPI”, “PVM”, “Globus”, “Scheduler” and the “Standard” Universe. If no Universe is specified, the “Standard” Universe is chosen automatically.

Vanilla Universe

The Vanilla Universe is used to run non-parallel jobs. The Standard Universe provides several helpful features, but has also some restrictions on the type of jobs to be executed. The vanilla universe does not have such restrictions at all. Every job that runs outside of Condor, runs in the Vanilla Universe.

MPI Universe

As the name implies, the MPI Universe executes jobs by employing the Message Passing Interface. One has to specify the number of nodes to be used in the parallel job. The attribute `machine_count` is used for this.

If the Condor pool consists of dedicated compute machines (which would be Beowulf cluster nodes) and opportunistic machines (such as desktop workstations) by default, Condor will schedule jobs on dedicated machines only.

PVM Universe

The PVM Universe allows jobs written in master-worker style to be executed. One master gets a pool of tasks and sends pieces of work out to the other worker jobs.

In this universe, dedicated compute machines and opportunistic machines can be added to and removed from the Parallel-Virtual-Machine (PVM).

Globus Universe

The Globus Universe is the Condor interface for users who wish to submit to machines managed by Globus.

For more information about Globus go to the website of the Globus Alliance (`http://www.globus.org/`).

Scheduler Universe

In the Scheduler Universe the submitted job will be immediately executed on the *submit* machine, as opposed to a remote execution machine. The purpose is to provide facility for job *meta-schedulers* which manage the submission and the removal of jobs to and from Condor. The scheduler “DAGMan” is an implementation of such a meta-scheduler.

The DAGMan scheduler is designed to manage complicated dependencies of jobs in Condor via creating a directed acyclic graph (DAG). The input, output or execution of the programs depends on one or more programs. Figure 2.5 shows an example file for DAGMan.

```
# file name: diamond.dag
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Figure 2.5: Diamond example of the input file for DAGMan

Standard Universe

The Standard Universe requires minimal extra effort to schedule jobs, but provides several helpful features:

- Transparent process *checkpoint* and *restart*
- Transparent process migration
- Remote System Calls
- Configurable file I/O buffering
- On-the-fly file compression/inflation

Condor operates explicitly in user mode (no kernel driver is needed). This leads to some limitations in checkpoints and process states in the Standard Universe:

- Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()` and `system()`.
- Interprocess communication is not permitted. This includes pipes, semaphores and shared memory.
- Network communication must be brief. A job *may* take network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpoints and migration.
- Multiple kernel-level threads are not allowed. However, multiple user-level threads (green threads) *are* allowed.
- All files should be accessed read-only or write-only. A file which is both read and written to could cause trouble if a job must be rolled back to a previous checkpoint image.

- On Linux, your job must be statically linked. Dynamic linking is allowed in the Standard Universe on some other platforms supported by Condor, and perhaps this restriction on Linux will be removed in a future Condor release.

Chapter 3

Distributed Application Frameworks

Of the frameworks currently available, this paper discusses the Google MapReduce framework [6] for two reasons. On the one hand, MapReduce employs the Google File System for managing data, on the other hand, it incorporates some aspects of the Condor Project for scheduling jobs submitted to MapReduce.

3.1 Definition

MapReduce is a programming model and a corresponding software framework developed by Google. Its main purpose is processing and generating large data sets on a substantial number of computers, commonly referred to as a cluster. Dealing with issues such as parallelization and scalability in every single program results in code redundancy, high complexity and waste of resources. By providing a powerful abstraction, MapReduce allows the writing of distributed programs without having to deal with

- parallelization
- scalability
- fault-tolerance
- data distribution
- and load balancing

3.2 Concept

Separating the program's logic from its distribution leads to simpler, smaller and more efficient programs which are easier to understand. In order to achieve this MapReduce provides a strong as well as simple interface. A programmer who uses the MapReduce library to write distributed code simply expresses the program's logic by the means of two functions:

- **Map function:**
  This function processes a key/value pair and generates a set of intermediate key/-value pairs.
  `map (k1,v1)` $\rightarrow$ `list(k2,v2)`

- **Reduce function:**
  This function merges all intermediate values associated with the same intermediate key.
  `reduce (k2,list(v2))` $\rightarrow$ `list(v2)`

Apart from some additional configurations, the programmer only concerns himself with the interface mentioned above. Messy details of the distribution are hidden by the framework.

3.3 Example

To illustrate the usage of the functions defined above, the following example is given, in which the appearances of different words in a large collection of documents is counted:

```
1  map(String key, String value):
2          // key: document name
3          // value: document contents
4          for each word w in value:
5                  EmitIntermediate(w, "1");
6
7  reduce(String key, Iterator values):
8          // key: a word
9          // values: a list of counts
10         int result = 0;
11         for each v in values:
12         result += ParseInt(v);
13         Emit(AsString(result));
```

Figure 3.1: MapReduce Example in pseudo-code

The map function has two parameters. The key represents a specific document, whereas the value represents its contents. In lines 4 and 5, for each word of the document an intermediate key/value pair, consisting of the word (key) and "1" (value), is generated.
The reduce function accepts an intermediate key, in this case a single word, and a list of intermediate values that are associated with this particular key. The list of intermediate values consists of "1"s. In line 11 and 12 the list of ones is summed up. Eventually, the result is returned.

3.4 Implementation Details

3.4.1 Node Types

Three different kinds of cluster-nodes are distinguished in the MapReduce environment:

- **Master**
  MapReduce jobs are managed by the master. It is in control of a pool of worker nodes and distributes the jobs to the workers. There is only one master node.

- **Worker nodes**
  The master assigns either map tasks or reduce tasks to the workers, therefore two different kinds of worker nodes exist:

  - Map Workers
  - Reduce Workers

3.4.2 Execution Flow

Figure 3.2 on page 14 illustrates the execution flow of a MapReduce operation, the execution of which can be split in eight steps. The numbers of the enumeration below are consistent with the labels in Figure 3.2.

1. In order to parallelize the processing of input data, the input has to be split into pieces, which can then be easily distributed and processed.

2. The master assigns map and reduce tasks to idle worker nodes.

3. A map worker reads the content of the assigned input split, extracts the input pairs and passes them on to the user-defined map function.

4. The output of the map function (intermediate key/value pairs) is buffered in the machine's memory and periodically written to the local disk.

5. The location of the intermediate pairs are communicated to the master, which partitions the intermediate data across tasks, using a partitioning function on the intermediate keys.

6. After the reduce worker has been told by the master which intermediate pairs to get, the reduce worker uses RPCs to read those pairs from the local disks of the map workers. Once the reduce worker has read all corresponding pairs, these are sorted by their intermediate keys.

7. The user-defined reduce function is called for each intermediate key and its corresponding values.

8. The output of a reduce worker is appended to a single output file.

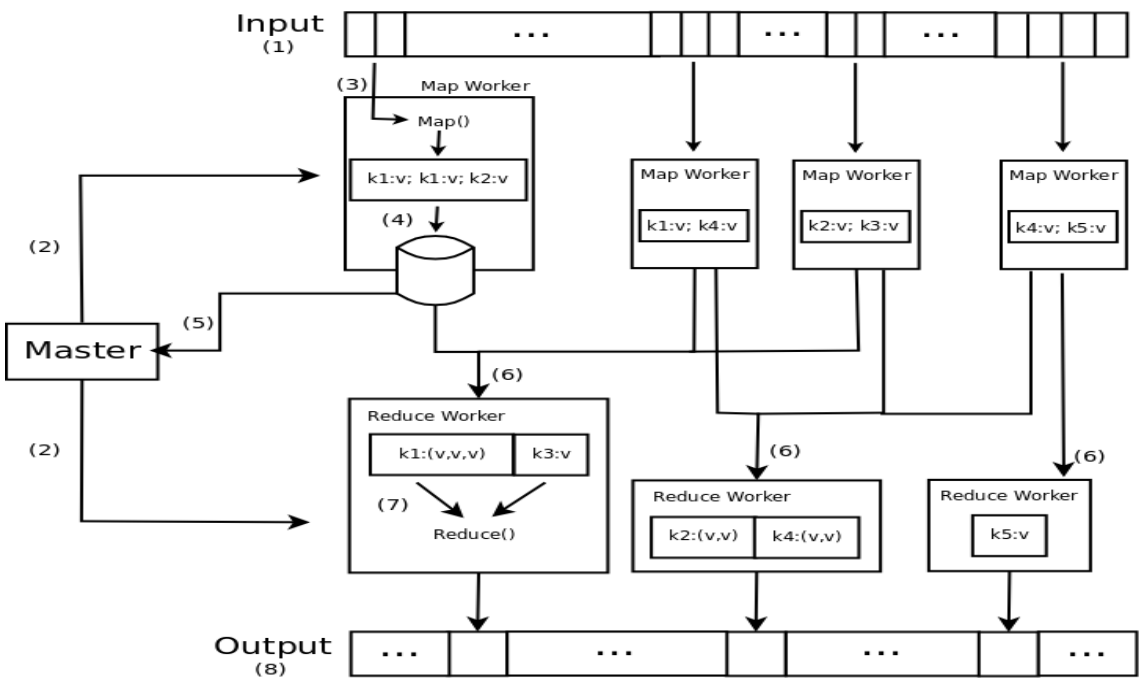

Figure 3.2: MapReduce Execution Flow

3.4.3 Communication Details

As network bandwidth is a limited resource in almost every cluster, communication between nodes is an issue which requires a solution. In order to reduce the communication overhead, the MapReduce framework incorporates various mechanisms that minimize or avoid communication and thus saves network bandwidth. An example for a mechanism that minimizes communication would be piggybacking. The MapReduce master constantly pings all of its workers. Some of the data that has to be transferred from a worker to the master, is piggybacked on a ping response. By using the principle of locality, communication can be reduced considerably or altogether avoided. Input data splits are stored on the local disks of the nodes using GFS. The master keeps track of the input split's location information and tries to assign a specific map task to one of those workers that have the corresponding input split on their local disk. If this is not possible, the master attempts to schedule the map task on a machine near the input split's location.

3.5 Common Issues of Distribution

As mentioned in section 3.1 on page 11, MapReduce permits the writing of distributed code that avoids some issues which usually lead to code redundancy and great complexity. This section is concerned with an in-depth examination of these issues and how they are handled by MapReduce.

3.5.1 Parallelization

The map as well as the reduce function operate on data splits which can be processed in parallel. As outlined in section 3.2 on page 11, user-defined map and reduce functions are distributed among the worker nodes and the code is then locally executed on these machines.

3.5.2 Scalability

The two user-defined functions - map and reduce - are written in such a way, that it does not matter if they are run on just one machine or a thousand. The master passes the code on to all worker nodes. The adding of worker nodes to the master's pool of workers is also facilitated.

3.5.3 Fault-Tolerance

In MapReduce, there are different levels of failure handling.

- Data Failure:
  MapReduce utilizes the Google File System to store large data sets on commodity machines. As already mentioned (see section 1.3 on page 3), GFS is a distributed file system running on unreliable commodity hardware and therefore contains failure handling mechanisms.

- Master Failure:
  Given the fact that the master is the central point of communication and management, it is also the central point of failure. If the master fails the whole system fails. Therefore, special steps have to be taken to ensure that a potential failure of the master has as little impact on the system as possible. One possibility would be to have the master periodically writing checkpoints of its state. In case the master fails, another node can be instantiated as master, resuming work from the last checkpoint.

- Worker Failure:
  The master checks regularly on its workers to see if they are still responding. If a worker does not reply within a certain amount of time, the master assumes that it has failed and reschedules its tasks on other workers. It has to be noted that tasks of a failed worker are re-executed from scratch, due to the inaccessibility of the local disk of a failed machine.

3.5.4 Data Distribution

The initial splitting of the input data is done by the MapReduce library. The user determines the size of an input split via an optional parameter, which can vary between 16 and 64 MB. The Google File System handles the physical distribution of the data.

3.5.5 Load Balancing

Users file their MapReduce jobs with the master, which incorporates a scheduling system. These jobs comprise a set of tasks which are then mapped to the available worker nodes by the scheduler. A "straggler" is a worker node that takes up an unusually large amount of time for completing a task. A worker can develop into a straggler as a result of internal failures or a workload that is too high. If the master detects one, it launches a backup task, meaning that it assigns the straggler's tasks to another worker so that they simultaneously work on the same assignment. As soon as one of them has finished, the other one is terminated.

Bibliography

[1] Luiz Andre Barroso, Jeffrey Dean, and Urs Hoelzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar./Apr. 2003.

[2] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *First USENIX Symposium on Networked Systems Design and Implementation*, 2004.

[3] Steve J. Chapin. Distributed and multiprocessor scheduling. *ACM Comput. Surv.*, 28(1):233–235, 1996.

[4] Steve J. Chapin. Cloud comp. *Commun. ACM*, 51(7):9–11, 2008.

[5] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 150–160, New York, NY, USA, 1994. ACM.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[7] Kemal A. Delic and Martin Anthony Walker. Emergence of the academic computing clouds. *Ubiquity*, 9(31):1–1, 2008.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[9] Brian Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[10] T. V. Raman. Cloud computing and equal access for all. In *W4A '08: Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A)*, pages 1–4, New York, NY, USA, 2008. ACM.

[11] M. Satyanarayanan and Mirjana Spasojevic. Afs and the web: competitors or collaborators? In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 89–94, New York, NY, USA, 1996. ACM.

[12] Mirjana Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Trans. Comput. Syst.*, 14(2):200–222, 1996.

[13] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[14] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.