21

DESIGNING DISTRIBUTED SYSTEMS: GOOGLE CASE STUDY

The ability to create an effective design is an important skill in distributed systems, requiring an awareness of the different technological choices featured throughout this book and a thorough understanding of the requirements of the relevant application domain. The eventual goal is to come up with a consistent distributed system architecture incorporating a consistent and complete set of design choices able to address the overall requirements. This is a demanding task and one that requires considerable experience with distributed systems development.

We illustrate distributed design through a substantial case study, examining in detail the design of the Google infrastructure, a platform and associated middleware that supports both Google search and a set of associated web services and applications including Google Apps. This includes the study of key underlying components including the physical infrastructure underpinning Google, the communication paradigms offered by the Google infrastructure and the associated core services for storage and computation.

Emphasis is placed on the key design principles behind the Google infrastructure and how they pervade the overall system architecture, resulting in a consistent and effective design.

21.1 Introduction

This book has focused on the key concepts that underpin the development of distributed systems, with an emphasis on addressing the main challenges of distributed systems, including heterogeneity, openness, security, scalability, failure handling, concurrency, transparency and quality of service. The subsequent treatment focuses inevitably on the constituent parts of a distributed system, including: communication paradigms such as remote invocation and its indirect alternatives; the programming abstractions offered by objects, components or web services; specific distributed systems services for security, naming and file system support; and algorithmic solutions such as coordination and agreement. It is equally important, however, to consider the overall design of distributed systems and how the constituent parts come together, addressing the inevitable trade-offs between the different challenges to derive an overall system architecture that meets the requirements of a large-scale application domain and operating environment. A fuller treatment of distributed systems design methods would necessarily take us into the field of software engineering methodologies for distributed systems. That is beyond the scope of this book; those interested should see the box below for some relevant topics and sources. Instead, we elect to provide insight into this area by presenting a case study of a complex distributed system, highlighting the key decisions and trade-offs involved in this design.

To motivate the studies in the book, Chapter 1 outlined three examples of key application domains that represent many of the major challenges in distributed systems: web search, massively multiplayer online games and financial trading. We could have picked any one of these areas and presented an enticing and illuminating case study, but we have chosen to focus on the first: web search (and indeed, we look beyond web search to more general support for web-based cloud services). In particular, in this chapter we present a case study on the distributed systems infrastructure that underpins Google (hereafter referred to as the Google infrastructure). Google is one of the largest distributed systems in use today, and the Google infrastructure has successfully dealt with a variety of demanding requirements, as discussed below. The underlying architecture and choice of concepts are also very interesting, picking up on many of the core topics presented in this book. A study of the Google infrastructure therefore

Software engineering for distributed systems

We refer the reader to the significant advances that have been made in areas such as:

- object-oriented design, including the use modelling notations such as UML [Booch *et al*. 2005];
- component-based software engineering (CBSE) and its relationship to enterprise architectures [Szyperski 2002];
- architectural patterns targeting distributed systems [Bushmann *et al*. 2007];
- model-driven engineering that seeks to generate complex systems (including distributed systems from higher-level abstractions (models) [France and Rumpe 2007].

provides a perfect way to round off our study of distributed systems. Note that, as well as providing an example of how to support web search, the Google infrastructure has emerged as a leading example of cloud computing, as will become apparent in the descriptions that follow.

Section 21.2 introduces the case study, providing background information on Google. Section 21.3 then presents the overall design of the Google infrastructure, considering both the underlying physical model and the associated system architecture. Section 21.4 examines the lowest level of the system architecture, the communication paradigms supported by the Google infrastructure; the subsequent two sections, Sections 21.5 and 21.6, discuss the core services provided by the Google infrastructure, featuring the services for storage and processing of massive quantities of data. Sections 21.3 to 21.6 together describe a complete middleware solution for web search and cloud computing. Finally, Section 21.7 summarizes the key points emerging from our discussion of the Google infrastructure. Throughout the presentation, an emphasis will be placed on identifying and justifying the core design decisions and the associated trade-offs that are inherent in the design.

21.2 Introducing the case study: Google

Google [www.google.com III] is a US-based corporation with its headquarters in Mountain View, California (the Googleplex), offering Internet search and broader web applications and earning revenue largely from advertising associated with such services. The name is a play on the word *googol*, the number 10^{100} (or 1 followed by a hundred zeros), emphasizing the sheer scale of information available in the Internet today. Google's mission is to tame this huge body of information: 'to organize the world's information and make it universally accessible and useful' [www.google.com III].

Google was born out of a research project at Stanford University, with the company launched in 1998. Since then, it has grown to have a dominant share of the Internet search market, largely due to the effectiveness of the underlying ranking algorithm used in its search engine (discussed further below). Significantly, Google has diversified, and as well as providing a search engine is now a major player in cloud computing.

From a distributed systems perspective, Google provides a fascinating case study with extremely demanding requirements, particularly in terms of scalability, reliability, performance and openness (see the discussion of these challenges in Section 1.5). For example, in terms of search, it is noteworthy that the underlying system has successfully scaled with the growth of the company from its initial production system in 1998 to dealing with over 88 billion queries a month by the end of 2010, that the main search engine has never experienced an outage in all that time and that users can expect query results in around 0.2 seconds [googleblog.blogspot.com I]. The case study we present here will examine the strategies and design decisions behind that success, and provide insight into design of complex distributed systems.

Before proceeding to the case study, though, it is instructive to look in more detail at the search engine and also at Google as a cloud provider.

The Google search engine • The role of the Google search engine is, as for any web search engine, to take a given query and return an ordered list of the most relevant results that match that query by searching the content of the Web. The challenges stem from the size of the Web and its rate of change, as well as the requirement to provide the most relevant results from the perspective of its users.

We provide a brief overview of the operation of Google search below; a fuller description of the operation of the Google search engine can be found in Langville and Meyer [2006]. As a running example, we consider how the search engine responds to the query 'distributed systems book'.

The underlying search engine consists of a set of services for crawling the Web and indexing and ranking the discovered pages, as discussed below.

Crawling: The task of the crawler is to locate and retrieve the contents of the Web and pass the contents onto the indexing subsystem. This is performed by a software service called Googlebot, which recursively reads a given web page, harvesting all the links from that web page and then scheduling further crawling operations for the harvested links (a technique known as *deep searching* that is highly effective in reaching practically all pages in the Web).

In the past, because of the size of the Web, crawling was generally performed once every few weeks. However, for certain web pages this was insufficient. For example, it is important for search engines to be able to report accurately on breaking news or changing share prices. Googlebot therefore took note of the change history of web pages and revisited frequently changing pages with a period roughly proportional to how often the pages change. With the introduction of Caffeine in 2010 [googleblog.blogspot.com II], Google has moved from a batch approach to a more continuous process of crawling intended to offer more freshness in terms of search results. Caffeine is built using a new infrastructure service called Percolator that supports the incremental updating of large datasets [Peng and Dabek 2010].

Indexing: While crawling is an important function in terms of being aware of the content of the Web, it does not really help us with our search for occurrences of 'distributed systems book'. To understand how this is processed, we need to have a closer look at indexing.

The role of indexing is to produce an index for the contents of the Web that is similar to an index at the back of a book, but on a much larger scale. More precisely, indexing produces what is known as an *inverted index* mapping words appearing in web pages and other textual web resources (including documents in *.pdf*, *.doc* and other formats) onto the positions where they occur in documents, including the precise position in the document and other relevant information such as the font size and capitalization (which is used to determine importance, as will be seen below). The index is also sorted to support efficient queries for words against locations.

As well as maintaining an index of words, the Google search engine also maintains an index of *links*, keeping track of which pages link to a given site. This is used by the PageRank algorithm, as discussed below.

Let us return to our example query. This inverted index will allow us to discover web pages that include the search terms 'distributed', 'systems' and 'book' and, by careful analysis, we will be able to discover pages that include all of these terms. For example, the search engine will be able to identify that the three terms can all be found in amazon.com, www.cdk5.net and indeed many other web sites. Using the index, it is therefore possible to narrow down the set of candidate web pages from billions to perhaps tens of thousands, depending on the level of discrimination in the keywords chosen.

Ranking: The problem with indexing on its own is that it provides no information about the relative importance of the web pages containing a particular set of keywords – yet this is crucial in determining the potential relevance of a given page. All modern search engines therefore place significant emphasis on a system of ranking whereby a higher rank is an indication of the importance of a page and it is used to ensure that important pages are returned nearer to the top of the list of results than lower-ranked pages. As mentioned above, much of the success of Google can be traced back to the effectiveness of its ranking algorithm, *PageRank* [Langville and Meyer 2006].

PageRank is inspired by the system of ranking academic papers based on citation analysis. In the academic world, a paper is viewed as important if it has a lot of citations by other academics in the field. Similarly, in PageRank, a page will be viewed as important if it is linked to by a large number of other pages (using the link data mentioned above). PageRank also goes beyond simple 'citation' analysis by looking at the importance of the sites that contain links to a given page. For example, a link from *bbc.co.uk* will be viewed as more important than a link from Gordon Blair's personal web page.

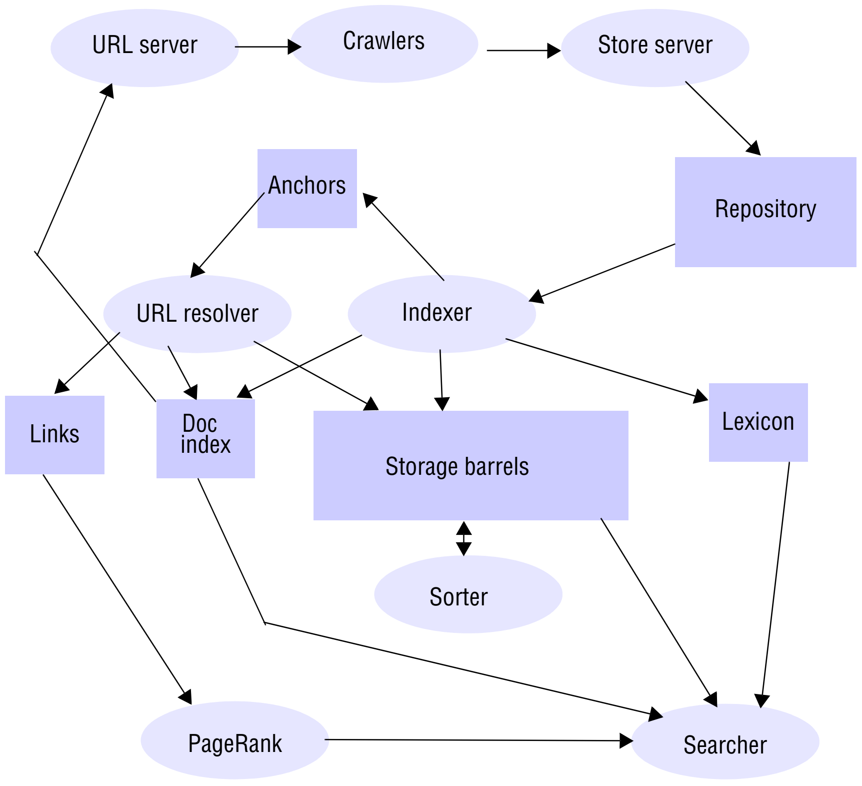
Ranking in Google also takes a number of other factors into account, including the proximity of keywords on a page and whether they are in a large font or are capitalized (based on the information stored in the inverted index).

Returning to our example, after performing an index lookup for each of the three words in the query, the search function ranks all the resulting page references according to perceived importance. For example, the ranking will pick out certain page references under amazon.com and www.cdk5.net because of the large number of links to those pages from other 'important' sites. The ranking will also prioritize pages where the terms 'distributed', 'systems' and 'book' appear in close proximity. Similarly, the ranking should pull out pages where the words appear near the start of the page or in capitals, perhaps indicating a list of distributed systems textbooks. The end result should be a ranked list of results where the entries at the top are the most important results.

Anatomy of a search engine: The founders of Google, Sergey Brin and Larry Page, wrote a seminal paper on the 'anatomy' of the Google search engine in 1998 [Brin and Page 1998], providing interesting insights into how their search engine was implemented. The overall architecture described in this paper is illustrated in Figure 21.1, redrawn from the original. In this diagram, we distinguish between services directly supporting web search, drawn as ovals, and the underlying storage infrastructure components, illustrated as rectangles.

While it is not the purpose of this chapter to present this architecture in detail, a brief overview will aid comparison with the more sophisticated Google infrastructure available today. The core function of *crawling* was described above. This takes as input lists of URLs to be fetched, provided by the *URL server*, with the resultant fetched pages placed into the *store server*. This data is then compressed and placed in the *repository* for further analysis, in particular creating the index for searching. The indexing function is performed in two stages. Firstly, the *indexer* uncompresses the data in the repository

Figure 21.1 Outline architecture of the original Google search engine [Brin and Page 1998]



and produces a set of *hits*, where a hit is represented by the document ID, the word, the position in the document and other information such as word size and capitalization. This data is then stored in a set of *barrels*, a key storage element in the initial architecture. This information is sorted by the document ID. The *sorter* then takes this data and sorts it by word ID to produce the necessary inverted index (as discussed above). The indexer also performs two other crucial functions as it parses the data: it extracts information about links in documents storing this information in an *anchors* file, and it produces a *lexicon* for the analyzed data (which at the time the initial architecture was used,consisted of 14 million words). The anchors file is processed by a *URL resolver*, which performs a number of functions on this data including resolving relative URLs into absolute URLs before producing a *links* database, as an important input into *PageRank* calculations. The URL resolver also create a *doc index*, which provides input to the URL server in terms of further pages to crawl. Finally, the *searcher* implements the core Google search capability, taking input from the doc index, PageRank, the inverted index held in the barrels and also the lexicon.

One thing that is striking about this architecture is that, while specific details of the architecture have changed, the key services supporting web search – that is, crawling, indexing (including sorting) and ranking (through PageRank) – remain the same.

Figure 21.2 Example Google applications

Application	*Description*
Gmail	Mail system with messages hosted by Google but desktop-like message management.
Google Docs	Web-based office suite supporting shared editing of documents held on Google servers.
Google Sites	Wiki-like web sites with shared editing facilities.
Google Talk	Supports instant text messaging and Voice over IP.
Google Calendar	Web-based calendar with all data hosted on Google servers.
Google Wave	Collaboration tool integrating email, instant messaging, wikis and social networks.
Google News	Fully automated news aggregator site.
Google Maps	Scalable web-based world map including high-resolution imagery and unlimited user-generated overlays.
Google Earth	Scalable near-3D view of the globe with unlimited user-generated overlays.
Google App Engine	Google distributed infrastructure made available to outside parties as a service (platform as a service).

Equally striking is that, as will become apparent below, the infrastructure has changed dramatically from the early attempts to identify an architecture for web search to the sophisticated distributed systems support provided today, both in terms of identifying more reusable building blocks for communication, storage and processing and in terms of generalizing the architecture beyond search.

Google as a cloud provider • Google has diversified significantly beyond search and now offers a wide range of web-based applications, including the set of applications promoted as Google Apps [www.google.com I]. More generally, Google is now a major player in the area of cloud computing. Recall that cloud computing was introduced in Chapter 1 and defined as 'a set of Internet-based application, storage and computing services sufficient to support most users' needs, thus enabling them to largely or totally dispense with local data storage and application software'. This is exactly what Google now strives to offer, in particular with significant offerings in the *software as a service* and *platform as a service* areas (as introduced in Section 7.7.1). We look at each area in turn below.

Software as a service: This area is concerned with offering application-level software over the Internet as web applications. A prime example is Google Apps, a set of web-based applications including Gmail, Google Docs, Google Sites, Google Talk and Google Calendar. Google's aim is to replace traditional office suites with applications supporting shared document preparation, online calendars, and a range of collaboration tools supporting email, wikis, Voice over IP and instant messaging.

Several other innovative web-based applications have recently been developed; these and the original Google Apps are summarized in Figure 21.2. One of the key observations for the purposes of this chapter is that Google encourages an open approach to innovation within the organization, and hence new applications are emerging all the time. This places particular demands on the underlying distributed systems infrastructure, a point that is revisited in Section 21.3.2.

Platform as a service: This area is concerned with offering distributed system APIs as services across the Internet, with these APIs used to support the development and hosting of web applications (note that the use of the term ‘platform’ in this context is unfortunately inconsistent with the way it is used elsewhere in this book, where it refers to the hardware and operating system level). With the launch of the Google App Engine, Google went beyond software as a service and now offers its distributed systems infrastructure as discussed throughout this chapter as a cloud service. More specifically, the Google business is already predicated on using this cloud infrastructure internally to support all its applications and services, including its web search engine. The Google App Engine now provides external access to a part of this infrastructure, allowing other organizations to run their own web applications on the Google platform.

We will see further details of the Google infrastructure as this chapter unfolds; refer to the Google web site for further details of the Google App Engine [code.google.com IV].

21.3 Overall architecture and design philosophy

This section looks at the overall architecture of the Google system, examining:

- the physical architecture adopted by Google;
- the associated system architecture that offers common services to the Internet search engine and the many web applications offered by Google.

21.3.1 Physical model

The key philosophy of Google in terms of physical infrastructure is to use very large numbers of commodity PCs to produce a cost-effective environment for distributed storage and computation. Purchasing decisions are based on obtaining the best performance per dollar rather than absolute performance with a typical spend on a single PC unit of around $1,000. A given PC will typically have around 2 terabytes of disk storage and around 16 gigabytes of DRAM (dynamic random access memory) and run a cut-down version of the Linux kernel. (This philosophy of building systems from commodity PCs reflects the early days of the original research project, when Sergey Brin and Larry Page built the first Google search engine from spare hardware scavenged from around the lab at Stanford University.)

In electing to go down the route of commodity PCs, Google has recognized that parts of its infrastructure will fail and hence, as we will see below, has designed the infrastructure using a range of strategies to tolerate such failures. Hennessy and Patterson [2006] report the following failure characteristics for Google:

- By far the most common source of failure is due to software, with about 20 machines needing to be rebooted per day due to software failures. (Interestingly, the rebooting process is entirely manual.)

- Hardware failures represent about 1/10 of the failures due to software with around 2–3% of PCs failing per annum due to hardware faults. Of these, 95% are due to faults in disks or DRAM.

This vindicates the decision to procure commodity PCs; given that the vast majority of failures are due to software, it is not worthwhile to invest in more expensive, more reliable hardware. A further paper by Pinheiro *et al.* [2007] also reports on the failure characteristics of commodity disks as used in the Google physical infrastructure, providing an interesting insight into failure patterns of disk storage in large-scale deployments.

The physical architecture is constructed as follows [Hennessy and Patterson 2006]:

- Commodity PCs are organized in racks with between 40 and 80 PCs in a given rack. The racks are double-sided with half the PCs on each side. Each rack has an Ethernet switch that provides connectivity across the rack and also to the external world (see below). This switch is modular, organized as a number of blades with each blade supporting either 8 100-Mbps network interfaces or a single 1-Gbps interface. For 40 PCs, five blades each containing eight network interfaces are sufficient to ensure connectivity within the rack. Two further blades, each supporting a 1-Gbps network interface, are used for connection to the outside world.

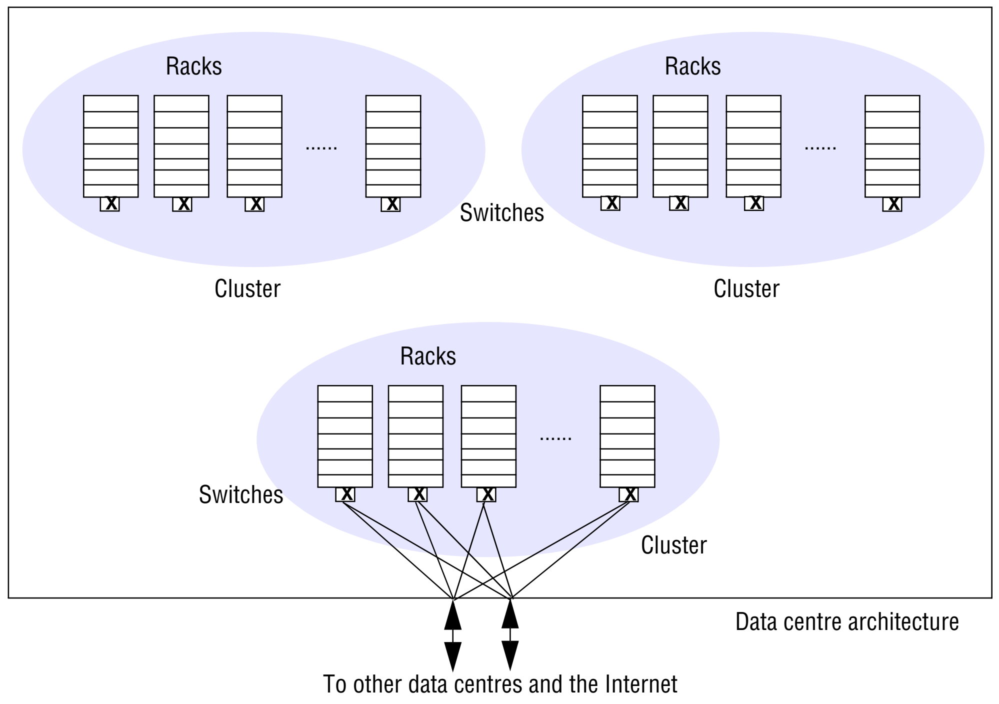
- Racks are organized into clusters (as discussed in Section 1.3.4), which are a key unit of management, determining for example the placement and replication of services. A cluster typically consists of 30 or more racks and two high-bandwidth switches providing connectivity to the outside world (the Internet and other Google centres). Each rack is connected to both switches for redundancy; in addition, for further redundancy, each switch has redundant links to the outside world.

- Clusters are housed in Google data centres that are spread around the world. In 2000, Google relied on key data centres in Silicon Valley (two centres) and in Virginia. At the time of writing, the number of data centres has grown significantly and there are now centres in many geographical locations across the US and in Dublin (Ireland), Saint-Ghislain (Belgium), Zurich (Switzerland), Tokyo (Japan) and Beijing (China). (A map of known data centres as of 2008 can be found here [royal.pingdom.com].)

A simplified view of this overall organization is provided in Figure 21.3. This physical infrastructure provides Google with enormous storage and computational capabilities, together with the necessary redundancy to build fault-tolerant, large-scale systems (note that, to avoid clutter, this figure only shows the Ethernet connections from one of the clusters to the external links).

Storage capacity: Let us consider the storage capacity available to Google. If each PC offers 2 terabytes of storage, then a rack of 80 PCs will provide 160 terabytes, with a cluster of 30 racks offering 4.8 petabytes. It is not known exactly how many machines Google has in total as the company maintains strict secrecy over this aspect of its business, but we can assume Google has on the order of 200 clusters, offering

Figure 21.3 Organization of the Google physical infrastructure



(To avoid clutter the Ethernet connections are shown from only one of the clusters to the external links)

total storage capacity of 960 petabytes or just under 1 exabyte of storage (10^{18} bytes). This is likely to be a conservative figure, as Google VP Marissa Mayer is already talking about the data explosion taking us well into the exascale range [www.parc.com].

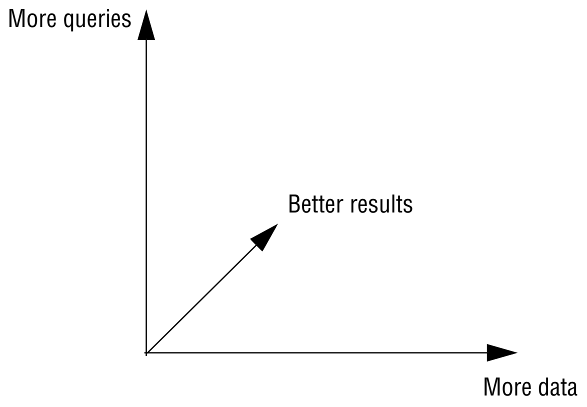
We shall see how Google uses this extensive storage and computational capability and the associated redundancy to offer core services in the remainder of this chapter.

21.3.2 Overall system architecture

Before examining the overall system architecture, it is helpful to examine the key requirements in more detail:

Scalability: The first and most obvious requirement for the underlying Google infrastructure is to master scalability and, in particular, to have approaches that scale to what is an Ultra-Large Scale (ULS) distributed system as featured in Chapter 2. For the search engine, Google views the scalability problem in terms of three dimensions: i) being able to deal with more data (for example, as the amount of information in the Web grows through initiatives such as the digitizing of libraries), ii) being able to deal with more queries (as the number of people using Google in their

Figure 21.4 The scalability problem in Google

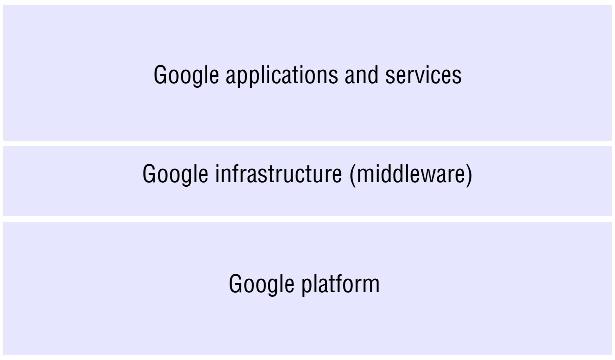


homes and workplaces grows) and iii) seeking better results (particularly important as this is a key determining factor in uptake of a web search engine). This view of the scalability problem is illustrated in Figure 21.4.

Scalability demands the use of (sophisticated) distributed systems strategies. Let us illustrate this with a simple analysis taken from Jeff Dean's keynote at PACT'06 [Dean 2006]. He assumed that the Web consists of around 20 billion web pages at 20 kilobytes each. This implies a total size of around 400 terabytes. To crawl this amount of data it would take a single computer over 4 months assuming that the computer can read 30 megabytes per second. In contrast, 1,000 machines can read this amount of data in less than 3 hours. In addition, as we saw in Section 21.2, searching is not only about crawling. The other functions, including indexing, ranking and searching, all require highly distributed solutions in order to be able to scale.

Reliability: Google has stringent reliability requirements, especially with regard to availability of services. This is particularly important for the search functionality, where there is a need to provide 24/7 availability (noting, however that it is intrinsically easy to mask failures in search as the user has no way of knowing if all search results are returned). This requirement also applies to other web applications, and it is interesting to note that Google offers a 99.9% service level agreement (effectively, a system guarantee) to paying customers of Google Apps covering Gmail, Google Calendar, Google Docs, Google Sites and Google Talk. The company has an excellent overall record in terms of availability of services, but the well-reported outage of Gmail on 1 September 2009 acts as a reminder of the continuing challenges in this area. (This outage, which lasted 100 minutes, was caused by a cascading problem of overloading servers during a period of routine maintenance). Note that the reliability requirement must be met in the context of the design choices in the physical architecture, which imply that (software and hardware) failures are anticipated with reasonable frequency. This demands both detecting failures and adopting strategies to mask or tolerate such failures. Such strategies rely heavily on the redundancy in the underlying physical architecture. We shall see examples of such strategies as the details of the system architecture emerge.

Figure 21.5 The overall Google systems architecture

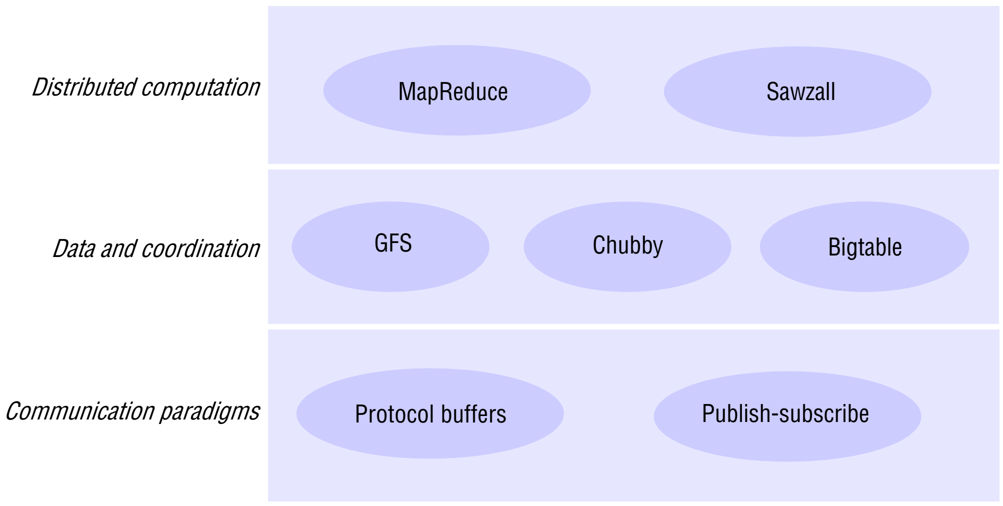

Google applications and services
Google infrastructure (middleware)
Google platform

Performance: The overall performance of the system is critical for Google, especially in achieving low latency of user interactions. The better the performance, the more likely it is that a user will return with more queries that, in turn, increase their exposure to ads hence potentially increasing revenue. The importance of performance is exemplified by the target of completing web search operations in 0.2 seconds (as mentioned above) and achieving the required throughput to respond to all incoming requests while dealing with very large datasets. This applies to a wide range of functions associated with the operation of Google, including web crawling, indexing and sorting. It is also important to note that performance is an end-to-end property requiring all associated underlying resources to work together, including network, storage and computational resources.

Openness: The above requirements are in many ways the obvious ones for Google to support its core services and applications. There is also a strong requirement for openness, particularly to support further development in the range of web applications on offer. It is well known that Google as an organization encourages and nurtures innovation, and this is most evident in the development of new web applications. This is only possible with an infrastructure that is extensible and provides support for the development of new applications.

Google has responded to these needs by developing the overall system architecture shown in Figure 21.5. This figure shows the underlying computing platform at the bottom (that is, the physical architecture as described above) and the well-known Google services and applications at the top. The middle layer defines a common distributed infrastructure providing middleware support for search and cloud computing. This is crucial to the success of Google. The infrastructure provides the common distributed system services for developers of Google services and applications and encapsulates key strategies for dealing with scalability, reliability and performance. The provision of a well-designed common infrastructure such as this can bootstrap the development of new applications and services through reuse of the underlying system services and, more subtly, provides an overall coherence to the growing Google code base by enforcing common strategies and design principles.

Figure 21.6 Google infrastructure



Google infrastructure • The system is constructed as a set of distributed services offering core functionality to developers (see Figure 21.6). This set of services naturally partitions into the following subsets:

- the underlying communication paradigms, including services for both remote invocation and indirect communication:
  - the *protocol buffers* component offers a common serialization format for Google, including the serialization of requests and replies in remote invocation.
  - the Google *publish-subscribe* service supports the efficient dissemination of events to potentially large numbers of subscribers.
- data and coordination services providing unstructured and semi-structured abstractions for the storage of data coupled with services to support coordinated access to the data:
  - GFS offers a distributed file system optimized for the particular requirements of Google applications and services (including the storage of very large files).
  - Chubby supports coordination services and the ability to store small volumes of data.
  - Bigtable provides a distributed database offering access to semi-structured data.
- distributed computation services providing means for carrying out parallel and distributed computation over the physical infrastructure:
  - MapReduce supports distributed computation over potentially very large datasets (for example, stored in Bigtable).

  - Sawzall provides a higher-level language for the execution of such distributed computations.

We step through each of these components in turn in Sections 21.4 through 21.6. First, however, it is instructive to reflect on key design principles associated with the architecture as a whole.

Associated design principles • To fully understand the design of Google infrastructure, it is important to also have an understanding of key design philosophies that pervade the organization:

- The most important design principle behind Google software is that of *simplicity*: software should do one thing and do it well, avoiding feature-rich designs wherever possible. For example, Bloch [2006] discusses how this principle applies to API design implying that API designs should be as small as possible and no smaller (an example of the application of Occam's Razor).
- Another key design principle is a strong emphasis on *performance* in the development of systems software, captured in the phrase 'every millisecond counts' [www.google.com IV]. In a keynote at LADIS'09, Jeff Dean (a member of the Google Systems Infrastructure Group) emphasized the importance of being able to estimate the performance of a system design through awareness of performance costs of primitive operations such as accessing memory and disk, sending packets over a network, locking and unlocking a mutex and so on, coupled with what he referred to as 'back of the envelope' calculations [www.cs.cornell.edu].
- A final principle is advocating stringent *testing* regimes on software, captured by the slogan 'if it ain't broke, you are not trying hard enough' [googletesting.blogspot.com]. This is complemented by a strong emphasis on *logging* and *tracing* to detect and resolve faults in the system.

With this background, we are now ready to examine the various constituent parts of the Google infrastructure, starting with the underlying communication paradigms. For each area, we present the overall design and highlight the key design decisions and associated trade-offs.

21.4 Underlying communication paradigms

Looking back at Chapters 3 to 6, it is clear that the choice of underlying communication paradigm(s) is crucial to the success of an overall system design. Options include:

- using an underlying interprocess communication service directly, such as that offered by socket abstractions (described in Chapter 4 and supported by all modern operating systems);
- using a remote invocation service (such as a request-reply protocol, remote procedure calls or remote method invocation, as discussed in Chapter 5) offering support for client-server interactions;

- using an indirect communication paradigm such as group communication, distributed event-based approaches, tuple spaces or distributed shared memory approaches (as discussed in Chapter 6).

In keeping with the design principles identified in Section 21.3, Google adopts a simple, minimal and efficient remote invocation service that is a variant of a remote procedure call approach.

Readers will recall that remote procedure call communication requires a serialization component to convert the procedure invocation data (procedure name and parameters, possibly structured) from their internal binary representation to a *flattened* or *serialized* processor-neutral format ready for transmission to the remote partner. Serialization for Java RPC is described in Section 4.3.2. XML has emerged more recently as a ‘universal’ serialized data format, but its generality brings substantial overheads. Google has therefore developed a simplified and high-performance serialization component known as *protocol buffers* that is used for a substantial majority of interactions within the infrastructure. This can be used over any underlying communication mechanism to provide an RPC capability. An open source version of protocol buffers is available [code.google.com I].

A separate *publish-subscribe service* is also used, recognizing the key role that this paradigm can offer in many areas of distributed system design, including the efficient and real-time dissemination of events to multiple participants. In common with many other distributed system platforms, the Google infrastructure therefore offers a hybrid solution allowing developers to select the best communication paradigm for their requirements. Publish-susbcribe is not an alternative to protocol buffers in the Google infrastructure, but rather an augmentation offering an added-value service for where it is most appropriate.

We examine the design of these two approaches below, with emphasis on protocol buffers (full details of the publish-subscribe protocol are not yet publicly available).

21.4.1 Remote invocation

Protocol buffers place emphasis on the description and subsequent serialization of data, and hence the concept is best compared to direct alternatives such as XML. The goal is to provide a language- and platform-neutral way to specify and serialize data in a manner that is simple, highly efficient and extensible; the serialized data can then be used for subsequent storage of data or transmission using an underlying communications protocol, or indeed for any other purpose that demands a serialization format for structured data. We will see later how this can be used as the basis for RPC-style exchange.

In protocol buffers, a language is provide for the specification of *messages*. We introduce the key features of this (simple) language by example, with Figure 21.7 showing how a book message might be specified.

As can be seen, the overall *Book* message consists of a series of uniquely numbered fields, each represented by a field name and the type of the associated value. The type can be one of:

- a *primitive data type* (including integer, floating-point, boolean, string or raw bytes);

Figure 21.7 Protocol buffers example

```
message Book {
    required string title = 1;
    repeated string author = 2;
    enum Status {
        IN_PRESS = 0;
        PUBLISHED = 1;
        OUT_OF_PRINT = 2;
    }
    message BookStats {
        required int32 sales = 1;
        optional int32 citations = 2;
        optional Status bookstatus = 3 [default = PUBLISHED];
    }
    optional BookStats statistics = 3;
    repeated string keyword = 4;
}
```

- an *enumerated type*;
- a *nested message* allowing a hierarchical structuring of data.

We can see examples of each in Figure 21.7.

Fields are annotated with one of three labels:

- *required* fields must be present in the message;
- *optional* fields may be present in the message;
- *repeated* fields can exist zero or more times in the message (the developers of protocol buffers view this as a type of dynamically sized array).

Again, we can see uses of each annotation in the *Book* message format illustrated in Figure 21.7.

The unique number (=1, =2 and so on) represents the tag that a particular field has in the binary encoding of the message.

This specification is contained in a *.proto* file and compiled by a *protoc* tool. The output of this tool is generated code that allows programmers to manipulate the particular message type, in particular assigning/extracting values to/from messages. In more detail, the *protoc* tool generates a *builder* class that provides *getter* and *setter* methods for each field together with additional methods to *test* if a method has been set and to *clear* a field to the associated null value. For example, the following methods would be generated for the *title* field:

```
public boolean hasTitle();
public java.lang.String getTitle();
public Builder setTitle(String value);
public Builder clearTitle();
```

The importance of the builder class is that while messages are immutable in protocol buffers, builders are mutable and are used to construct and manipulate new messages.

For repeated fields the generated code is slightly more complicated, with methods provided to return a *count* of the number of elements in the associated list, to *get* or *set* specific fields in the list, to *append* a new element to a list and to add a set of elements to a list (the *addAll* method). We illustrate this by example by listing the methods generated for the *keyword* field:

```
public List<string> getKeywordList();
public int getKeywordCount();
public string getKeyword(int index);
public Builder setKeyword(int index, string value);
public Builder addKeyword(string value);
public Builder addAllKeyword(Iterable<string> value);
public Builder clearKeyword();
```

The generated code also provides a range of other methods to manipulate messages, including methods such as *toString* to provide a readable representation of the message (often used for debugging for example) and also a series of methods to parse incoming messages.

As can be seen, this is a very simple format compared to XML (for example, contrast the specification above with equivalent specifications in XML as shown in Section 4.3.3), and one that its developers claim is 3 to 10 times smaller than XML equivalents and 10 to 100 times faster in operation. The associated programming interface providing access to the data is also considerably simpler than equivalents for XML.

Note that this is a somewhat unfair comparison, for two reasons. Firstly, the Google infrastructure is a relatively closed system and hence, unlike, XML, it does not address interoperability across open systems. Secondly, XML is significantly richer in that it generates self-describing messages that contain the data and associated metadata describing the structure of the messages (see Section 4.3.3). Protocol buffers do not provide this facility directly (although it is possible to achieve this effect, as described in the relevant web pages in a section on techniques [code.google.com II]). In outline, this is achieved by asking the *protoc* compiler to generate a *FileDescriptorSet* that contains self-descriptions of messages, and then including this explicitly in message descriptions. The developers of protocol buffers, though, emphasize that this is not seen as a particularly useful feature and that it is rarely used in the Google infrastructure code.

Supporting RPC • As mentioned above, protocol buffers are a general mechanism that can be used for storage or communication. The most common use of protocol buffers, however, is to specify RPC exchanges across the network, and this is accommodated with extra syntax in the language. Again, we illustrate the syntax by example:

```
service SearchService {
    rpc Search (RequestType) returns (ResponseType);
}
```

This code fragment specifies a service interface called *SearchService* containing one remote operation, *Search*, which takes one parameter of type *RequestType* and returns one parameter of type *ResponseType*. For example, the types could correspond to a list of keywords and a list of *Books* matching this set of keywords, respectively. The *protoc* compiler takes this specification and produces both an abstract interface *SearchService* and a stub that supports type-safe RPC-style calls to the remote service using protocol buffers.

As well as being language- and platform-neutral, protocol buffers are also agnostic with respect to the underlying RPC protocol. In particular, the stub assumes that implementations exist for two abstract interfaces *RpcChannel* and *RpcController,* the former offering a common interface to underlying RPC implementations and the latter offering a common control interface, for example, to manipulate settings associated with that implementation. A programmer must provide implementations of these abstract interfaces, effectively selecting the desired RPC implementation. For example, this could pass serialized messages using HTTP or TCP or could map onto one of a number of third-party RPC implementations available and linked from the protocol buffers site [code.google.com III].

Note that a service interface can support multiple remote operations, but each operation must adhere to the pattern of taking a single parameter and returning a single result (with both being protocol buffer messages). This is unusual compared to the designs of RPC and RMI systems – as we saw in Chapter 5 – remote invocations can have an arbitrary number of parameters, and in the case of RMI the parameters or results can be objects or indeed object references (although note that Sun RPC, as documented in Section 5.3.3, adopts a similar approach to protocol buffers). The rationale for having one request and one reply is to support extensibility and software evolution; whereas the more general styles of interface may change significantly over time, this more constrained style of interface is likely to remain more constant. This approach also pushes the complexity towards the data in a manner that is reminiscent of the REST philosophy, with its constrained set of operations and emphasis on manipulating resources (see Section 9.2).

21.4.2 Publish-subscribe

Protocol buffers are used extensively but not exclusively as the communication paradigm in the Google infrastructure. To complement protocol buffers, the infrastructure also supports a publish-subscribe system intended to be used where distributed events need to be disseminated in real time and with reliability guarantees to potentially large numbers of recipients. As mentioned above, the publish-subscribe service is an augmentation to protocol buffers and indeed uses protocol buffers for its underlying communication.

One key use for the publish-subscribe system, for example, is to underpin the Google Ads system, recognizing that advertisments in Google are world-wide and that advertisement serving systems anywhere in the network need to know in a fraction of a second the eligibility of certain advertisements that can be shown in response to a query.

The RPC system described above would clearly be inappropriate and highly inefficient for this style of interaction, especially given the potentially large numbers of subscribers and the guarantees required by the associated applications. In particular, the sender would need to know the identity of all the other advertisment serving systems, which could be very large. RPCs would need to be sent to all the individual serving systems, consuming many connections and a great deal of associated buffer space at the sender, not to mention the bandwidth requirements of sending the data across wide area network links. A publish-subsribe solution, in contrast, with its inherent time and space uncoupling, overcomes these difficulties and also offers natural support for failure and recovery of subscribers (see Section 6.1).

Google has not made details of the publish-subscribe system publicly available. We therefore restrict our discussion to some high-level features of the system.

Google adopts a *topic-based* publish-subscribe system, providing a number of *channels* for event streams with channels corresponding to particular topics. A topic-based system was chosen for its ease of implementation and its relative predictability in terms of performance compared to content-based approaches – that is, the infrastructure can be set up and tailored to deliver events related to a given topic. The downside is a lack of expressive power in specifying events of interest. As a compromise, the Google publish-subscribe system allows enhanced subscriptions defined not just by selecting a channel but also by selecting subsets of events from within that channel. In particular, a given event consists of a header, an associated set of keywords and a payload, which is opaque to the programmer. Subscription requests specify the channel together with a filter defined over the set of keywords. Channels are intended to be used for relatively static and coarse-grained data streams requiring high throughputs of events (at least 1-Mbps), so the added capability for expressing refined subscriptions using filters helps greatly. For example, if a topic generates less than this volume of data, it will be subsumed within another topic but identifiable by keyword.

The publish-subscribe system is implemented as a *broker overlay* in the form of a set of trees, where each tree represents a topic. The root of the tree is the publisher and the leaf nodes represent subscribers. When filters are introduced, they are pushed as far back in the tree as possible to minimize unnecessary traffic.

Unlike the publish-subscribe systems discussed in Chapter 6, there is a strong emphasis on both reliable and timely delivery:

- In terms of reliability, the system maintains redundant trees; in particular, two separate tree overlays are maintained per logical channel (topic).
- In terms of timely delivery, the system implements a quality of service management technique to control message flows. In particular, a simple rate control scheme is introduced based on an imposed rate limit enforced on a per user/ per topic basis. This replaces a more complex approach and manages the anticipated resource usage across the tree in terms of memory, CPU and message and byte rates.

Trees are initially constructed and constantly re-evaluated according to a shortest path algorithm (see Chapter 3).

21.4.3 Summary of key design choices for communication

The overall design choices relating to communication paradigms in the Google infrastructure are summarized in Figure 21.8. This table highlights the most important decisions associated with the overall design and the constituent elements (protocol buffers and the publish-subscribe system) and summarizes the rationale and the particular trade-offs associated with each choice.

Figure 21.8 Summary of design choices related to communication paradigms

Element	*Design choice*	*Rationale*	*Trade-offs*
Protocol buffers	The use of a language for specifying data formats	Flexible in that the same language can be used for serializing data for storage or communication	-
	Simplicity of the language	Efficient implementation	Lack of expressiveness when compared, for example, with XML
	Support for a style of RPC (taking a single message as a parameter and returning a single message as result)	More efficient, extensible and supports service evolution	Lack of expressiveness when compared with other RPC or RMI packages
	Protocol-agnostic design	Different RPC implementation s can be used	No common semantics for RPC exchanges
Publish-subscribe	Topic-based approach	Supports efficient implementation	Less expressive than content-based approaches (mitigated by the additional filtering capabilities)
	Real-time and reliability guarantees	Supports maintenance of consistent views in a timely manner	Additional algorithmic support required with associated overhead

Overall, we see a hybrid approach offering two distinct communication paradigms designed to support different styles of interaction within the architecture. This allows developers to select the best paradigm for each particular problem domain.

We shall repeat this style of analysis at the end of each of the following sections, thus providing an overall perspective of key design decisions relating to the Google infrastructure.

21.5 Data storage and coordination services

We now present the three services that together provide data and coordination services to higher-level applications and services: the Google File System, Chubby and Bigtable. These are complementary services in the Google infrastructure:

- The Google File System is a distributed file system offering a similar service to NFS and AFS, as discussed in Chapter 12. It offers access to unstructured data in the form of files, but optimized to the styles of data and data access required by Google (very large files, for example).
- Chubby is a multi-faceted service supporting, for example, coarse-grained distributed locking for coordination in the distributed environment and the storage of very small quantities of data, complementing the large-scale storage offered by the Google File System.
- Bigtable offers access to more structured data in the form of tables that can be indexed in various ways including by row or column. Bigtable is therefore a style of distributed database, but unlike many databases it does not support full relational operators (these are viewed by Google as unnecessarily complex and unscalable).

These three services are also interdependent. For example, Bigtable uses the Google File System for storage and Chubby for coordination.

We look at each service in detail below.

21.5.1 The Google File System (GFS)

Chapter 12 presented a detailed study of the topic of distributed file systems, analyzing their requirements and their overall architecture and examining two case studies in detail, namely NFS and AFS. These file systems are *general-purpose* distributed file systems offering file and directory abstractions to a wide variety of applications in and across organizations. The Google File System (GFS) is also a distributed file system; it offers similar abstractions but is specialized for the very particular requirements that Google has in terms of storage and access to very large quantities of data [Ghemawat *et al*. 2003]. These requirements led to very different design decisions from those made in NFS and AFS (and indeed other distributed file systems), as we will see below. We start our discussion of GFS by examining the particular requirements identified by Google.

GFS requirements • The overall goal of GFS is to meet the demanding and rapidly growing needs of Google's search engine and the range of other web applications offered by the company. From an understanding of this particular domain of operation, Google identified the following requirements for GFS (see Ghemawat *et al.* [2003]):

- The first requirement is that GFS must run reliably on the physical architecture discussed in Section 21.3.1 – that is a very large system built from commodity hardware. The designers of GFS started with the assumption that components will fail (not just hardware components but also software components) and that the design must be sufficiently tolerant of such failures to enable application-level services to continue their operation in the face of any likely combination of failure conditions.

- GFS is optimized for the patterns of usage within Google, both in terms of the types of files stored and the patterns of access to those files. The number of files stored in GFS is not huge in comparison with other systems, but the files tend to be massive. For example, Ghemawat *et al.* [2003] report the need for perhaps one million files averaging 100 megabytes in size, but with some files in the gigabyte range. The patterns of access are also atypical of file systems in general. Accesses are dominated by sequential reads through large files and sequential writes that append data to files, and GFS is very much tailored towards this style of access. Small, random reads and writes do occur (the latter very rarely) and are supported, but the system is not optimized for such cases. These file patterns are influenced, for example, by the storage of many web pages sequentially in single files that are scanned by a variety of data analysis programs. The level of concurrent access is also high in Google, with large numbers of concurrent appends being particularly prevalent, often accompanied by concurrent reads.
- GFS must meet all the requirements for the Google infrastructure as a whole; that is, it must scale (particularly in terms of volume of data and number of clients), it must be reliable in spite of the assumption about failures noted above, it must perform well and it must be open in that it should support the development of new web applications. In terms of performance and given the types of data file stored, the system is optimized for high and sustained throughput in reading data, and this is prioritized over latency. This is not to say that latency is unimportant, rather, that this particular component (GFS) needs to be optimized for high-performance reading and appending of large volumes of data for the correct operation of the system as a whole.

These requirements are markedly different from those for NFS and AFS (for example), which must store large numbers of often small files and where random reads and writes are commonplace. These distinctions lead to the very particular design decisions discussed below.

GFS interface • GFS provides a conventional file system interface offering a hierarchical namespace with individual files identified by pathnames. Although the file system does not provide full POSIX compatibility, many of the operations will be familiar to users of such file systems (see, for example, Figure 12.4):

create – create a new instance of a file;

delete – delete an instance of a file;

open – open a named file and return a handle;

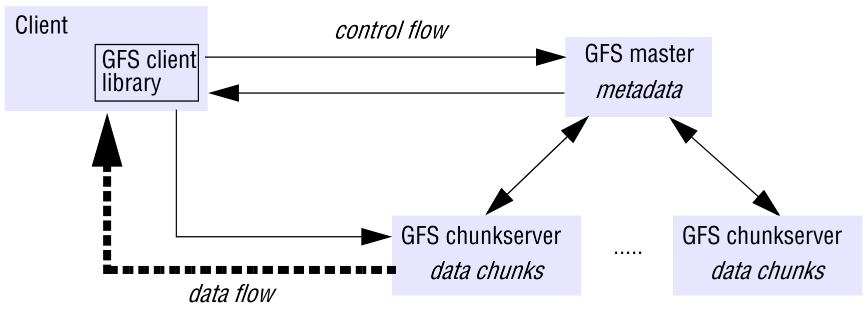
close – close a given file specified by a handle;

read – read data from a specified file;

write – write data to a specified file.

It can be seen that main GFS operations are very similar to those for the flat file service described in Chapter 12 (see Figure 12.6). We should assume that the GFS *read* and

Figure 21.9 Overall architecture of GFS



write operations take a parameter specifying a starting offset within the file, as is the case for the flat file service.

The API also offers two more specialized operations, *snapshot* and *record append*. The former operation provides an efficient mechanism to make a copy of a particular file or directory tree structure. The latter supports the common access pattern mentioned above whereby multiple clients carry out concurrent appends to a given file.

GFS architecture • The most influential design choice in GFS is the storage of files in fixed-size *chunks*, where each chunk is 64 megabytes in size. This is quite large compared to other file system designs. At one level this simply reflects the size of the files stored in GFS. At another level, this decision is crucial to providing highly efficient sequential reads and appends of large amounts of data. We return to this point below, once we have discussed more details of the GFS architecture.

Given this design choice, the job of GFS is to provide a mapping from files to chunks and then to support standard operations on files, mapping down to operations on individual chunks. This is achieved with the architecture shown in Figure 21.9, which shows an instance of a GFS file system as it maps onto a given physical cluster. Each GFS cluster has a single *master* and multiple *chunkservers* (typically on the order of hundreds), which together provide a file service to large numbers of clients concurrently accessing the data.

The role of the master is to manage metadata about the file system defining the namespace for files, access control information and the mapping of each particular file to the associated set of chunks. In addition, all chunks are replicated (by default on three independent chunkservers, but the level of replication can be specified by the programmer). The location of the replicas is maintained in the master. Replication is important in GFS to provide the necessary reliability in the event of (expected) hardware and software failures. This is in contrast to NFS and AFS, which do not provide replication with updates (see Chapter 12).

The key metadata is stored persistently in an operation log that supports recovery in the event of crashes (again enhancing reliability). In particular, all the information mentioned above is logged apart from the location of replicas (the latter is recovered by polling chunkservers and asking them what replicas they currently store).

Although the master is centralized, and hence a single point of failure, the operations log is replicated on several remote machines, so the master can be readily restored on failure. The benefit of having such a single, centralized master is that it has a *global view* of the file system and hence it can make optimum management decisions, for example related to chunk placement. This scheme is also simpler to implement, allowing Google to develop GFS in a relatively short period of time. McKusick and Quinlan [2010] present the rationale for this rather unusual design choice.

When clients need to access data starting from a particular byte offset within a file, the GFS client library will first translate this to a file name and chunk index pair (easily computed given the fixed size of chunks). This is then sent to the master in the form of an RPC request (using protocol buffers). The master replies with the appropriate chunk identifier and location of the replicas, and this information is cached in the client and used subsequently to access the data by direct RPC invocation to one of the replicated chunkservers. In this way, the master is involved at the start and is then completely out of the loop, implementing a separation of control and data flows – a separation that is crucial to maintaining high performance of file accesses. Combined with the large chunk size, this implies that, once a chunk has been identified and located, the 64 megabytes can then be read as fast as the file server and network will allow without any other interactions with the master until another chunk needs to be accessed. Hence interactions with the master are minimized and throughput optimized. The same argument applies to sequential appends.

Note that one further repercussion of the large chunk size is that GFS maintains proportionally less metadata (if a chunk size of 64 kilobytes was adopted, for example, the volume of metadata would increase by a factor of 1,000). This in turn implies that GFS masters can generally maintain all their metadata in main memory (but see below), thus significantly decreasing the latency for control operations.

As the system has grown in usage, problems have emerged with the centralized master scheme:

- Despite the separation of control and data flow and the performance optimization of the master, it is emerging as a bottleneck in the design.
- Despite the reduced amount of metadata stemming from the large chunk size, the amount of metadata stored by each master is increasing to a level where it is difficult to actually keep all metadata in main memory.

For these reasons, Google is now working on a new design featuring a distributed master solution.

Caching: As we saw in Chapter 12, *caching* often plays a crucial role in the performance and scalability of a file system (see also the more general discussion on caching in Section 2.3.1). Interestingly, GFS does not make heavy use of caching. As mentioned above, information about the locations of chunks is cached at clients when first accessed, to minimize interactions with the master. Apart from that, no other client caching is used. In particular, GFS clients do not cache file data. Given the fact that most accesses involve sequential streaming, for example reading through web content to produce the required inverted index, such caches would contribute little to the performance of the system. Furthermore, by limiting caching at clients, GFS also avoids the need for cache coherency protocols.

GFS also does not provide any particular strategy for server-side caching (that is, on chunkservers) rather relying on the buffer cache in Linux to maintain frequently accessed data in memory.

Logging: GFS is a key example of the use of *logging* in Google to support debugging and performance analysis. In particular, GFS servers all maintain extensive diagnostic logs that store significant server events and all RPC requests and replies. These logs are monitored continually and used in the event of system problems to identify the underlying causes.

Managing consistency in GFS • Given that chunks are replicated in GFS, it is important to maintain the consistency of replicas in the face of operations that alter the data – that is, the *write* and *record append* operations. GFS provides an approach for consistency management that:

- maintains the previously mentioned separation between control and data and hence allows high-performance updates to data with minimal involvement of masters;
- provides a relaxed form of consistency recognizing, for example, the particular semantics offered by *record append*.

The approach proceeds as follows.

When a mutation (i.e., a *write*, *append* or *delete* operation) is requested for a chunk, the master grants a chunk *lease* to one of the replicas, which is then designated as the *primary*. This primary is responsible for providing a serial order for all the currently pending concurrent mutations to that chunk. A global ordering is thus provided by the ordering of the chunk leases combined with the order determined by that primary. In particular, the lease permits the primary to make mutations on its local copies and to control the order of the mutations at the secondary copies; another primary will then be granted the lease, and so on.

The steps involved in mutations are therefore as follows (slightly simplified):

- On receiving a request from a client, the master grants a lease to one of the replicas (the primary) and returns the identity of the primary and other (secondary) replicas to the client.
- The client sends all data to the replicas, and this is stored temporarily in a buffer cache and not written until further instruction (again, maintaining a separation of control flow from data flow coupled with a lightweight control regime based on leases).
- Once all the replicas have acknowledged receipt of this data, the client sends a write request to the primary; the primary then determines a serial order for concurrent requests and applies updates in this order at the primary site.
- The primary requests that the same mutations in the same order are carried out at secondary replicas and the secondary replicas send back an acknowledgement when the mutations have succeeded'
- If all acknowledgements are received, the primary reports success back to the client; otherwise, a failure is reported indicating that the mutation succeeded at the primary and at some but not all of the replicas. This is treated as a failure and

leaves the replicas in an inconsistent state. GFS attempts to overcome this failure by retrying the failed mutations. In the worst case, this will not succeed and therefore consistency is not guaranteed by the approach.

It is interesting to relate this scheme to the techniques for replication discussed in Chapter 18. GFS adopts a passive replication architecture with an important twist. In passive replication, updates are sent to the primary and the primary is then responsible for sending out subsequent updates to the backup servers and ensuring they are coordinated. In GFS, the client sends data to all the replicas but the request goes to the primary, which is then responsible for scheduling the actual mutations (the separation between data flow and control flow mentioned above). This allows the transmission of large quantities of data to be optimized independently of the control flow.

In mutations, there is an important distinction between *write* and *record append* operations. *writes* specify an offset at which mutations should occur, whereas *record appends* do not (the mutations are applied at the end of the file wherever this might be at a given point in time). In the former case the location is predetermined, whereas in the latter case the system decides. Concurrent *writes* to the same location are not serializable and may result in corrupted regions of the file. With *record append* operations, GFS guarantees the append will happen at least once and atomically (that is, as a contiguous sequence of bytes); the system does not guarantee, though, that all copies of the chunk will be identical (some may have duplicate data). Again, it is helpful to relate this to the material in Chapter 18. The replication strategies in Chapter 18 are all general-purpose, whereas this strategy is domain-specific and weakens the consistency guarantees, knowing the resultant semantics can be tolerated by Google applications and services (a further example of domain-specific replication – the replication algorithm by Xu and Liskov [1989] for tuple spaces can be found in Section 6.5.2).

21.5.2 Chubby

Chubby [Burrows 2006] is a crucial service at the heart of the Google infrastructure offering storage and coordination services for other infrastructure services, including GFS and Bigtable. Chubby is a multi-faceted service offering four distinct capabilities:

- It provides coarse-grained distributed locks to synchronize distributed activities in what is a large-scale, asynchronous environment.
- It provides a file system offering the reliable storage of small files (complementing the service offered by GFS).
- It can be used to support the election of a primary in a set of replicas (as needed for example by GFS, as discussed in Section 21.5.1 above).
- It is used as a name service within Google.

At first sight, this might appear to contradict the overall design principle of simplicity (doing one thing and doing it well). As we unfold the design of Chubby, however, we will see that its heart is one core service that is offering a solution to *distributed consensus* and that the other facets emerge from this core service, which is optimized for the style of usage within Google.

We begin our study of Chubby by examining the interface it offers; then we look in detail at the architecture of a Chubby system and how this maps onto the physical architecture. We conclude the examination by looking in detail at the implementation of the consensus algorithm at the heart of Chubby, *Paxos*.

Chubby interface • Chubby provides an abstraction based on a file system, taking the view first promoted in Plan 9 [Pike *et al*. 1993] that every data object is a file. Files are organized into a hierarchical namespace using directory structures with names having the form of:

/ls/chubby_cell/directory_name/.../file_name

where */ls* refers to the lock service, designating that this is part of the Chubby system, and *chubby_cell* is the name of a particular instance of a Chubby system (the term *cell* is used in Chubby to denote an instance of the system). This is followed by a series of directory names culminating in a *file_name*. A special name, */ls/local* will be resolved to the most local cell relative to the calling application or service.

Chubby started off life as a lock service, and the intention was that everything would be a lock in the system. However, it quickly became apparent that it would be useful to associate (typically small) quantities of data with Chubby entities – we see an example of this below when we look at how Chubby is used in primary elections. Thus entities in Chubby share the functionality of locks and files; they can be used solely for locking, to store small quantities of data or to associate small quantities of data (effectively metadata) with locking operations.

A slightly simplified version of the API offered by Chubby is shown in Figure 21.10. *Open* and *Close* are standard operations, with *Open* taking a named file or directory and returning a Chubby *handle* to that entity. The client can specify various parameters associated with the *Open*, including declaring the intended usage (for example, for reading, writing or locking), and permissions checks are carried out at this stage using access control lists. *Close* simply relinquishes use of the handle. *Delete* is used to remove the file or directory (this operation fails if applied to a directory with children).

In the role of a file system, Chubby offers a small set of operations for *whole-file* reading and writing; these are single operations that return the complete data of the file and write the complete data of the file. This whole-file approach is adopted to discourage the creation of large files, as this is not the intended use of Chubby. The first operation, *GetContentsAndStat*, returns both the contents of the file and any metadata associated with the file (an associated operation, *GetStat*, just returns the metadata; a *ReadDir* operation is also provided to read names and metadata associated with children of a directory. *SetContents* writes the contents of a file and *SetACL* provides a means to set access control list data. The reading and writing of whole files are *atomic* operations.

In the role of a lock-management tool, the main operations provided are *Acquire*, *TryAcquire* and *Release*. *Acquire* and *Release* correspond to the operations of the same name as introduced in Section 16.4; *TryAcquire* is a non-blocking variant of *Acquire*. Note that although locks are *advisory* in Chubby an application or service must go through the proper protocol of acquiring and releasing locks. The developers of Chubby did consider an alternative of mandatory locks, whereby locked data is inaccessible to all other users and this is enforced by the system, but the extra flexibility and resilience

Figure 21.10 Chubby API

Role	*Operation*	*Effect*
General	*Open*	Opens a given named file or directory and returns a handle
	Close	Closes the file associated with the handle
	Delete	Deletes the file or directory
File	*GetContentsAndStat*	Returns (atomically) the whole file contents and metadata associated with the file
	GetStat	Returns just the metadata
	ReadDir	Returns the contents of a directory – that is, the names and metadata of any children
	SetContents	Writes the whole contents of a file (atomically)
	SetACL	Writes new access control list information
Lock	*Acquire*	Acquires a lock on a file
	TryAquire	Tries to acquire a lock on a file
	Release	Releases a lock

of advisory locks was preferred, leaving the responsibility of checking for conflicts to the programmer [Burrows 2006].

Should an application need to protect a file from concurrent access, it can use both the roles together, storing data in the file and acquiring locks before accessing this data.

Chubby can also be used to support a *primary election* in distributed systems – that is, the election of one replica as the primary in passive replication management (refer back to Sections 15.3 and 18.3.1 for discussions of election algorithms and passive replication respectively). First, all candidate primaries attempt to acquire a lock associated with the election. Only one will succeed. This candidate becomes the primary,S with all other candidates then being secondaries. The primary records its victory by writing its identity to the associated file, and other processes can then determine the identity of the primary by reading this data. As mentioned above, this is a key example of combining the roles of lock and file together for a useful purpose in a distributed system. This also shows how primary election can be implemented on top of a consensus service as an alternative to the algorithms such as the ring-based approach or the bully algorithm introduced in Section 15.3.

Finally, Chubby supports a simple *event mechanism* enabling clients to register when opening a file to receive event messages concerning the file. More specifically, the client can subscribe to a range of events as an option in the *Open* call. The associated events are then delivered asynchronously via callbacks. Examples of events include that the file contents have been modified, a handle has become invalid, and so on.

Chubby is very much a cut-down file system programming interface compared to, for example, POSIX. Not only does Chubby require read and update operations to apply to whole files, but it does not support operations to move files between directories, nor does it support symbolic or hard links. Also Chubby maintains only limited metadata (related to access control, versioning and a checksum to protect data integrity).

Chubby architecture • As mentioned above, a single instance of a Chubby system is known as a cell; each cell consists of a relatively small number of replicas (typically five) with one designated as the master. Client applications access this set of replicas via a Chubby library, which communicates with the remote servers using the RPC service described in Section 21.4.1. The replicas are placed at failure-independent sites to minimize the potential for correlated failures – for example, they will not be contained within the same rack. All replicas are typically contained within a given physical cluster, although this is not required for the correct operation of the protocol and experimental cells have been created that span Google data centres.

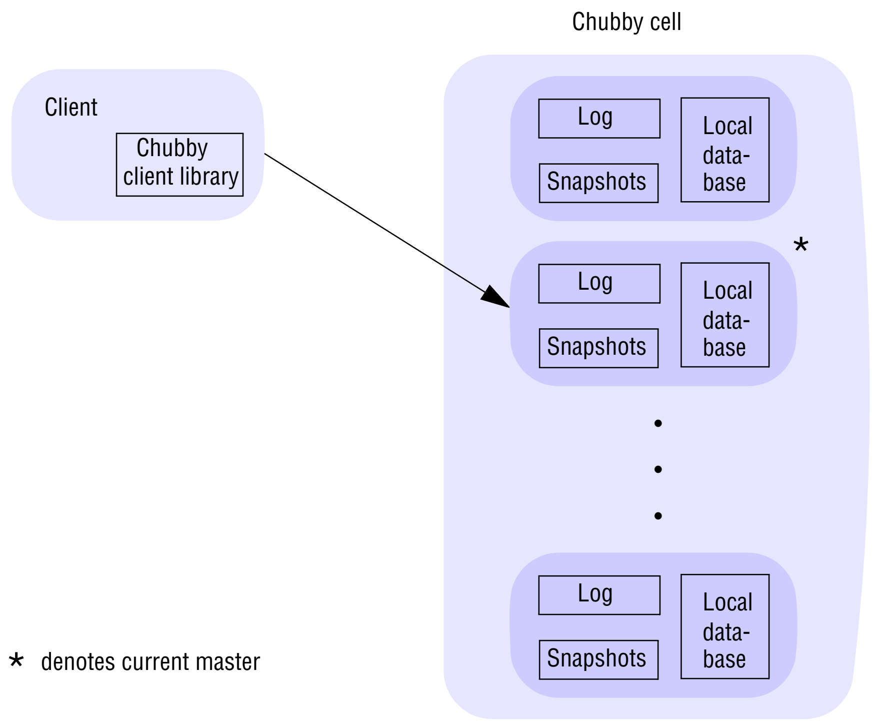
Each replica maintains a small database whose elements are entities in the Chubby namespace – that is, directories and files/locks. The consistency of the replicated database is achieved using an underlying consensus protocol (an implementation of Lamport's Paxos algorithm [Lamport 1989, Lamport 1998]) that is based around maintaining *operation logs* (we look at the implementation of this protocol below). As logs can become very large over time, Chubby also supports the creation of *snapshots* – complete views of system state at a given point of time. Once a snapshot is taken, previous logs can be deleted with the consistent state of the system at any point determined by the previous snapshot together with the applications of the set of operations in the log. This overall structure is shown in Figure 21.11.

A Chubby *session* is a relationship between a client and a Chubby cell. This is maintained using *KeepAlive* handshakes between the two entities. To improve performance, the Chubby library implements *client caching*, storing file data, metadata and information on open handles. In contrast to GFS (with its large, sequential reads and appends), client caching is effective in Chubby with its small files that are likely to be accessed repeatedly. Because of this caching, the system must maintain consistency between a file and a cache as well as between the different replicas of the file. The required *cache consistency* in Chubby is achieved as follows. Whenever a mutation is to occur, the associated operation (for example, *SetContents*) is blocked until all associated caches are invalidated (for efficiency, the invalidation requests are piggybacked onto *KeepAlive* replies from the master with the replies sent immediately when an invalidation occurs). Cached data is also never updated directly.

The end result is a very simple protocol for cache consistency that delivers deterministic semantics to Chubby clients. Contrast this with the client caching regime in NFS, for example, where mutations do not result in the immediate updating of cached copies, resulting in potentially different versions of files on different client nodes. It is also interesting to compare this with the cache consistency protocol in AFS, but we leave that as an exercise to the reader (see Exercise 21.7).

This determinism is important for many of the client applications and services that use Chubby (for example, Bigtable, as discussed in Section 21.5.3) to store access control lists. Bigtable requires consistent update of access control lists, across all replicas and in terms of cached copies. Note that it is this determinism that led to the use

Figure 21.11 Overall architecture of Chubby



of Chubby as a name server within Google. We mentioned in Section 13.2.3 that the DNS allows naming data to become inconsistent. While this is tolerable in the Internet, the developers of the Google infrastructure preferred the more consistent view offered by Chubby, using Chubby files to maintain name to address mappings. Burrows [2006] discusses this use of Chubby as a name service in more detail.

Implementing Paxos • Paxos is a family of protocols providing distributed consensus (see Section 15.5 for a wider discussion of distributed consensus protocols). Consensus protocols operate over a set of replicas with the goal of reaching agreement between the servers managing the replicas to update to a common value. This is achieved in an environment where:

- Replica servers may operate at an arbitrary speed and may fail (and subsequently recover).
- Replica servers have access to stable, persistent storage that survives crashes.
- Messages may be lost, reordered or duplicated. They are delivered without corruption but may take an arbitrarily long time to be delivered.

Paxos is therefore fundamentally a distributed consensus protocol for *asynchronous systems* (see Section 2.4.1) and indeed is the dominant offering in this space. The developers of Chubby stress that the above assumptions reflect the true nature of

Internet-based systems such as Google and caution practitioners about consensus algorithms that make stronger assumptions (for example, algorithms for synchronous systems) [Burrows 2006].

Recall from Chapter 15 that it is impossible to guarantee consistency in asynchronous systems but that various techniques have been proposed to work around this result. Paxos works by ensuring correctness but not liveness – that is, Paxos is not guaranteed to terminate (we return to this issue below once we have looked at the details of the algorithm).

The algorithm was first introduced by Leslie Lamport in 1989 in a paper called The Part-Time Parliament, [Lamport 1989, Lamport 1998]. Inspired by his description of Byzantine Generals (as discussed in Section 15.5.1), he again presented the algorithm with reference to an analogy, this time referring to behaviour of a mythical parliament on the Greek island of Paxos. Lamport writes amusingly about the reaction to this presentation on his web site [research.microsoft.com].

In the algorithm, any replica can submit a value with the goal of achieving consensus on a final value. In Chubby, agreement equates to all replicas having this value as the next entry in their update logs, thus achieving a consistent view of the logs across all sites. The algorithm is guaranteed to eventually achieve consensus if a majority of the replicas run for long enough with sufficient network stability. More formally, Kirsch and Amir [2008] present the following liveness properties for Paxos:

Paxos-L1 (Progress): If there exists a stable majority set of servers, then if a server in the set initiates an update, some member of the set eventually executes the update.

Paxos-L2 (Eventual Replication): If server s executes an update and there exists a set of servers containing s and r, and a time after which the set does not experience any communication or process failures, then r eventually executes the update.

The intuition here is that the algorithm cannot guarantee to reach consistency when the network behaves asynchronously but will eventually reach consistency when more synchronous (or stable) conditions are experienced.

The Paxos algorithm: The Paxos algorithm proceeds as follows:

Step 1: The algorithm relies on an ability to elect a *coordinator* for a given consensus decision. Recognizing that coordinators can fail, a flexible election process is adopted that can result in multiple coordinators coexisting, old and new, with the goal of recognizing and rejecting messages from old coordinators. To identify the right coordinator, an ordering is given to coordinators through the attachment of a *sequence number*. Each replica maintains the highest sequence number seen so far and, if bidding to be a coordinator, will pick a higher *unique* number and broadcast this to all replicas in a *propose* message.

It is clearly important that the sequence number picked by a potential coordinator is indeed unique, two (or more) coordinators must not be able to pick the same value. Let us assume we have n replicas. A unique sequence number can be guaranteed if every replica is assigned a unique identifier, i_r, between 0 and n–1, and then selects the smallest sequence number s that is larger than any sequence numbers seen so far, so that $s \ mod \ n = i_r$ (for example, if the number of replicas is 5, we look at the replica with unique identifier 3 and the last sequence number seen was 20, then this replica will pick a sequence number of 23 for its next bid).

If other replicas have not seen a higher bidder, they either reply with a *promise* message indicating that they promise to not deal with other (that is, older) coordinators with lower sequence numbers, or they send a negative acknowledgement indicating they will not vote for this coordinator. Each *promise* message also contains the most recent value the sender has heard as a proposal for consensus; this value may be null if no other proposals have been observed. If a majority of *promise* messages are received, the receiving replica is elected as a coordinator, with the majority of replicas supporting this coordinator known as the *quorum*.

Step 2: The elected coordinator must select a value and subsequently send an *accept* message with this value to the associated quorum. If any of the *promise* messages contained a value, then the coordinator must pick a value (any value) from the set of values it has received; otherwise, the coordinator is free to select its own value. Any member of the quorum that receives the *accept* message must accept the value and then *acknowledge* the acceptance. The coordinator waits, possibly indefinitely in the algorithm, for a majority of replicas to acknowledge the *accept* message.

Step 3: If a majority of replicas do acknowledge, then a consensus has effectively been achieved. The coordinator then broadcasts a *commit* message to notify replicas of this agreement. If not, then the coordinator abandons the proposal and starts again.

Note that the terminology above is that used by Google, for example in Chandra *et al.* [2007]. In the literature, descriptions of the protocol may use other terminology, for example based around the roles of proposers, acceptors, learners and so on.

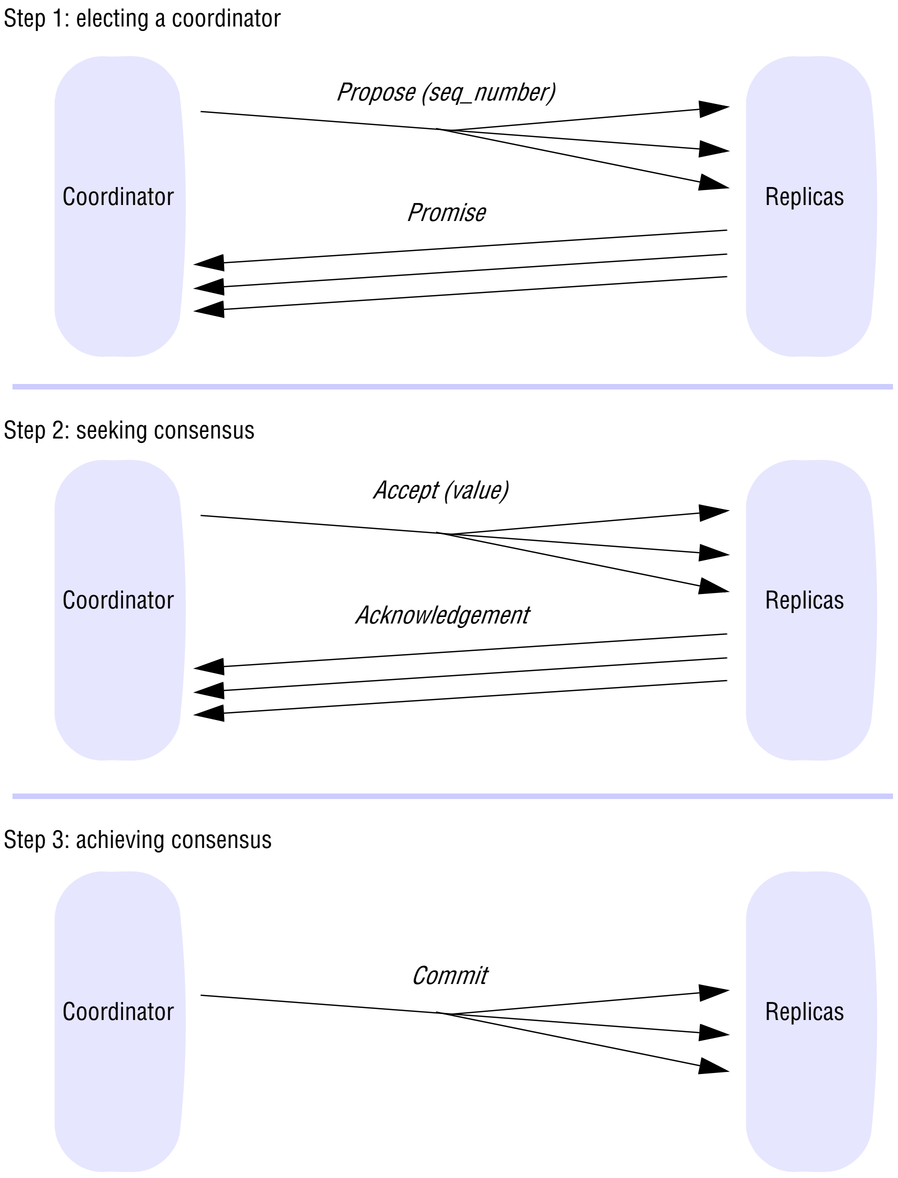
In the absence of failure, consensus is therefore achieved with the message exchanges shown in Figure 21.12. The algorithm is also safe in the presence of failures – for example, the failure of a coordinator or of another replica or problems with lost, reordered or duplicated or delayed messages, as discussed above. A proof of correctness is beyond the scope of this book but relies heavily on the ordering imposed by step 1 coupled with the fact that, because of the quorum mechanism, if two majorities have agreed on a proposed value there must be at least one replica in common that agreed to both. The quorum mechanism also ensures correct behaviour if the network partitions, since only at most one partition is able to construct a majority.

Returning to the issue of termination, it is possible for Paxos to fail to terminate if two proposers compete against each other and indefinitely outbid each other in terms of higher and higher sequence numbers. This is consistent with the impossibility result of Fischer *et al.* [1985] concerning absolute guarantees of consensus in asynchronous systems.

Additional implementation issues: In Chubby, it is not sufficient to reach agreement on a single value; there is a need to reach agreement on a sequence of values. In practice the algorithm must therefore repeat to agree a set of entries in the log. This is referred to as Multi-Paxos by Chandra *et al.* [2007]. In Multi-Paxos certain optimizations are possible, including the election of a coordinator for a (potentially long) period of time thus avoiding repeated executions of step 1.

The paper by Chandra *et al.* also discusses the engineering challenges of implementing Paxos in a real-world setting, and in particular in the complex distributed system setting offered by the Google infrastructure. In this entertaining and instructive

Figure 21.12 Message exchanges in Paxos (in absence of failures)



paper, they discuss the challenges of moving from algorithmic description and formal proof to making the algorithm operate effectively as part of the Chubby system, including dealing with disk corruptions and other contextual events such as system upgrades. The paper emphasizes the importance of a stringent testing regime, especially for such key building blocks of fault-tolerant systems, resonating with the overall Google principle of extensive testing mentioned in Section 21.3.

21.5.3 Bigtable

GFS offers a system for storing and accessing large ‘flat’ files whose content is accessed relative to byte offsets within a file, allowing programs to store large quantities of data and perform read and write (especially append) operations optimized for the typical use within the organization. While this is an important building block, it is not sufficient to meet all of Google’s data needs. There is a strong need for a distributed storage system that provides access to data that is indexed in more sophisticated ways related to its content and structure. Web search and nearly all of the other Google applications, including the crawl infrastructure, Google Earth/Maps, Google Analytics and personalized search, use structured data access. Google Analytics, for example, stores information on raw clicks associated with users visiting a web site in one table and summarizes the analyzed information in a second table. The former is around 200 terabytes in size and the latter 20 terabytes. (The analysis is carried out using MapReduce, described in Section 21.6 below.)

One choice for Google would be to implement (or reuse) a distributed database, for example a relational database with a full set of relational operators provided (for example, *union*, *selection*, *projection*, *intersection* and *join*). But the achievement of good performance and scalability in such distributed databases is recognized as a difficult problem and, crucially, the styles of application offered by Google do not demand this full functionality. Google therefore has introduced Bigtable [Chang *et al*. 2008], which retains the *table* model offered by relational databases but with a much simpler interface suitable for the style of application and service offered by Google and also designed to support the efficient storage and retrieval of quite massive structured datasets. We describe this interface in some detail below before looking at the internal architecture of Bigtable, highlighting how these properties are achieved.

Bigtable interface • Bigtable is a distributed storage system that supports the storage of potentially vast volumes of structured data. The name is strongly indicative of what it offers, providing storage for what are very big tables (often in the terabyte range). More precisely, Bigtable supports the fault-tolerant storage, creation and deletion of tables where a given table is a three-dimensional structure containing cells indexed by a row key, a column key and a timestamp:

Rows: Each row in a table has an associated row key that is an arbitrary string of up to 64 kilobytes in size, although most keys are significantly smaller. A row key is mapped by Bigtable to the address of a row. A given row contains potentially large amounts of data about a given entity such as a web page. Given that it is common within Google to process information about web pages, it is quite common, for example, for row keys to be URLs with the row then containing information about the resources referenced by the URLs. Bigtable maintains a lexicographic ordering of a given table by row key, and this has some interesting repercussions. In particular, as we will see below when we examine the underlying architecture, subsequences of rows map onto tablets, which are the unit of distribution and placement. Hence it is beneficial to manage locality by assigning row keys that will be close or even adjacent in the lexicographic order. This implies that URLs may make bad key choices, but URLs with the domain portion reversed will provide much stronger locality for data accesses because common domains will be sorted together, supporting domain analyses. To illustrate this, consider information stored on the BBC web site related to sport. If such information is stored under URLs such as www.bbc.co.uk/sports and www.bbc.co.uk/football, then the resultant sort will be rather random and dominated by the lexicographic order of early fields. If, however, it is stored under uk.co.bbc.www/sport and uk.co.bbc.www/football, the related information is likely to be stored in the same tablet. It should be stressed that this key assignment is left entirely to the programmer so they must be aware of this (ordering) property to exploit the system optimally. To deal with concurrency issues, all accesses to rows are atomic (echoing similar design decisions in GFS and Chubby).

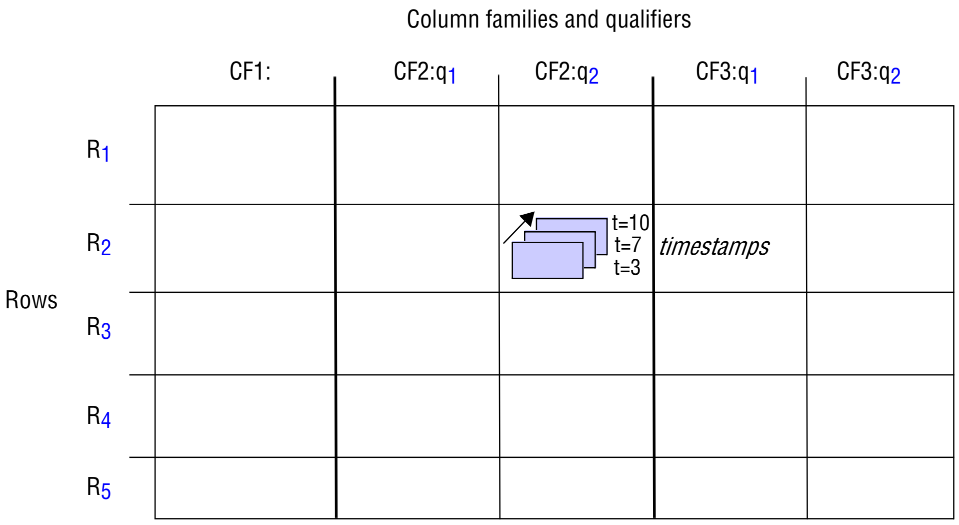
Columns: The naming of columns is more structured than that of rows. Columns are organized into a number of *column families* – logical groupings where the data under a family tends to be of the same type, with individual columns designated by *qualifiers* within families. In other words, a given column is referred to using the syntax *family:qualifier*, where *family* is a printable string and *qualifier* is an arbitrary string. The intended use is to have a relatively small number of families for a given table but a potentially large number of columns (designated by distinct qualifiers) within a family. Using the example from Chang *et al.* [2008], this can be used to structure data associated with web pages, with valid families being the *contents*, any *anchors* associated with the page and the *language* that is used in the web page. If a family name refers to just one column it is possible to omit the qualifier. For example, a web page will have one contents field, and this can be referred to using the key name *contents:*.

Timestamps: Any given cell within Bigtable can also have multiple versions indexed by timestamp, where the timestamp is either related to real time or can be an arbitrary value assigned by the programmer (for example, a *logical time*, as discussed in Section 14.4, or a version identifier). The various versions are sorted by reverse timestamp with the most recent version available first. This facility can be used, for example, to store different versions of the same data, including the content of web pages, allowing analyses to be carried out over historical data as well as the current data. Tables can be set up to apply garbage collection on older versions automatically, therefore reducing the burden on the programmer to manage the large datasets and associated versions. This three-dimensional table abstraction is illustrated in Figure 21.13.

Bigtable supports an API that provides a wide range of operations, including:

- the creation and deletion of tables;
- the creation and deletion of column families within tables;
- accessing data from given rows;
- writing or deleting cell values;
- carrying out atomic row mutations including data accesses and associated write and delete operations (more global, cross-row transactions are not supported);
- iterating over different column families, including the use of regular expressions to identify column ranges;
- associating metadata such as access control information with tables and column families.

Figure 21.13 The table abstraction in Bigtable



As can be seen, Bigtable is considerably simpler than a relational database but well suited to the styles of application within Google. Chang *et al.*discuss how this interface supports the storage of tables of data on web pages (where the rows represent individual web pages and the columns represent data and metadata associated with that given web page), the storage of both raw and processed data for Google Earth (with rows representing geographical segments and columns being different images available for that segment), and also data to support Google Analytics (for example, maintaining a raw click table where rows represent an end user session and columns the associated activity).

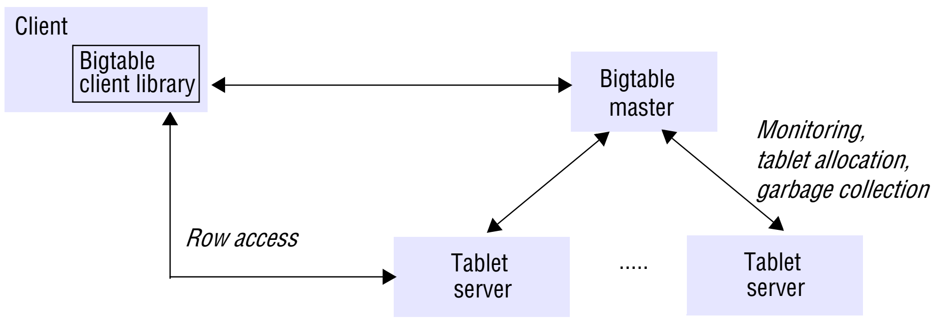
The overall architecture of the underlying system is presented below.

Bigtable architecture • A Bigtable is broken up into *tablets*, with a given tablet being approximately 100–200 megabytes in size. The main tasks of the Bigtable infrastructure are therefore to manage tablets and to support the operations described above for accessing and changing the associated structured data. The implementation also has the task of mapping the tablet structure onto the underlying file system (GFS) and ensuring effective load balancing across the system. As we shall see below, Bigtable makes heavy use of both GFS and Chubby for the storage of data and distributed coordination.

A single instance of a Bigtable implementation is known as a *cluster*, and each cluster can store a number of tables. The architecture of a Bigtable cluster is similar to that of GFS, consisting of three major components (as shown in Figure 21.14):

- a *library* component on the client side;
- a *master server*;
- a potentially large number of *tablet servers*.

Figure 21.14 Overall architecture of Bigtable



In terms of scale, Chang *et al*. report that, as of 2008, 388 production clusters ran across multiple Google machine clusters, with an average of around 63 tablet servers per cluster but with many being significantly larger (some with more than 500 tablet servers per cluster). The number of tablet servers per cluster is also dynamic, and it is common to add new tablet servers to the system at runtime to increase throughput.

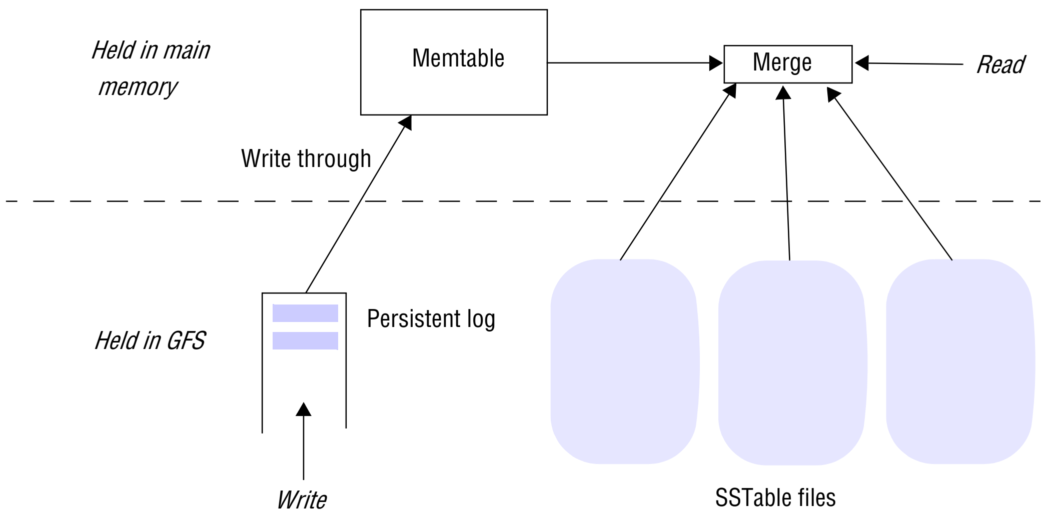
Two of the key design decisions in Bigtable are identical to those made for GFS. Firstly, Bigtable adopts a *single master* approach, for exactly the same reasons – that is, to maintain a centralized view of the state of the system thus supporting optimal placement and load-balancing decisions and because of the inherent simplicity in implementing this approach. Secondly, the implementation maintains a strict *separation between control and data* with a lightweight control regime maintained by the master and data access entirely through appropriate tablet servers, with the master not involved at this stage (to ensure maximum throughput when accessing large datasets by interacting directly with the tablet servers). In particular, the control tasks associated with the master are as follows:

- monitoring the status of tablet servers and reacting to both the availability of new tablet servers and the failure of existing ones;
- assigning tablets to tablet servers and ensuring effective load balancing;
- garbage collection of the underlying files stored in GFS.

Bigtable goes further than GFS in that the master is not involved in the core task of mapping tablets onto the underlying persistent data (which is, as mentioned above, stored in GFS). This means that Bigtable clients do not have to communicate with the master at all (compare this with the *open* operation in GFS, which does involve the master), a design decision that significantly reduces the load on the master and the possibility of the master becoming a bottleneck.

We now look at how Bigtable uses GFS for storing its data and uses Chubby in rather innovative ways for implementing monitoring and load balancing.

Figure 21.15 The storage architecture in Bigtable



Data storage in Bigtable: The mapping of tables in Bigtable onto GFS involves several stages, as summarized below:

- A table is split into multiple tablets by dividing the table up by row, taking a row range up to a size of around 100–200 megabytes and mapping this onto a tablet. A given table will therefore consist of multiple tablets depending on its size. As tables grow, extra tablets will be added.
- Each tablet is represented by a storage structure that consists of set of files that store data in a particular format (the SSTable) together with other storage structures implementing logging.
- The mapping from tablets to SSTables is provided by a hierarchical index scheme inspired by B^+-trees.

We look at the storage representation and mapping in more detail below.

The precise storage representation of a tablet in Bigtable is shown in Figure 21.15. The main unit of storage in Bigtable is the *SSTable* (a file format that is also used elsewhere in the Google infrastructure). An SSTable is organized as an ordered and immutable map from keys to values, with both being arbitrary strings. Operations are provided to efficiently read the value associated with a given key and to iterate over a set of values in a given key range. The index of an SSTable is written at the end of the SSTable file and read into memory when an SSTable is accessed. This means that a given entry can be read with a single disk read. An entire SSTable can optionally be stored in main memory.

A given tablet is represented by a number of SSTables. Rather than performing mutations directly on SSTables, writes are first committed to a log to support recovery (see Chapter 17), with the log also held in GFS. The log entries are written through to the *memtable* held in main memory. The SSTables therefore act as a snapshot of the state of a tablet and, on failure, recovery is implemented by replaying the most recent log entries since the last snapshot. Reads are serviced by providing a merged view of the data from the SSTables combined with the memtable. Different levels of compaction are performed on this data structure to maintain efficient operation, as reported in Chang *et al*. [2008]. Note that SSTables can also be compressed to reduce the storage requirements of particular tables in Bigtable. Users can specify whether tables are to be compressed and also the compression algorithm to be used.

As mentioned above, the master is not involved in the mapping from tables to stored data. Rather, this is managed by traversing an index based on the concept of B$^+$-trees (a form of B-tree where all the actual data is held in leaf nodes, with other nodes containing indexing data and metadata).

A Bigtable client seeking the location of a tablet starts the search by looking up a particular file in Chubby that is known to hold the location of a *root tablet* – that is, a tablet containing the root index of the tree structure. This root tablet contains metadata about other tablets – specifically about other metadata tablets, which in turn contain the location of the actual data tablets. The root tablet together with the other metadata tablets form a metadata table, with the only distinction being that the entries in the root tablet contain metadata about metadata tablets, which in turn contain metadata about the actual data tablets. With this scheme, the depth of the tree is limited to three. The entries in the metadata table map portions of tablets onto location information, including information about the storage representation for this tablet (including the set of SSTables and the associated log).

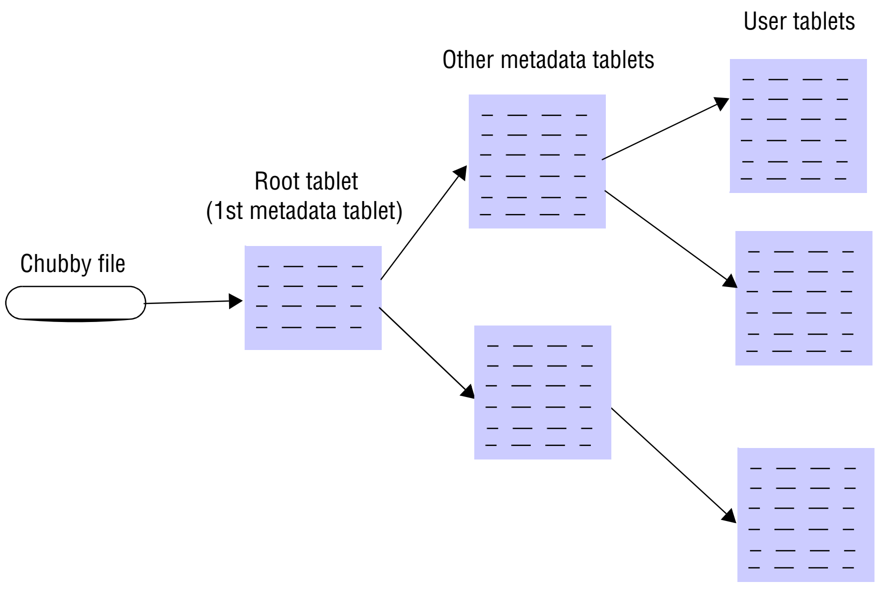
This overall structure is represented in Figure 21.16. To shortcut this three-level hierarchy, clients cache location information and also prefetched metadata associated with other tables when accessing the data structure.

Monitoring: Bigtable uses Chubby in a rather interesting way to monitor tablet servers. Bigtable maintains a directory in Chubby containing files representing each of the available tablet servers. When a new tablet server comes along, it creates a new file in this directory and, crucially, obtains an exclusive lock on this file. The existence of this file acts as the flag that the tablet server is fully operational and ready to be assigned tablets by the master, with the lock providing a means of communication between the two parties:

From the tablet server side: every tablet server monitors its exclusive lock and, if this is lost, it stops serving its tablets. This is most likely due to a network partition that compromises the Chubby session. The tablet server will attempt to reacquire the exclusive lock if the file still exists (see below), and if the file disappears, the server terminates itself. If a server terminates for another reason, for example because it is informed that its machine is needed for another purpose, the tablet server can surrender its exclusive lock, thus triggering a reassignment.

From the master side: the master periodically requests the status of the lock. If the lock is lost or if a tablet server does not respond, then clearly there is a problem either with the tablet server or with Chubby. The master attempts to acquire the lock, and if it succeeds it can infer that Chubby is alive and that the problem rests with the tablet server. The master then deletes the file from the directory, which will result in the tablet server terminating itself if it has not already failed. The master then must reassign all of that server's tablets to alternative tablet servers.

Figure 21.16 The hierarchical indexing scheme adopted by Bigtable



The rationale is to reuse Chubby, which is a well-tested and reliable service, to achieve the extra level of monitoring rather than providing a specific monitoring service specifically for this purpose.

Load balancing: To assign tablets, the master must map the available tablets in the cluster to appropriate tablet servers. From the algorithm above, the master has an accurate list of tablet servers that are ready and willing to host tablets and a list of all the tablets associated with the cluster. The master also maintains the current mapping information together with a list of unassigned tablets (which is populated, for example, when a tablet server is removed from the system). By having this global view of the system, the master ensures unassigned tablets are assigned to appropriate tablet servers based on responses to load requests, updating the mapping information accordingly.

Note that a master also has an exclusive lock of its own (the master lock), and if this is lost due to the Chubby session being compromised, the master must terminate itself (again, reusing Chubby to implement additional functionality). This does not stop access to data but rather prevents control operations from proceeding. Bigtable is therefore still available at this stage. When the master restarts, it must retrieve the current status. It does this by first creating a new file and obtaining the exclusive lock ensuring it is the only master in the cluster, and then working through the directory to find tablet servers, requesting information on tablet assignments from the tablet servers and also building a list of all tablets under its responsibility to infer unassigned tablets. The master then proceeds with its normal operation.

Figure 21.17 Summary of design choices related to data storage and coordination

Element	*Design choice*	*Rationale*	*Trade-offs*
GFS	The use of a large chunk size (64 megabytes)	Suited to the size of files in GFS; efficient for large sequential reads and appends; minimizes the amount of metadata	Would be very inefficient for random access to small parts of files
	The use of a centralized master	The master maintains a global view that informs management decisions; simpler to implement	Single point of failure (mitigated by maintaining replicas of operations logs)
	Separation of control and data flows	High-performance file access with minimal master involvement	Complicates the client library as it must deal with both the master and chunkservers
	Relaxed consistency model	High performance, exploiting semantics of the GFS operations	Data may be inconsistent, in particular duplicated
Chubby	Combined lock and file abstraction	Multipurpose, for example supporting elections	Need to understand and differentiate between different facets
	Whole-file reading and writing	Very efficient for small files	Inappropriate for large files
	Client caching with strict consistency	Deterministic semantics	Overhead of maintaining strict consistency
Bigtable	The use of a table abstraction	Supports structured data efficiently	Less expressive than a relational database
	The use of a centralized master	As above, master has a global view; simpler to implement	Single point of failure; possible bottleneck
	Separation of control and data flows	High-performance data access with minimal master involvement	-
	Emphasis on monitoring and load balancing	Ability to support very large numbers of parallel clients	Overhead associated with maintaining global states

21.5.4 Summary of key design choices

The overall design choices relating to data storage and coordination services are summarized in Figure 21.17.

The most striking feature emerging from this analysis is the design choice of providing three separate services that individually are relatively simple and targeted towards a given style of usage but together provide excellent coverage for the needs of Google applications and services. This is most evident from the complementary styles offered by GFS and Chubby, with Bigtable then providing structured data building on the services offered by both the underlying services. This design choice also echoes the approach adopted for communication paradigms (see Section 21.4.3) whereby multiple techniques are offered, each optimized for the intended style of application.

21.6 Distributed computation services

To complement the storage and coordination services, it is also important to support high-performance distributed computation over the large datasets stored in GFS and Bigtable. The Google infrastructure supports distributed computation through the MapReduce service and also the higher-level Sawzall language. We look at MapReduce in detail and then briefly examine the key features of the Sawzall language.

21.6.1 MapReduce

Given the huge datasets in use at Google, it is a strong requirement to be able to carry out distributed computation by breaking up the data into smaller fragments and carrying out analyses of such fragments in parallel, making use of the computational resources offered by the physical architecture described in Section 21.3.1. Such analyses include common tasks such as sorting, searching and constructing inverted indexes (indexes that contain a mapping from words to locations in different files, this being key in implementing search functions). MapReduce [Dean and Ghemawat 2008] is a simple programming model to support the development of such applications, hiding underlying detail from the programmer including details related to the parallelization of the computation, monitoring and recovery from failure, data management and load balancing onto the underlying physical infrastructure.

We look at details of the programming model offered by MapReduce before examining how the system is implemented.

MapReduce interface • The key principle behind MapReduce is the recognition that many parallel computations share the same overall pattern – that is:

- break the input data into a number of chunks;
- carry out initial processing on these chunks of data to produce intermediary results;
- combine the intermediary results to produce the final output.

The specification of the associated algorithm can then be expressed in terms of two functions, one to carry out the initial processing and the second to produce the final results from the intermediary values. It is then possible to support multiple styles of computation by providing different implementations of these two functions. Crucially, by factoring out these two functions, the rest of the functionality can be shared across the different computations, thus achieving huge reductions in complexity in constructing such applications.

More specifically, MapReduce specifies a distributed computation in terms of two functions, *map* and *reduce* (an approach partially influenced by the design of functional programming languages such as Lisp, which provide functions of the same name, although in functional programming the motivation is not parallel computation):

- *map* takes a set of key-value pairs as input and produces a set of intermediary key-value pairs as output.

- The intermediary pairs are then sorted by key value so that all intermediary results are ordered by intermediary key. This is broken up into groups and passed to *reduce* instances, which carry out their processing to produce a list of values for each group (for some computations, this could be a single value).

To illustrate the operation of MapReduce, let us consider a simple example. In section 21.2, we illustrated the various aspects of a web search for '*distributed systems book*' Let us simplify this further by just searching for this complete string – that is, a search for the phrase 'distributed systems book' as it appears in a large body of content such as the crawled contents of the Web. In this example, the *map* and *reduce* functions would perform the following tasks:

- Assuming it is supplied with a web page name and its contents as input, the *map* function searches linearly through the contents, emitting a key-value pair consisting of (say) the phrase followed by the name of the web document containing this phrase wherever it finds the strings 'distributed' followed by 'system' followed by 'book' (the example can be extended to also emit a position within the document).

- The *reduce* function is in this case is trivial, simply emitting the intermediary results ready to be collated together into a complete index.

The MapReduce implementation is responsible for breaking the data into chunks, creating multiple instances of the *map* and *reduce* functions, allocating and activating them on available machines in the physical infrastructure, monitoring the computations for any failures and implementing appropriate recovery strategies, despatching intermediary results and ensuring optimal performance of the whole system.

With this approach, it is possible to make significant savings in terms of lines of code by reusing the underlying MapReduce framework. For example, Google reimplemented the main production indexing system in 2003 and reduced the number of lines of C++ code in MapReduce from 3,800 to 700 – a significant reduction, albeit in a relatively small system. This also results in other key benefits, including making it easier to update algorithms as there is a clean separation of concerns between what is effectively the application logic and the associated management of the distributed computation (a similar principle to the separation of concerns intrinsic to container-based systems, as reported in Section 8.4). In addition, improvements to the underlying MapReduce implementation immediately benefit all MapReduce applications. The downside is a more prescriptive framework, albeit one that can be customized by specifying the *map* and *reduce* and indeed other functions, as will become apparent below.

Figure 21.18 Examples of the use of MapReduce

Function	*Initial step*	*Map phase*	*Intermediate step*	*Reduce phase*
Word count		For each occurrence of word in data partition, emit <*word*, 1>		For each word in the intermediary set, count the number of 1s
Grep		Output a line if it matches a given pattern		Null
Sort *N.B. This relies heavily on the intermediate step*	*Partition data into fixed-size chunks for processing*	For each entry in the input data, output the key-value pairs to be sorted	*Merge/sort all key-value keys according to their intermediary key*	Null
Inverted index		Parse the associated documents and output a <*word*, document ID> pair wherever that word exists		For each word, produce a list of (sorted) document IDs

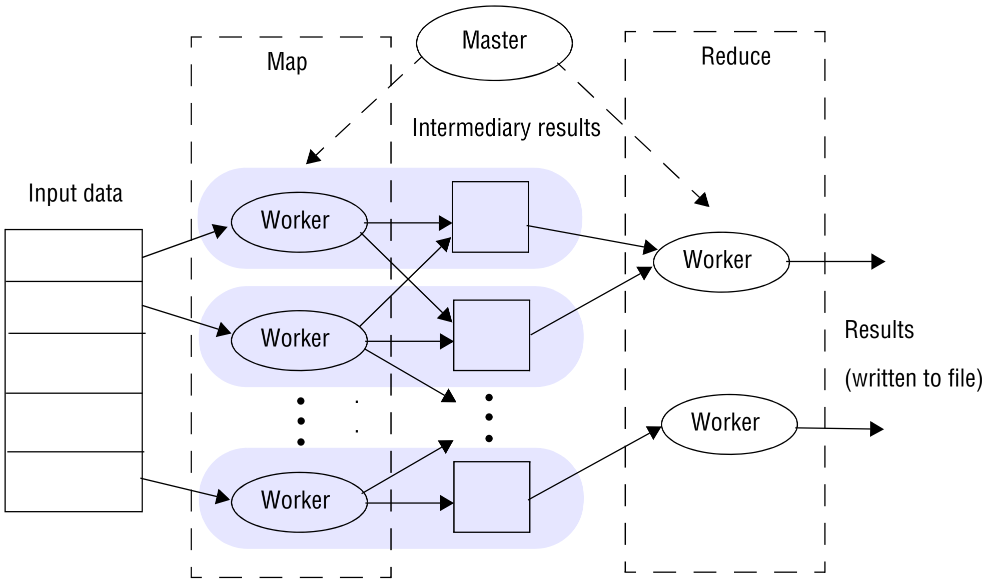
To further illustrate the use of MapReduce, we provide in Figure 21.18 a set of examples of common functions and how they would be implemented using *map* and *reduce* functions. The shared steps in the computation performed by the MapReduce framework are also shown for completeness. Further details of these examples can be found in Dean and Ghemawat [2004].

MapReduce architecture • MapReduce is implemented by a library that, as mentioned above, hides the details associated with parallelization and distribution and allows the programmer to focus on specifying the *map* and *reduce* functions. This library is built on top of other aspects of the Google infrastructure, in particular using RPC for communication and GFS for the storage of intermediary values. It is also common for MapReduce to take its input data from Bigtable and produce a table as a result, for example as with the Google Analytics example mentioned above (Section 21.5.3).

The overall execution of a MapReduce program is captured in Figure 21.19 which shows the key phases involved in execution:

- The first stage is to split the input file into *M* pieces, with each piece being typically 16–64 megabytes in size (therefore no bigger than a single chunk in GFS). The actual size is tunable by the programmer and therefore the programmer is able to optimize this for the particular parallel processing to follow. The key space associated with the intermediary results is also partitioned into *R* pieces using a (programmable) partition function. The overall computation therefore involves *M map* executions and *R reduce* executions.

Figure 21.19 The overall execution of a MapReduce program



- The MapReduce library then starts a set of worker machines (*workers*) from the pool available in the cluster, with one being designated as the *master* and others being used for executing *map* or *reduce* steps. The number of workers is normally much less than *M*+*R*. For example, Dean and Ghemawat [2008] report on typical figures of *M*=200,000, *R*=5000 with 2000 worker machines allocated to the task. The goal of the master is to monitor the state of workers and allocate idle workers to tasks, the execution of *map* or *reduce* functions. More precisely, the master keeps track of the status of *map* and *reduce* tasks in terms of being *idle*, *in-progress* or *completed* and also maintains information on the location of intermediary results for passing to workers allocated a *reduce* task.
- A worker that has been assigned a *map* task will first read the contents of the input file allocated to that *map* task, extract the key-value pairs and supply them as input to the *map* function. The output of the *map* function is a processed set of key/value pairs that are held in an intermediary buffer. As the input data is stored in GFS, the file will be replicated on (say) three machines. The master attempts to allocate a worker on one of these three machines to ensure locality and minimize the use of network bandwidth. If this is not possible, a machine near the data will be selected.
- The intermediary buffers are periodically written to a file local to the map computation. At this stage, the data are partitioned according to the partition function, resulting in *R* regions. This partition function, which is crucial to the operation of MapReduce, can be specified by the programmer, but the default is to perform a hash function on the key and then apply modulo *R* to the hashed value to produce *R* partitions, with the end result that intermediary results are grouped according to the hash value. Dean and Ghemawat [2004] provide the alternative

example where keys are URLs and the programmer wants to group intermediary results by the associated host: *hash(Hostname(key)) mod R*. The master is notified when partitioning has completed and is then able to request the execution of associated *reduce* functions.

- When a worker is assigned to carry out a *reduce* function, it reads its corresponding partition from the local disk of the map workers using RPC. This data is sorted by the MapReduce library ready for processing by the *reduce* function. Once sorting is completed, the *reduce* worker steps through the key-value pairs in the partition applying the *reduce* function to produce an accumulated result set, which is then written to an output file. This continues until all keys in the partition are processed.

Achieving fault tolerance: The MapReduce implementation provides a strong level of fault tolerance, in particular guaranteeing that if the *map* and *reduce* operations are *deterministic* with respect to their inputs (that is they always produce the same outputs for a given set of inputs), then the overall MapReduce task will produce the same output as a sequential execution of the program, even in the face of failures.

To deal with failure, the master sends a *ping* message periodically to check that a worker is alive and carrying out its intended operation. If no response is received, it is assumed that the worker has failed and this is recorded by the master. The subsequent action then depends on whether the task executing was a *map* or a *reduce* task:

- If the worker was executing a *map* task, this task is marked as *idle*, implying it will then be rescheduled. This happens irrespective of whether the associated task is in progress or completed. Remember that results are stored on local disks, and hence if the machine has failed the results will be inaccessible.
- If the worker was executing a *reduce* task, this task is marked as *idle* only if it was still in progress; if it is completed, the results will be available as they are written to the global (and replicated) file system.

Note that to achieve the desired semantics, it is important that the outputs from *map* and *reduce* tasks are written atomically, a property ensured by the MapReduce library in cooperation with the underlying file system. Details of how this is achieved can be found in Dean and Ghemawat [2008].

MapReduce also implements a strategy to deal with workers that may be taking a long time to complete (known as *stragglers*). Google has observed that it is relatively common for some workers to run slowly, for example because of a faulty disk that may perform badly due to a number of error-correction steps involved in data transfers. To deal with this, when a program execution is close to completion, the master routinely starts backup workers for all remaining *in-progress* tasks. The associated tasks are marked as *completed* when either the original or the new worker completes. This is reported as having a significant impact on completion times, again circumventing the problem of working with commodity machines that can and do fail.

As mentioned above, MapReduce is designed to operate together with Bigtable in processing large volumes of structured data. Indeed, within Google it is common to find applications that use a mix of all the infrastructural elements. In the box on page 961, we describe the support provided to the Google Maps and Google Earth applications by MapReduce, Bigtable and GFS.

Supporting Google Maps and Google Earth

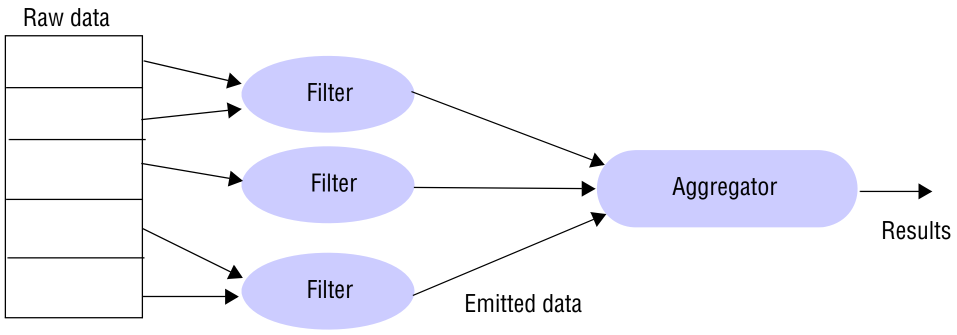
Client programs for Google Maps [maps.google.com] and Google Earth [earth.google.com] rely on the availability of huge sets of *image tiles* for loading by the clients from Google servers. Image tiles are square arrays of pixel values containing rendered images of geographic features and are organized in layers holding different types of geographic features. A base layer of tiles showing a street map is built from an up-to-date geographic database and another layer is built from scaled satellite and aerial images showing physical characteristics of the Earth's surface. Other partially transparent layers are also held and can be called up to show public transport networks and other infrastructure features, elevation contours and even real-time traffic flows. The tile sets for each layer cover the Earth's entire land surface and are replicated to show different levels of detail at up to 20 zoom levels (i.e., scales).

Much of the basic geographic data changes only slowly, but data defining new and changed roads and other physical infrastructure becomes available all the time and the consequent regeneration of tile sets calls for a high-performance distributed application running on Google servers that converts geographic vectors, points and raw image data to tiles. The implementation makes heavy use of Bigtable to store the underlying data. The basic geographic data is stored in an XML format known as Keyhole Markup Language or KML (Keyhole being the name of the company that first developed the software, acquired by Google in 2004). Raw vector and image data are received in many formats and resolutions from a variety of sources, including satellite and airborne imagery, and are stored with KML metadata in a single table where the rows represent particular geographical locations and the columns represent different geographic features and raw images, organized in families of columns. The naming scheme for rows ensures that physical features that are close together are stored in adjacent rows so that the data needed to generate each tile will lie in one tablet, or not more than a small number of them. This table is around 70 terabytes in size and has nine billion cells. It is relatively sparse as there are normally few features or images per geographical location.

Data are continually added to the table as more geographic data and imagery become available. At selected time points, the addition of data is suspended and a tile update begins. A set of concurrent *Map* processes (as in MapReduce) work on the raw data to transform and correct all the georeferencing coordinates for the flat presentation of the data and to blend the images together. This *Map* stage generates a table structure containing locally sorted geographic data that is passed to a set of concurrent *Reduce* processes that render tiles as raster images. The entire MapReduce task takes around 8 hours to generate a complete set of tiles processing raw data at around 1 megabyte per second [Chang *et al*. 2008].

The resultant image tiles are stored in GFS, with an associated index stored in another table in Bigtable. This table is heavily replicated across hundreds of tablet servers on clusters at several data centres, thus enabling it to serve the very large numbers of concurrent users of Google Maps and Google Earth. The index is around 500 gigabytes in size and significant portions are held in main memory, reducing the latency associated with reads.

Figure 21.20 The overall execution of a Sawzall program



21.6.2 Sawzall

Sawzall [Pike *et al*. 2005] is an interpreted programming language for performing parallel data analysis over very large datasets in highly distributed environments such as that provided by the physical The Google infrastructure. Although MapReduce readily supports the construction of such highly parallel and distributed programs, the goal of Sawzall is to simplify the construction of such programs. This is borne out in practice, with Sawzall programs often being 10 to 20 times smaller than the equivalent programs written for MapReduce [Pike *et al*. 2005]. The implementation of the Sawzall language builds on much of the existing The Google infrastructure, making use of MapReduce to create and manage the underlying parallel executions, of GFS to store data associated with the computation and of protocol buffers to provide a common data format for stored records.

Like MapReduce, Sawzall assumes that parallel computations follow a given pattern, which we have summarized in Figure 21.20. Sawzall assumes that the input to a computation consists of *raw data*, which in turn consists of a set of records to be processed. Computations then proceed by executing *filters*, which process each record in parallel, producing *emitted* results. *Aggregators* take the emitted data and produce the overall results of the computation.

Sawzall also makes two assumptions about the execution of filters and aggregators:

- The execution of filters and aggregators should be *commutative* across all records; that is, filters can be executed in any order and the result will be the same.
- The aggregator operations should be *associative*. That is the (implicit) parentheses in execution do not matter, allowing more degrees of freedom in execution.

As might be expected from examining MapReduce, Sawzall programs that express the filter operations and data emissions run in the map phase of MapReduce, with aggregators corresponding to the reduce phase. A set of predefined aggregators are provided by the language, including aggregators that carry out a summation of all the emitted values (*sum*) or build a collection of all the emitted values (*collection*). Other aggregators are more statistical, for example constructing a cumulative probability distribution (*quantile*) or estimating the values that are most common (*top*). It is also

Figure 21.21 Summary of design choices related to distributed computation)

Element	*Design choice*	*Rationale*	*Trade-offs*
MapReduce	The use of a common framework	Hides details of parallelization and distribution from the programmer; improvements to the infrastructure immediately exploited by all MapReduce applications	Design choices within the framework may not be appropriate for all styles of distributed computation
	Programming of system via two operations, *map* and *reduce*	Very simple programming model allowing rapid development of complex distributed computations	Again, may not be appropriate for all problem domains
	Inherent support for fault-tolerant distributed computations	Programmer does not need to worry about dealing with faults (particularly important for long-running tasks running over a physical infrastructure where failures are expected)	Overhead associated with fault-recovery strategies
Sawzall	Provision of a specialized programming language for distributed computation	Again, support for rapid development of often complex distributed computations with complexity hidden from the programmer (even more so than with MapReduce)	Assumes that programs can be written in the style supported (in terms of filters and aggregators)

possible for a programmer to develop new aggregators, although this is expected to be relatively rare.

We illustrate the use of Sawzall by a single, simple example from Pike *et al.* [2005] illustrating the above features:

```
count: table sum of int;
total: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
```

This program takes as input simple records of type *float* (a stream of values accessed through the local variable *x*). The program also defines two aggregators introduced with the keyword *table*, with the added keyword *sum* indicating that these are summation aggregators (this keyword could equally have been one of *collection*, *quantile* or *top*, for example). The calls of *emit* produce a stream of values that are processed by the aggregators producing the desired results (in this case, a count of all the values in the input stream together with a sum of all these values).

A full description of the Sawzall language and further examples can be found in Pike *et al.*

21.6.3 Summary of key design choices

The overall design choices relating to MapReduce and Sawzall are summarized in Figure 21.21.

The overall benefits in both approaches stem from encouraging a particular style of distributed computation and then providing common infrastructure to enable efficient implementation of systems developed using this style. This approach has been demonstrated to be effective right across Google applications and services, including support of the core search functionality and in the demanding area of supporting cloud applications such as Google Earth.

This work has sparked an interesting debate in the data management community as to whether such abstractions are sufficient for all classes of application. For an insight into this debate, refer to papers by Dean and Ghemawat [2010] and Stonebraker *at al*. [2010] in *Communications of the ACM*.

21.7 Summary

This chapter concludes the book by addressing the key issue of how one very large Internet enterprise has approached the design of a distributed system to support a demanding set of real-world applications. This is a very challenging topic and one that requires a thorough understanding of the technological choices available to distributed systems developers at all levels of system development, including communication paradigms, available services and associated distributed algorithms. The inevitable trade-offs associated with the design choices demand a thorough understanding of the application domain.

The approach taken in this chapter is to highlight the art of distributed systems design through a substantial case study – that is, the examination of the design of the underlying Google infrastructure, the platform and middleware used by Google to support its search engine and expanding set of applications and services. This is a compelling case study as it addresses what is the most complex and large-scale distributed system ever constructed, and one that has demonstrably met its design requirements.

This case study examined the overall architecture of the system together with in-depth studies of the key underlying services – specifically, protocol buffers, the publish-subscribe service, GFS, Chubby, Bigtable, MapReduce and Sawzall – which all work together to support complex distributed applications and services including the core search engine and Google Earth. One key lesson to be taken from this case study is the importance of really understanding your application domain, deriving a core set of underlying design principles and applying them consistently. In the case of Google, this manifests itself in a strong advocacy of simplicity and low-overhead approaches coupled with an emphasis on testing, logging and tracing. The end result is an architecture that is highly scalable, reliable, high performance and open in terms of supporting new applications and services.

The Google infrastructure is one of a number of middleware solutions for cloud computing that have emerged in recent years (albeit only fully available within Google).

Other solutions include the Amazon Web Services (AWS) [aws.amazon.com], Microsoft's Azure [www.microsoft.com IV] and open source solutions including Hadoop (which includes an implementation of MapReduce) [hadoop.apache.org], Eucalyptus [open.eucalyptus.com], the Google App Engine (available externally and providing a window on some but not all of the functionality offered by the Google infrastructure) [code.google.com IV] and Sector/Sphere [sector.sourceforge.net]. OpenStreetMap [www.openstreetmap.org], an open alternative to Google Maps that operates in a similar manner using voluntarily developed software and non-commercial servers, has also been developed. Details of these implementations are generally available and the reader is encouraged to study a selection of these architectures, comparing the design choices with those presented in the above case study.

Beyond that, there is a real paucity of published case studies related to distributed systems design, and this is a pity given the potential educational value of studying overall distributed systems architectures and their associated design principles. The main contribution of this chapter is therefore to provide a first in-depth case study illustrating the complexities of designing and implementing a complete distributed system solution.