6.3 Stability of a two-level resource allocation architecture

In Section 6.2 we saw that we can assimilate a server with a closed-loop control system and we can apply control theory principles to resource allocation. In this section we discuss a two-level resource

[4]Pontryagin's principle is used in the optimal control theory to find the best possible control that leads a dynamic system from one state to another, subject to a set of constraints.

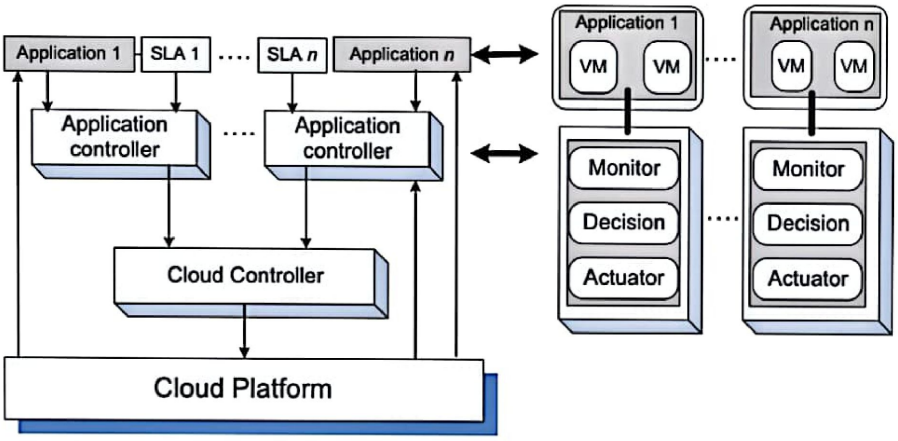


FIGURE 6.2

A two-level control architecture. Application controllers and cloud controllers work in concert.

allocation architecture based on control theory concepts for the entire cloud. The automatic resource management is based on two levels of controllers, one for the service provider and one for the application, see Figure 6.2.

The main components of a control system are the inputs, the control system components, and the outputs. The inputs in such models are the offered workload and the policies for admission control, the capacity allocation, the load balancing, the energy optimization, and the QoS guarantees in the cloud. The system components are *sensors* used to estimate relevant measures of performance and *controllers* that implement various policies; the output is the resource allocations to the individual applications .

The controllers use the feedback provided by sensors to stabilize the system; stability is related to the change of the output. If the change is too large, the system may become unstable. In our context the system could experience thrashing, the amount of useful time dedicated to the execution of applications becomes increasingly small and most of the system resources are occupied by management functions.

There are three main sources of instability in any control system:

1. The delay in getting the system reaction after a control action.
2. The granularity of the control, the fact that a small change enacted by the controllers leads to very large changes of the output.
3. Oscillations, which occur when the changes of the input are too large and the control is too weak, such that the changes of the input propagate directly to the output.

Two types of policies are used in autonomic systems: (i) threshold-based policies and (ii) sequential decision policies based on Markovian decision models. In the first case, upper and lower bounds on performance trigger adaptation through resource reallocation. Such policies are simple and intuitive but require setting per-application thresholds.

Lessons learned from the experiments with two levels of controllers and the two types of policies are discussed in [109]. A first observation is that the actions of the control system should be carried out in

a rhythm that does not lead to instability. Adjustments should be carried out only after the performance of the system has stabilized. The controller should measure the time for an application to stabilize and adapt to the manner in which the controlled system reacts.

If upper and lower thresholds are set, instability occurs when they are too close to one another if the variations of the workload are large enough and the time required to adapt does not allow the system to stabilize. The actions consist of allocation/deallocation of one or more virtual machines; sometimes allocation/deallocation of a single VM required by one of the thresholds may cause crossing of the other threshold and this may represent, another source of instability.

6.9 Fair queuing

Computing and communication on a cloud are intimately related. Therefore, it should be no surprise that the first algorithm we discuss can be used for scheduling packet transmission as well as threads. Interconnection networks allow cloud servers to communicate with one another and with users. These networks consist of communication links of limited bandwidth and switches/routers/gateways of limited capacity. When the load exceeds its capacity, a switch starts dropping packets because it has limited input buffers for the switching fabric and for the outgoing links, as well as limited CPU cycles.

A switch must handle multiple flows and pairs of source-destination endpoints of the traffic. Thus, a scheduling algorithm has to manage several quantities at the same time: the *bandwidth*, the amount of data each flow is allowed to transport; the *timing* when the packets of individual flows are transmitted; and the *buffer space* allocated to each flow. A first strategy to avoid network congestion is to use a FCFS scheduling algorithm. The advantage of the FCFS algorithm is a simple management of the three quantities: bandwidth, timing, and buffer space. Nevertheless, the FCFS algorithm does not guarantee fairness; greedy flow sources can transmit at a higher rate and benefit from a larger share of the bandwidth.

To address this problem, a fair queuing algorithm proposed in [252] requires that separate queues, one per flow, be maintained by a switch and that the queues be serviced in a round-robin manner. This algorithm guarantees the fairness of buffer space management, but does not guarantee fairness of bandwidth allocation. Indeed, a flow transporting large packets will benefit from a larger bandwidth (see Figure 6.8).

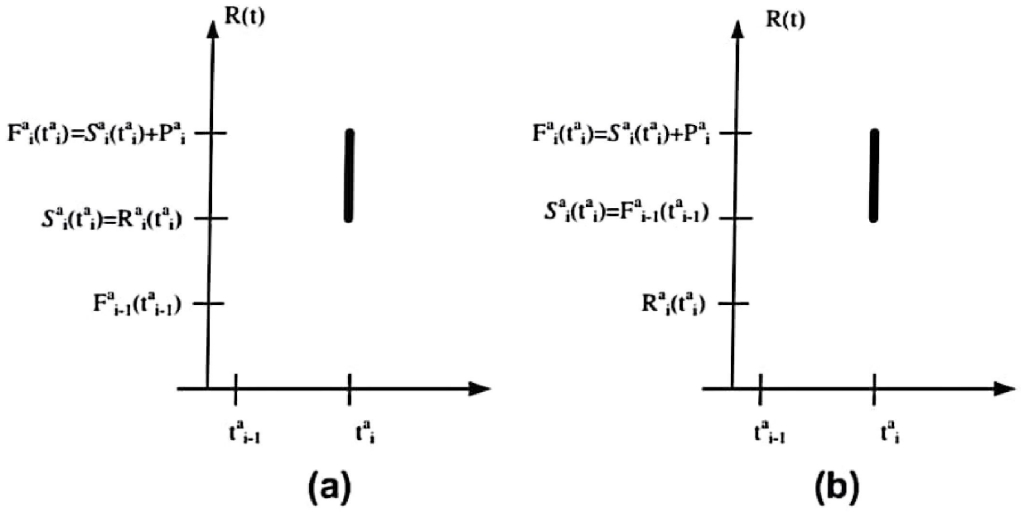
The *fair queuing (FQ)* algorithm in [102] proposes a solution to this problem. First, it introduces a *bit-by-bit round-robin (BR)* strategy; as the name implies, in this rather impractical scheme a single bit from each queue is transmitted and the queues are visited in a round-robin fashion. Let $R(t)$ be the number of rounds of the BR algorithm up to time t and $N_{active}(t)$ be the number of active flows through the switch. Call t_i^a the time when the packet i of flow a, of size P_i^a bits arrives, and call S_i^a and F_i^a the values of $R(t)$ when the first and the last bit, respectively, of the packet i of flow a are transmitted. Then,

$$F_i^a = S_i^a + P_i^a \quad \text{and} \quad S_i^a = \max\left[F_{i-1}^a, R(t_i^a)\right]. \tag{6.28}$$

The quantities $R(t)$, $N_{active}(t)$, S_i^a, and F_i^a depend only on the arrival time of the packets, t_i^a, and not on their transmission time, provided that a flow a is active as long as

$$R(t) \leqslant F_i^a \quad \text{when} \quad i = \max\left(j | t_i^a \leqslant t\right). \tag{6.29}$$

The authors of [102] use for packet-by-packet transmission time the following nonpreemptive scheduling rule, which emulates the BR strategy: *The next packet to be transmitted is the one with the smallest* F_i^a. A preemptive version of the algorithm requires that the transmission of the current packet be interrupted as soon as one with a shorter finishing time, F_i^a, arrives.



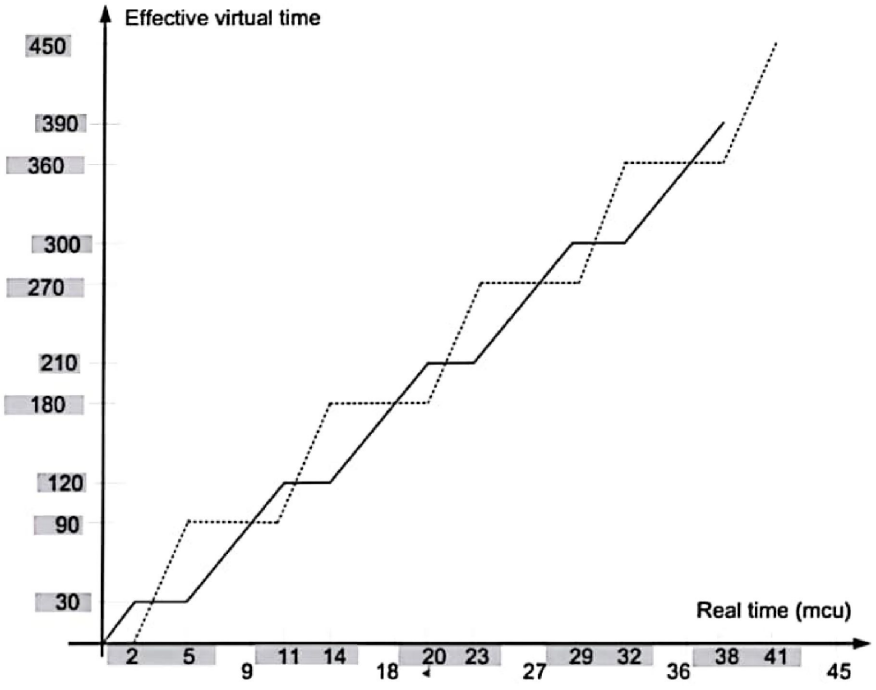


FIGURE 6.11

Example 1, the effective virtual time and the real time of threads *a* (solid line) and *b* (dotted line) with weights $w_a = 2w_b$ when the actual virtual time is incremented in steps of $\Delta = 90$ *mcu*. The real time the two threads are allowed to use the CPU is proportional to their weights. The virtual times are equal, but thread *a* consumes it more slowly. There is no time warp. The threads are dispatched based on their actual virtual time.

to run for its time allocation. The scheduler compares the effective virtual time of the threads and first runs the one with the minimum effective virtual time.

Figure 6.11 displays the effective virtual time and the real time of threads a and b. When a thread is running, its effective virtual time increases as the real time increases; a running thread appears as a diagonal line. When a thread is runnable but not running, its effective virtual time is constant. A runnable period is displayed as a horizontal line. We see that the two threads are allocated equal amounts of virtual time, but thread a, with a larger weight, consumes its real time more slowly.

6.13 Scheduling *MapReduce* applications subject to deadlines

Now we turn our attention to applications of the analysis in Section 6.12 and discuss scheduling of *MapReduce* applications on the cloud subject to deadlines. Several options for scheduling Apache *Hadoop*, an open-source implementation of the *MapReduce* algorithm, are:

- The default FIFO schedule.
- The Fair Scheduler [383].
- The Capacity Scheduler.
- The Dynamic Proportional Scheduler [315].

A recent paper [186] applies the deadline scheduling framework analyzed to *Hadoop* tasks. Table 6.8 summarizes the notations used for the analysis of *Hadoop*; the term *slots* is equivalent with *nodes* and means the number of instances.

We make two assumptions for our initial derivation:

- The system is homogeneous; this means that ρ_m and ρ_r, the cost of processing a unit data by the *map* and the *reduce* task, respectively, are the same for all servers.
- Load equipartition.

Under these conditions the duration of the job J with input of size σ is

$$\mathcal{E}(n_m, n_r, \sigma) = \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right]. \tag{6.86}$$

Table 6.8 The parameters used for scheduling with deadlines.

Name	Description
Q	The query $Q = (A, \sigma, D)$
A	Arrival time of query Q
D	Deadline of query Q
Π_m^i	A map task, $1 \leqslant i \leqslant u$
Π_r^j	A reduce task, $1 \leqslant j \leqslant v$
J	The job to perform the query $Q = (A, \sigma, D)$, $J = \left(\Pi_m^1, \Pi_m^2, \ldots, \Pi_m^u, \Pi_r^1, \Pi_r^2, \ldots, \Pi_r^v\right)$
τ	Cost for transferring a data unit
ρm	Cost of processing a unit data in map task
ρ_r	Cost of processing a unit data in reduce task
n_m	Number of map slots
n_r	Number of reduce slots
n_m^{min}	Minimum number of slots for the map task
n	Total number of slots, $n = n_m + n_r$
t_m^0	Start time of the map task
t_r^{max}	Maximum value for the start time of the reduce task
α	Map distribution vector; the EPR strategy is used and, $\alpha_i = 1/u$
ϕ	Filter ratio, the fraction of the input produced as output by the map process

Thus, the condition that query $Q = (A, \sigma, D)$ with arrival time A meets the deadline D can be expressed as

$$t_m^0 + \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right] \leqslant A + D. \tag{6.87}$$

It follows immediately that the maximum value for the start-up time of the *reduce* task is

$$t_r^{max} = A + D - \sigma\phi \left(\frac{\rho_r}{n_r} + \tau \right). \tag{6.88}$$

We now plug the expression of the maximum value for the start-up time of the *reduce* task into the condition to meet the deadline

$$t_m^0 + \sigma \frac{\rho_m}{n_m} \leqslant t_r^{max}. \tag{6.89}$$

It follows immediately that n_m^{min}, the minimum number of slots for the *map* task, satisfies the condition

$$n_m^{min} \geqslant \frac{\sigma\rho_m}{t_r^{max} - t_m^0}, \quad \text{thus,} \quad n_m^{min} = \lceil \frac{\sigma\rho_m}{t_r^{max} - t_m^0} \rceil. \tag{6.90}$$

The assumption of homogeneity of the servers can be relaxed and we assume that individual servers have different costs for processing a unit workload $\rho_m^i \neq \rho_m^j$ and $\rho_t^i \neq \rho_t^j$. In this case we can use the minimum values $\rho_m = \min \rho_m^i$ and $\rho_r = \min \rho_r^i$ in the expression we derived.

A Constraints Scheduler based on this analysis and an evaluation of the effectiveness of this scheduler are presented in [186].

8.5 Google File System

The Google File System (GFS) was developed in the late 1990s. It uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs [136]. It is not surprising that a main concern of the GFS designers was to ensure the reliability of a system exposed to hardware failures, system software errors, application errors, and last but not least, human errors.

The system was designed after a careful analysis of the file characteristics and of the access models. Some of the most important aspects of this analysis reflected in the GFS design are:

- Scalability and reliability are critical features of the system; they must be considered from the beginning rather than at some stage of the design.
- The vast majority of files range in size from a few GB to hundreds of TB.
- The most common operation is to `append` to an existing file; random `write` operations to a file are extremely infrequent.
- Sequential `read` operations are the norm.
- The users process the data in bulk and are less concerned with the response time.
- The consistency model should be relaxed to simplify the system implementation, but without placing an additional burden on the application developers.

Several design decisions were made as a result of this analysis:

1. Segment a file in large chunks.
2. Implement an atomic file `append` operation allowing multiple applications operating concurrently to `append` to the same file.
3. Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate the flow of control from the data flow; schedule the high-bandwidth data flow by pipelining the data transfer over TCP connections to reduce the response time. Exploit network topology by sending data to the closest node in the network.
4. Eliminate caching at the client site. Caching increases the overhead for maintaining consistency among cached copies at multiple client sites and it is not likely to improve performance.
5. Ensure consistency by channeling critical file operations through a *master*, a component of the cluster that controls the entire system.
6. Minimize the involvement of the *master* in file access operations to avoid hot-spot contention and to ensure scalability.
7. Support efficient checkpointing and fast recovery mechanisms.
8. Support an efficient garbage-collection mechanism.

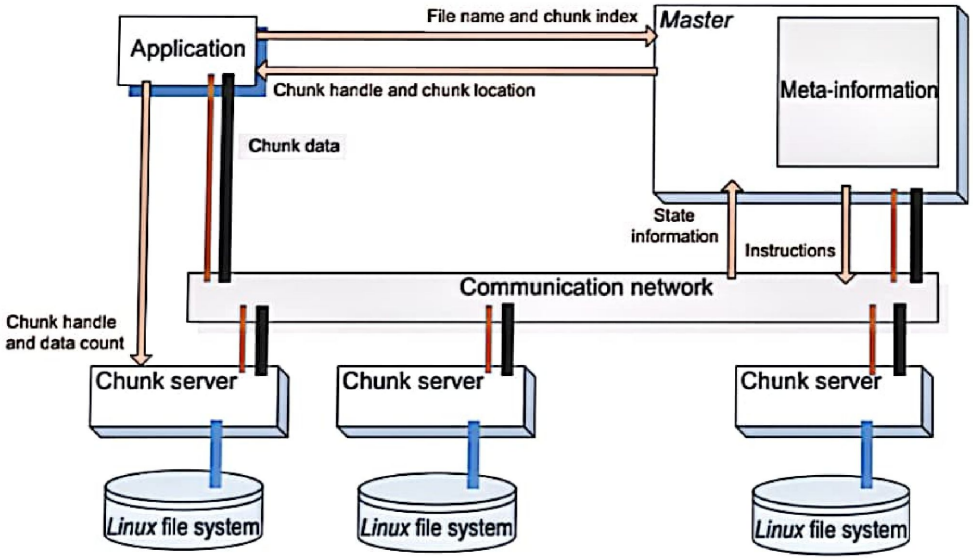
GFS files are collections of fixed-size segments called *chunks*; at the time of file creation each chunk is assigned a unique *chunk handle*. A chunk consists of 64 KB blocks and each block has a 32-bit checksum. Chunks are stored on *Linux* files systems and are replicated on multiple sites; a user may change the number of the replicas from the standard value of three to any desired value. The chunk size is 64 MB; this choice is motivated by the desire to optimize performance for large files and to reduce the amount of metadata maintained by the system.

A large chunk size increases the likelihood that multiple operations will be directed to the same chunk; thus it reduces the number of requests to locate the chunk and, at the same time, it allows the application to maintain a persistent network connection with the server where the chunk is located. Space fragmentation occurs infrequently because the chunk for a small file and the last chunk of a large file are only partially filled.

The architecture of a GFS cluster is illustrated in Figure 8.7. A *master* controls a large number of *chunk servers*; it maintains metadata such as filenames, access control information, the location of all the replicas for every chunk of each file, and the state of individual chunk servers. Some of the metadata is stored in persistent storage (e.g., the *operation log* records the file namespace as well as the file-to-chunk mapping).

The locations of the chunks are stored only in the control structure of the *master*'s memory and are updated at system startup or when a new chunk server joins the cluster. This strategy allows the *master* to have up-to-date information about the location of the chunks.

System reliability is a major concern, and the operation log maintains a historical record of metadata changes, enabling the *master* to recover in case of a failure. As a result, such changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage. To recover from a failure, the *master* replays the operation log. To minimize the recovery time, the *master* periodically checkpoints its state and at recovery time replays only the log records after the last checkpoint.



8.6 *Apache Hadoop*

A wide range of data-intensive applications such as marketing analytics, image processing, machine learning, and Web crawling use *Apache Hadoop*, an open-source, Java-based software system.[6] *Hadoop* supports distributed applications handling extremely large volumes of data. Many members of the community contributed to the development and optimization of *Hadoop* and several related Apache projects such as *Hive* and *HBase*.

Hadoop is used by many organizations from industry, government, and research; the long list of *Hadoop* users includes major IT companies such as Apple, IBM, HP, Microsoft, Yahoo!, and Amazon; media companies such as The New York Times and Fox; social networks, including Twitter, Facebook, and LinkedIn; and government agencies, such as the U.S. Federal Reserve. In 2011, the Facebook *Hadoop* cluster had a capacity of 30 PB.

A *Hadoop* system has two components, a *MapReduce* engine and a database (see Figure 8.8). The database could be the *Hadoop File System (HDFS)*, Amazon *S3*, or *CloudStore*, an implementation of the Google File System discussed in Section 8.5. *HDFS* is a distributed file system written in Java; it is portable, but it cannot be directly mounted on an existing operating system. *HDFS* is not fully POSIX compliant, but it is highly performant.

The *Hadoop* engine on the master of a multinode cluster consists of a *job tracker* and a *task tracker*, whereas the engine on a slave has only a *task tracker*. The *job tracker* receives a *MapReduce* job

[6] *Hadoop* requires Java Runtime Environment (JRE) 1.6 or higher.

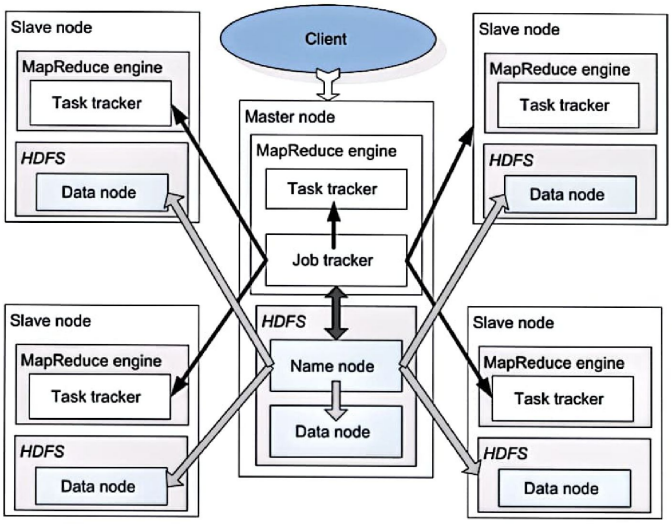


FIGURE 8.8

A *Hadoop* cluster using *HDFS*. The cluster includes a master and four slave nodes. Each node runs a *MapReduce* engine and a database engine, often *HDFS*. The *job tracker* of the master's engine communicates with the *task trackers* on all the nodes and with the *name node* of *HDFS*. The *name node* of *HDFS* shares information about data placement with the *job tracker* to minimize communication between the nodes on which data is located and the ones where it is needed.

from a client and dispatches the work to the *task trackers* running on the nodes of a cluster. To increase efficiency, the *job tracker* attempts to dispatch the tasks to available slaves closest to the place where it stored the task data. The *task tracker* supervises the execution of the work allocated to the node. Several scheduling algorithms have been implemented in *Hadoop* engines, including Facebook's fair scheduler and Yahoo!'s capacity scheduler; see Section 6.8 for a discussion of cloud scheduling algorithms.

HDFS replicates data on multiple nodes. The default is three replicas; a large dataset is distributed over many nodes. The *name node* running on the master manages the data distribution and data replication and communicates with *data nodes* running on all cluster nodes; it shares with the *job tracker* information about data placement to minimize communication between the nodes on which data is located and the ones where it is needed. Although *HDFS* can be used for applications other than those based on the *MapReduce* model, its performance for such applications is not at par with the ones for which it was originally designed.

8.9 *BigTable*

BigTable is a distributed storage system developed by Google to store massive amounts of data and to scale up to thousands of storage servers [73]. The system uses the Google File System discussed in Section 8.5 to store user data as well as system information. To guarantee atomic `read` and `write` operations, it uses the *Chubby* distributed lock service (see Section 8.7); the directories and the files in the namespace of *Chubby* are used as locks.

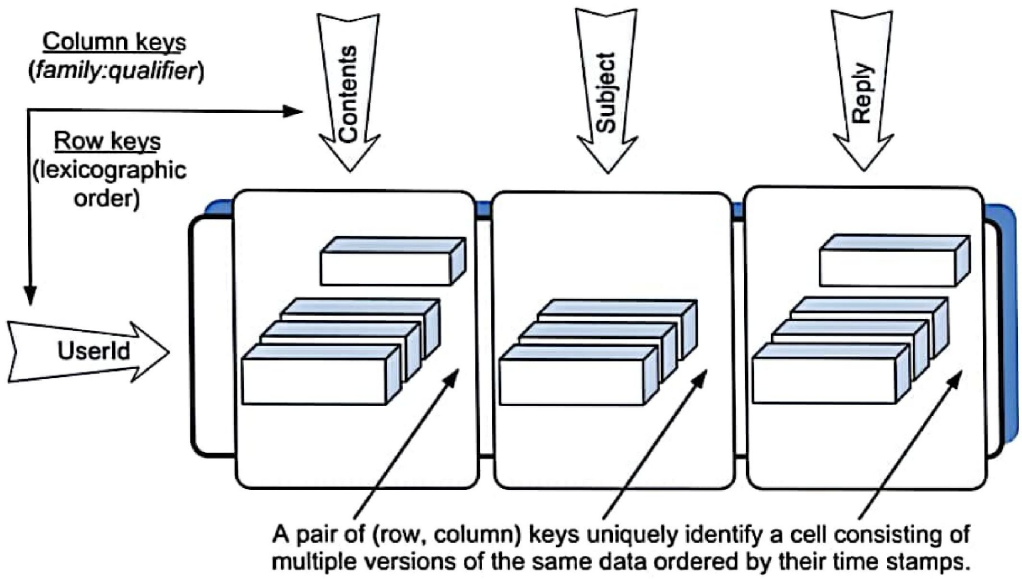
The system is based on a simple and flexible data model. It allows an application developer to exercise control over the data format and layout and reveals data locality information to the application clients. Any `read` or `write` row operation is atomic, even when it affects more than one column. The column keys identify *column families*, which are units of access control. The data in a column family is of the same type. Client applications written in C++ can add or delete values, search for a subset of data, and look up data in a row.

A row key is an arbitrary string of up to 64 KB, and a row range is partitioned into *tablets* serving as units for load balancing. The time stamps used to index various versions of the data in a cell are 64-bit integers; their interpretation can be defined by the application, whereas the default is the time of an event in microseconds. A column key consists of a string defining the family name, a set of printable characters, and an arbitrary string as qualifier.

The organization of a *BigTable* (see Figure 8.11) shows a sparse, distributed, multidimensional map for an email application. The system consists of three major components: a library linked to application clients to access the system, a master server, and a large number of tablet servers. The master server controls the entire system, assigns tablets to tablet servers and balances the load among them, manages garbage collection, and handles table and column family creation and deletion.

Internally, the space management is ensured by a three-level hierarchy: the *root tablet*, the location of which is stored in a *Chubby* file, points to entries in the second element, the *metadata* tablet, which, in turn, points to *user* tablets, collections of locations of users' tablets. An application client searches through this hierarchy to identify the location of its tablets and then caches the addresses for further use.

The performance of the system reported in [73] is summarized in Table 8.2. The table shows the number of random and sequential `read` and `write` and scan operations for 1,000 bytes, when the number of servers increases from 1 to 50, then to 250, and finally to 500. Locking prevents the system from achieving a linear speed-up, but the performance of the system is still remarkable due to a fair number of optimizations. For example, the number of scans on 500 tablet servers is $7{,}843/2 \times 10^3$ instead of $15{,}385/2 \times 10^3$. It is reported that only 12 clusters use more than 500 tablet servers, whereas some 259 clusters use between 1 and 19 tablet servers.



clean and consolidate the data for the serving phase. The serving table stored on GFS is "only" 500 GB, and it is distributed across several hundred tablet servers, which maintain in-memory column families. This organization enables the serving phase of *Google Earth* to provide a fast response time to tens of thousands of queries per second.

Google Analytics provides aggregate statistics such as the number of visitors to a Web page per day. To use this service, Web servers embed a *JavaScript* code into their Web pages to record information every time a page is visited. The data is collected in a *raw-click BigTable* of some 200 TB, with a row for each end-user session. A *summary* table of some 20 TB contains predefined summaries for a Website.