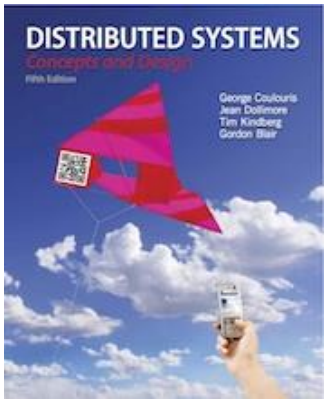
Slides for Chapter 21: Designing Distributed Systems: Google Case Study



From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems: Concepts and Design

Google Company

- **Google is a US-based corporation with its headquarter in Mountain View, CA. 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 in Internet today.**
- **Google was born out of a research project at Standford with the company launched in 1998.**

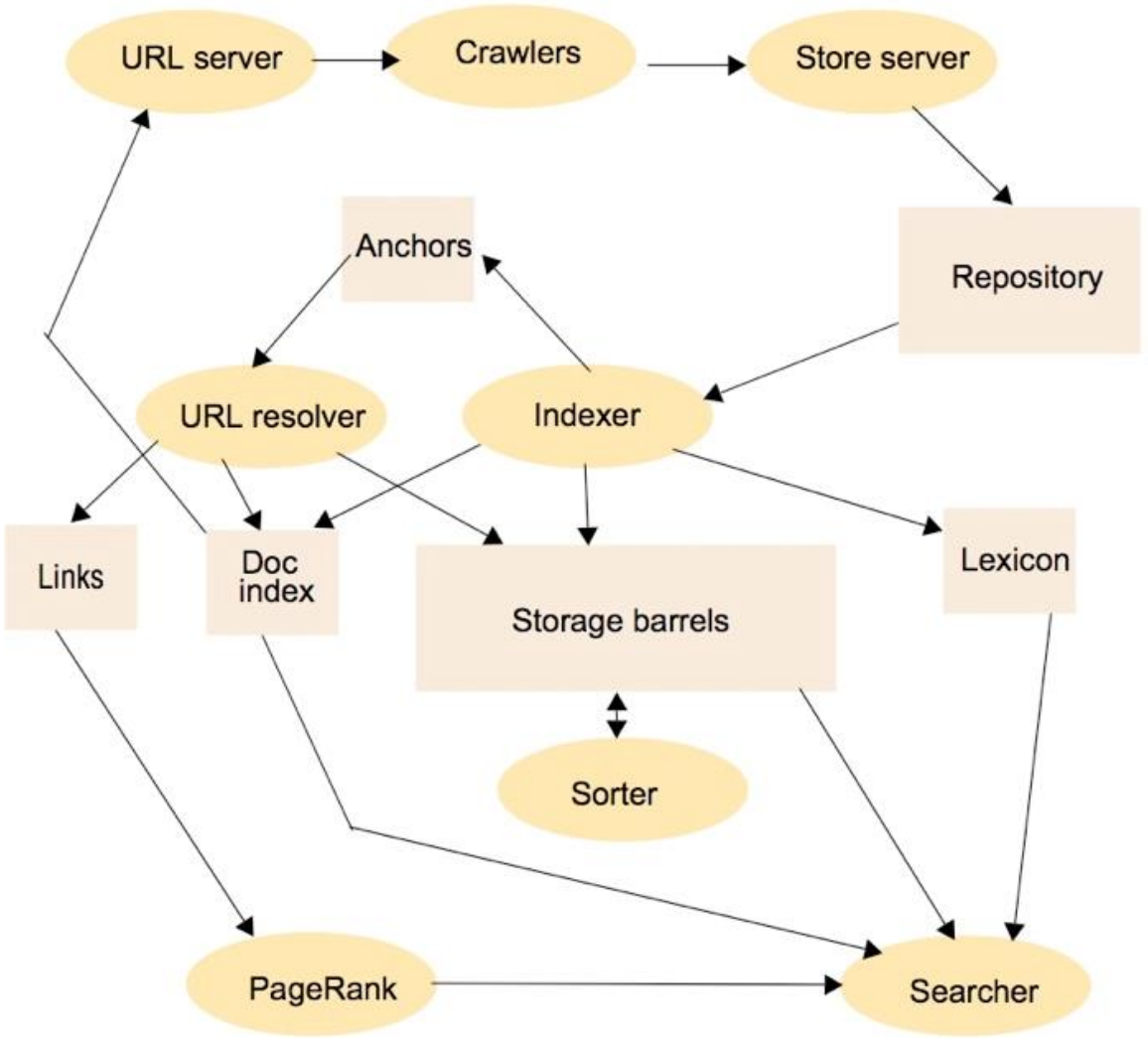
Google Distributed System: Design Strategy

- **Google has diversified and as well as providing a search engine is now a major player in cloud computing.**
- **88 billion queries a month by the end of 2010. The user can expect query result in 0.2 seconds.**
- **Good performance in terms of scalability, reliability, performance and openness.**
  - We will examine the strategies and design decision behind that success, and provide insight into design of complex distributed system.

Google Search Engine

- **Consist of a set of services**
- **Crawling: to locate and retrieve the contents of the web and pass the content onto the indexing subsystem. Performed by a software called Googlebot.**
- **Indexing: 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. Indexing produces what is known as an inverted index mapping words appearing in web pages and other textual web resources onto the position where they occur in documents. In addition, index of links is also maintained to keep track of links to a given site.**
- **Ranking: Relevance of the retrieved links. Ranking algorithm is called PageRank inspired by citation number for academic papers. A page will be viewed as important if it is linked to by a large number of other pages.**

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



Google as a cloud provider

- **Google is now a major player in cloud computing which is defined as “a set of Internet-based application, storage and computing services sufficient to support most user's needs, thus enabling them to largely or totally dispense with local data storage and application software.**
- **Software as a service: offering application-level software over the Internet as web application. A prime example is a set of web-based applications including Gmail, Google Docs, Google Talk and Google Calendar. Aims to replace traditional office suites. (more examples in the following table)**
- **Platform as a service: concerned with offering distributed system APIs and services across the Internet, with these APIs used to support the development and hosting of web applications. With the launch of Google App Engine, Google went beyond software as a service and now offers it distributed system infrastructure as as a cloud service. Other organizations to run their own web applications on the Google platform.**

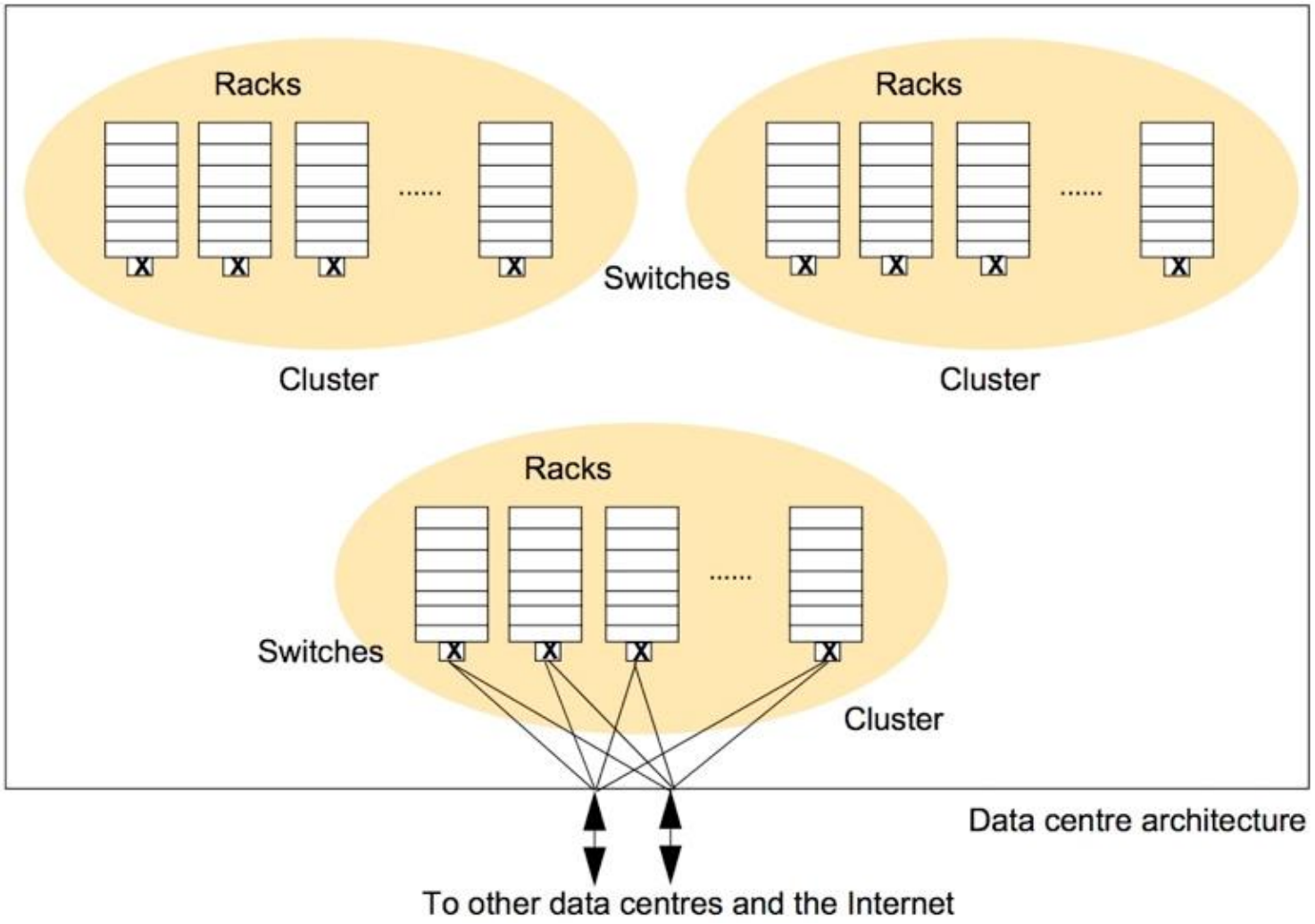
Figure 21.2
Example Google applications

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).

Google 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 decision are based on obtained the best performance per dollar rather than absolute performance. When Brin and Page built the first Google search engine from spare hardware scavenged from around the lab at Standford university.**
  - Typical spend is $1k per PC unit with 2 Terabytes of disk storage and 16 gigabytes of memory and run a cut-down version of Linux kernel.
  - Physical Architecture of Google is constructed as:
    - Commodity PCs are organized in racks with between 40 to 80 PCs in a given rack. Each rack has a Ethernet Switch.
    - 30 or more Racks are organized into a cluster, which are a key unit of management for placement and replication of services. Each cluster has two switched connected the outside world or other data centers.
    - Clusters are housed in data centers that spread around the world.

Figure 21.3: Physical model
Organization of the Google physical infrastructure



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

Key Requirements

- **Scalability: i). Deal with more data ii) deal with more queries and iii) seeking better results**
- **Reliability: There is a need to provide 24/7 availability. Google offers 99.9% service level agreement to paying customers of Google Apps covering Gmail, Google Calendar, Google Docs, Google sites and Google Talk. The well-reported outage of Gmail on Sept. 1st 2009 (100 minutes due to cascading problem of overloading servers) acts as reminder of challenges.**
- **Performance: Low latency of user interaction. Achieving the throughput to respond to all incoming requests while dealing with very large datasets over network.**
- **Openness: Core services and applications should be open to allow innovation and new applications.**

Figure 21.5
The overall Google systems architecture

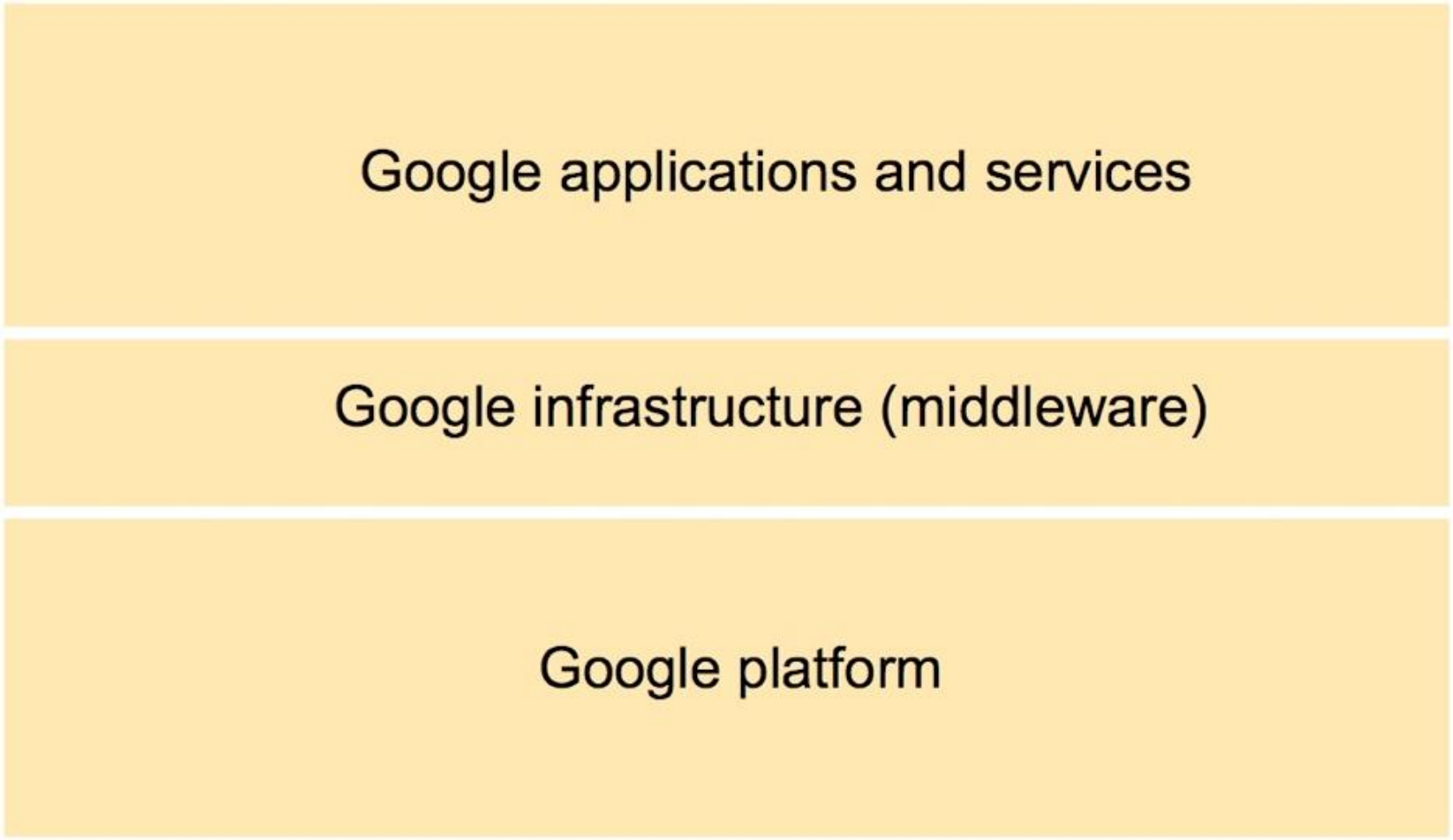
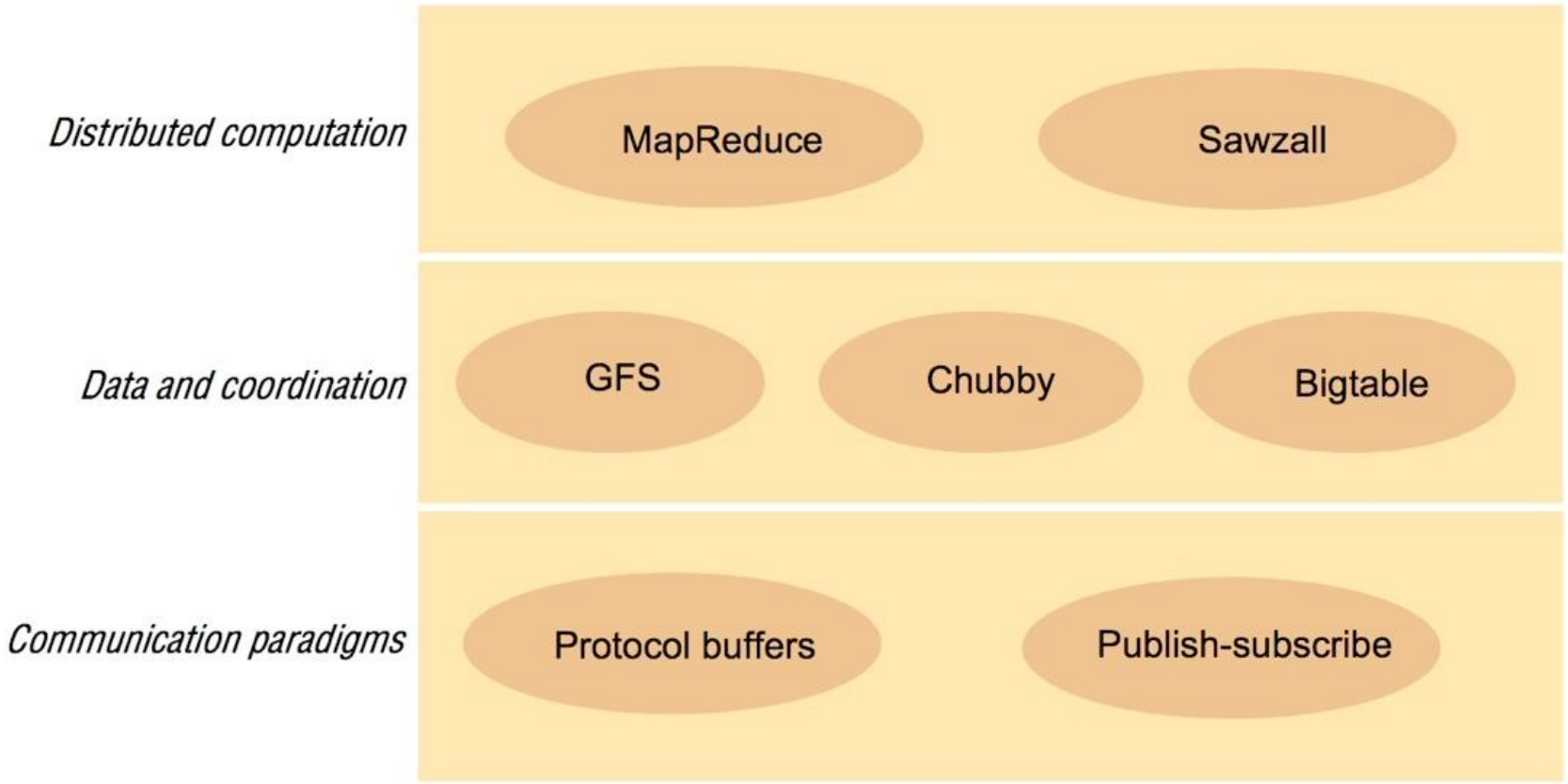


Figure 21.6
Google infrastructure



Google Infrastructure

- **The underlying communication paradigms, including services for both remote invocation and indirect communication.**
  - The protocol buffers offers a common serialization format including the serialization of requests and replies in remote invocation.
  - The publish-subscribe supports the efficient dissemination of events to large numbers of subscribers.
- **Data and coordination services providing unstructured and semi-structured abstractions for the storage of data coupled with services to support access to the data.**
- GFS offers a distributed file system optimized for Google application and services like large file storage.
- Chubby supports coordination services and the ability to store small volumes of data
- BigTable provides a distributed database offering access to semi-structure 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 computation.

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;
}
```

Figure 21.8a
Summary of design choices related to communication paradigms - *part 1*

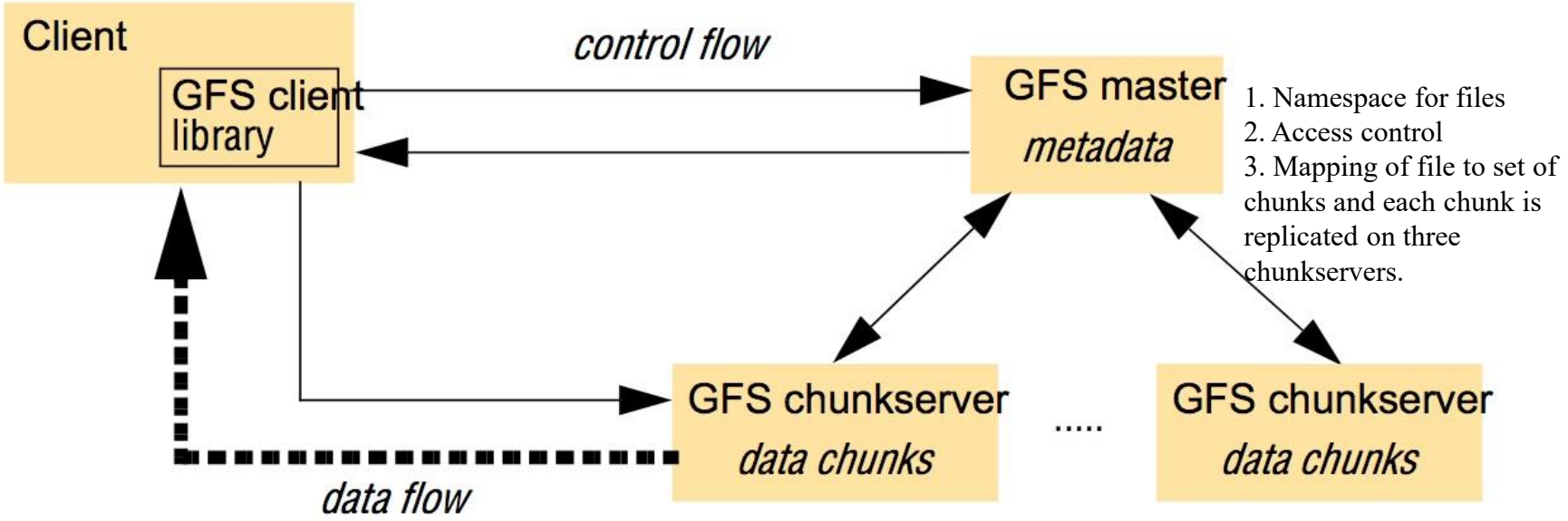
Element	*Design choice*	*Rationale*	*Trade-offs*
Protocol buffers	The use of a language for specifying data formats	Flexible in that the same language can be used for serializing data for storage or communication	-
	Simplicity of the language	Efficient implementation	Lack of expressiveness when compared, 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 implementations can be used	No common semantics for RPC exchanges

Figure 21.8b
Summary of design choices related to communication paradigms - *part 2*

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

Data Storage and Coordination Services

Figure 21.9 Overall architecture of GFS



NFS and AFS are general-purpose distributed file system offering file and directory abstraction. The GFS offers similar abstractions but is specialized for storage and access to very large quantities of data (not huge number of files but each file is massive 100Mega or 1Giga) and sequential reads and sequential write as opposed to random reads and writes. Must also run reliably in the face of any failure condition.

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

Four distinct capabilities:
1.Distribute locks to synchronize distributed activities in a large-scale asynchronous environment.
2.File system offering reliable storage of small files complementing the service offered by GFS.
3.Support the election of a primary in a set of replicas.
4.Used as a name service within Google.

It might appear to contradict the over design principle of simplicity doing one thing and doing it well. However, we will see that its heart is one core service that is offering a solution to distributed consensus and other facets emerge from this core service.

Figure 21.11
Overall architecture of Chubby

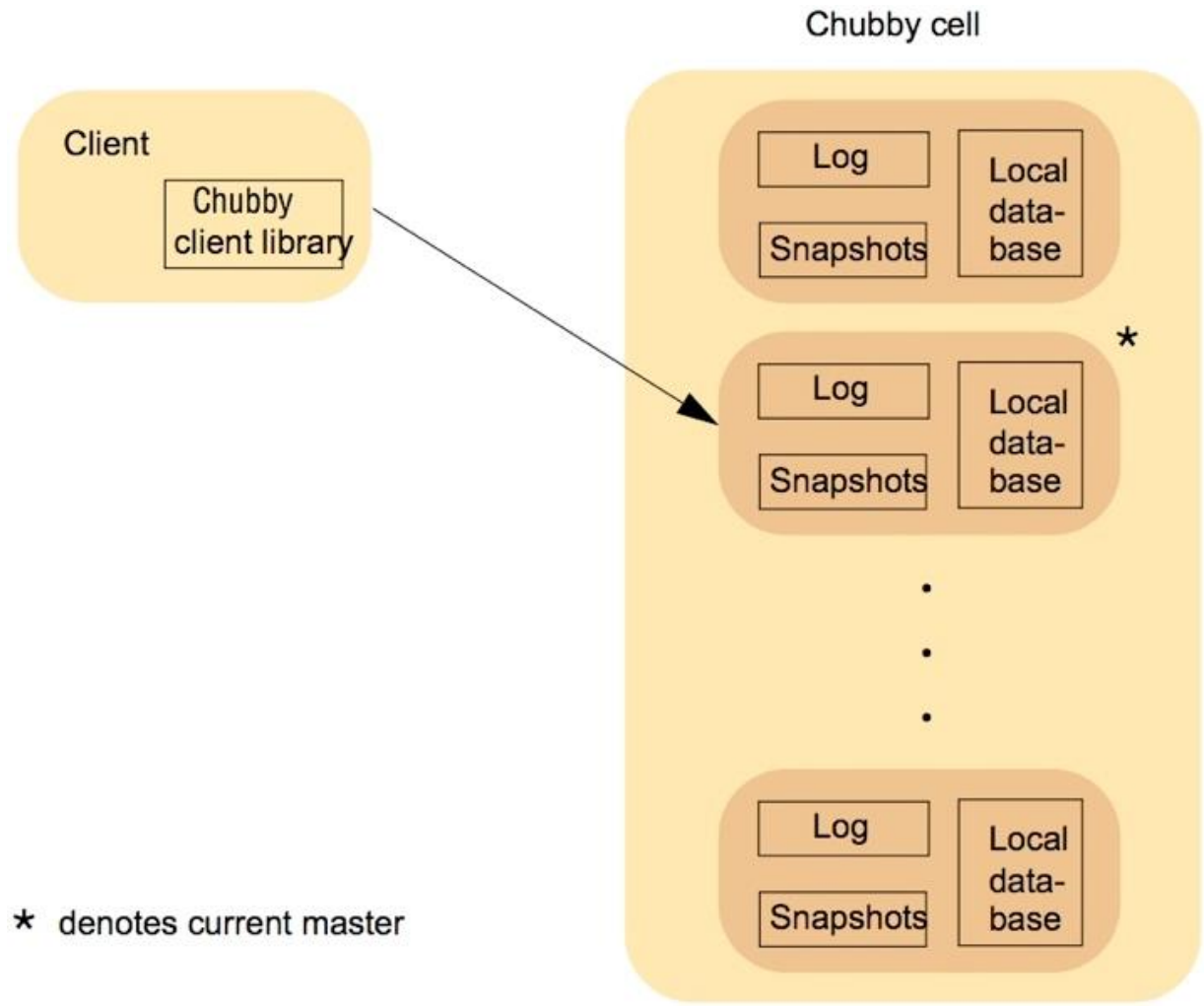


Figure 21.12
Message exchanges in Paxos (in absence of failures) - step 1

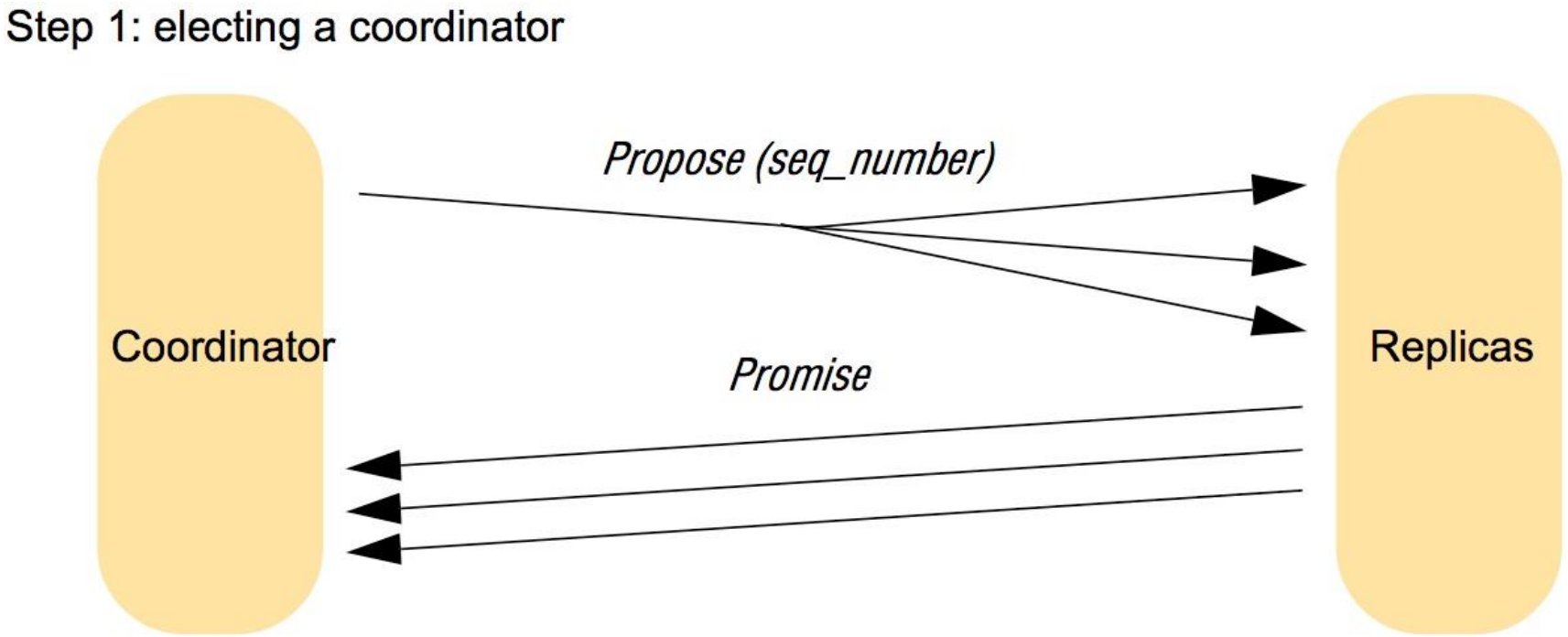


Figure 21.12
Message exchanges in Paxos (in absence of failures) - step 2

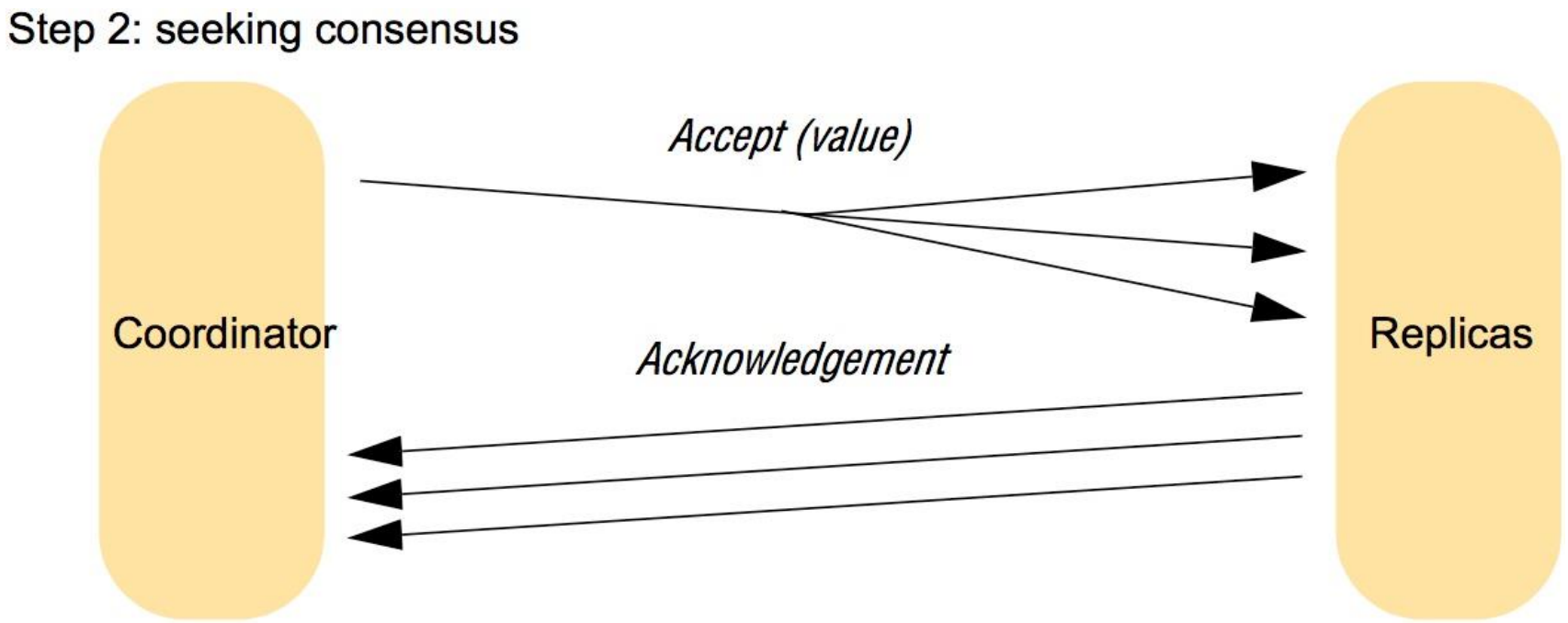


Figure 21.12
Message exchanges in Paxos (in absence of failures) - step 3

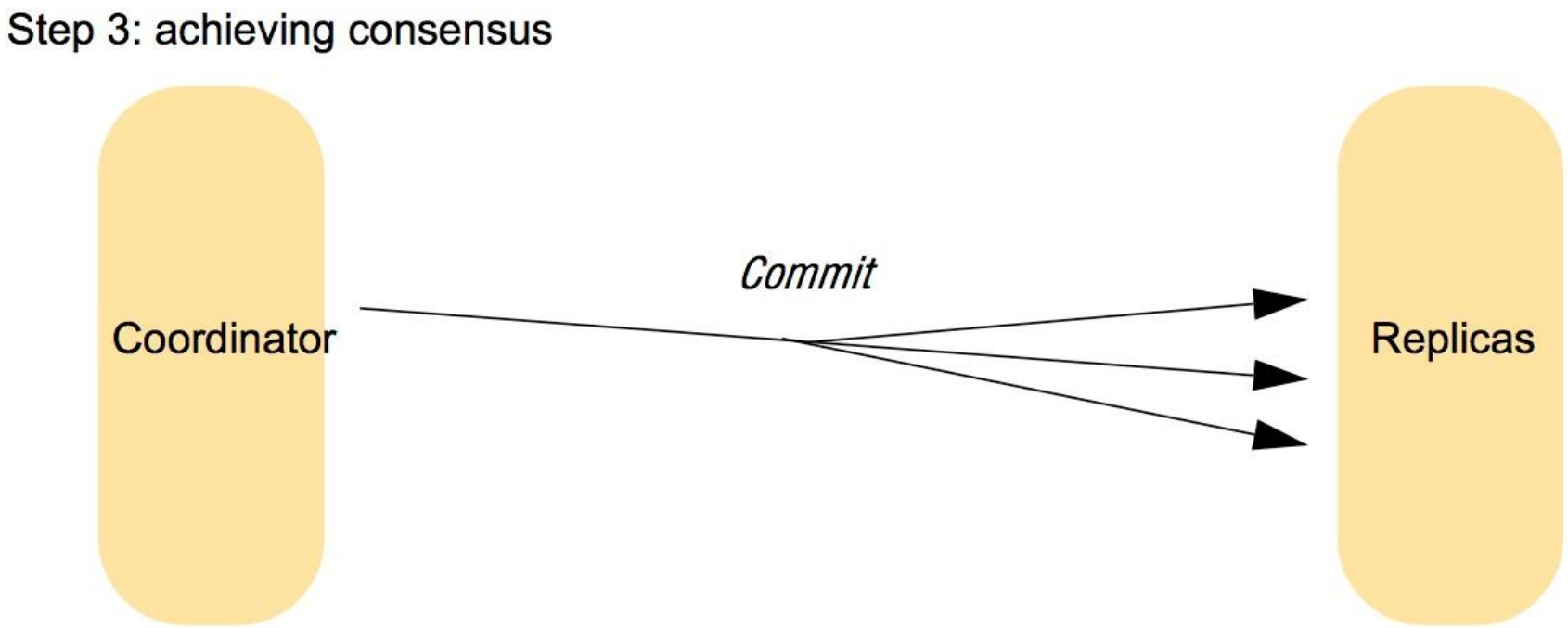
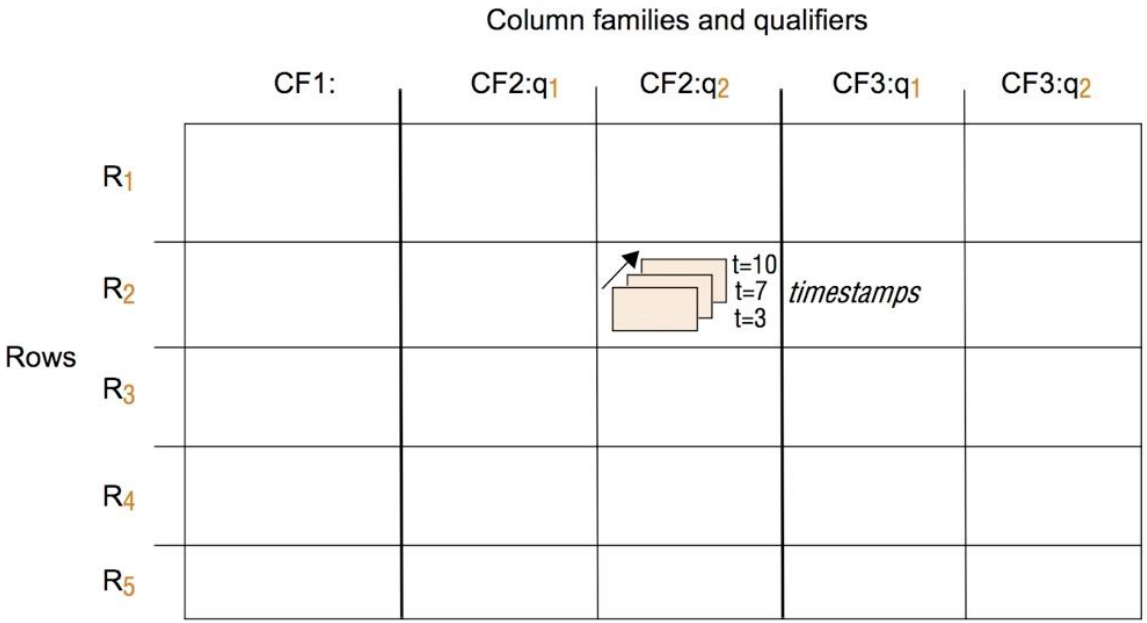


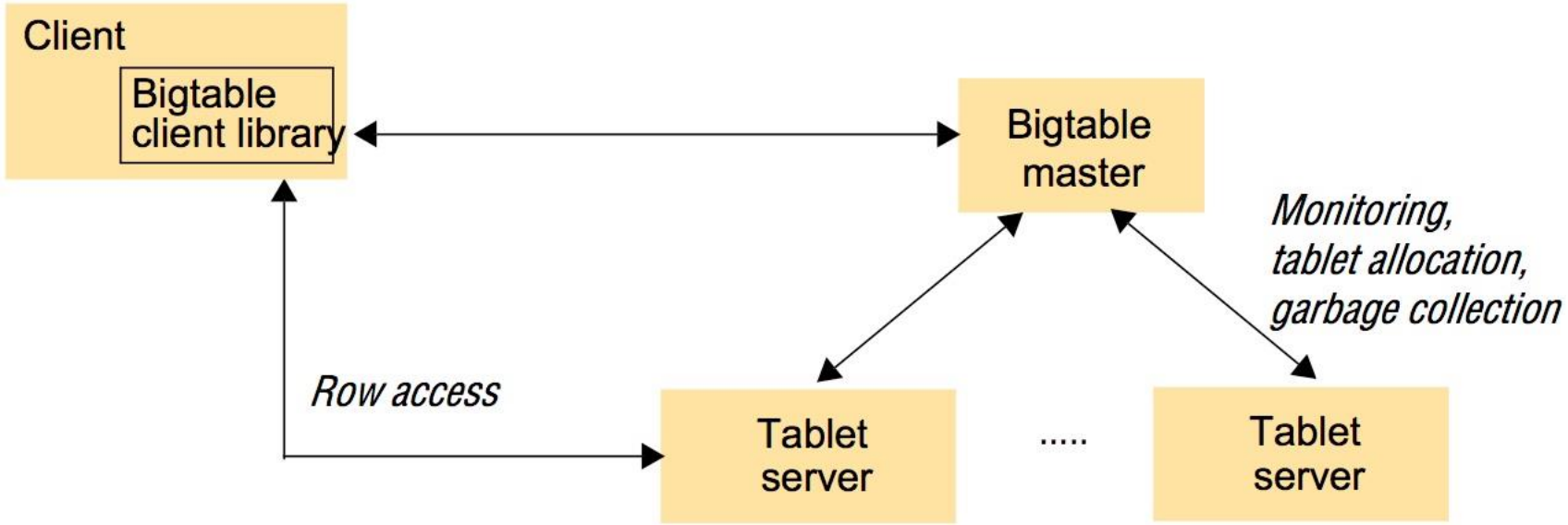
Figure 21.13
The table abstraction in Bigtable



- For example, web pages uses rows to represent individual web pages, and the columns to represent data and metadata associated with that given web page.
- For example, Google earth uses rows to represent geographical segments and columns to represent different images available for that segment.

- GFS offers storing and accessing large flat file which is accessed relative to byte offsets within a file. It is efficient to store large quantities of data and perform sequential read and write (append) operations. However, there is a strong need for a distributed storage system that provide access to data that is indexed in more sophisticated ways related to its content and structure.
- Instead of using an existing relational database with a full set of relational operators (union, selection, projection, intersection and join). However, the performance and scalability is a problem. So Google uses BigTable in 2008 which retains the table model but with a much simpler interface.
- Given table is a three-dimensional structure containing cells indexed by a row key, a column key and a timestamp to save multiple versions.

Figure 21.14
Overall architecture of Bigtable



- A Bigtable is broken up into tablets, with a given tablet being approximately 100 to 200 megabytes in size. It use both GFS and Chubby for data storage and distributed coordination.
- Three major components:
  - A library component on the client side
  - A master server
  - A potential large number of tablet servers

Figure 21.15
The storage architecture in Bigtable

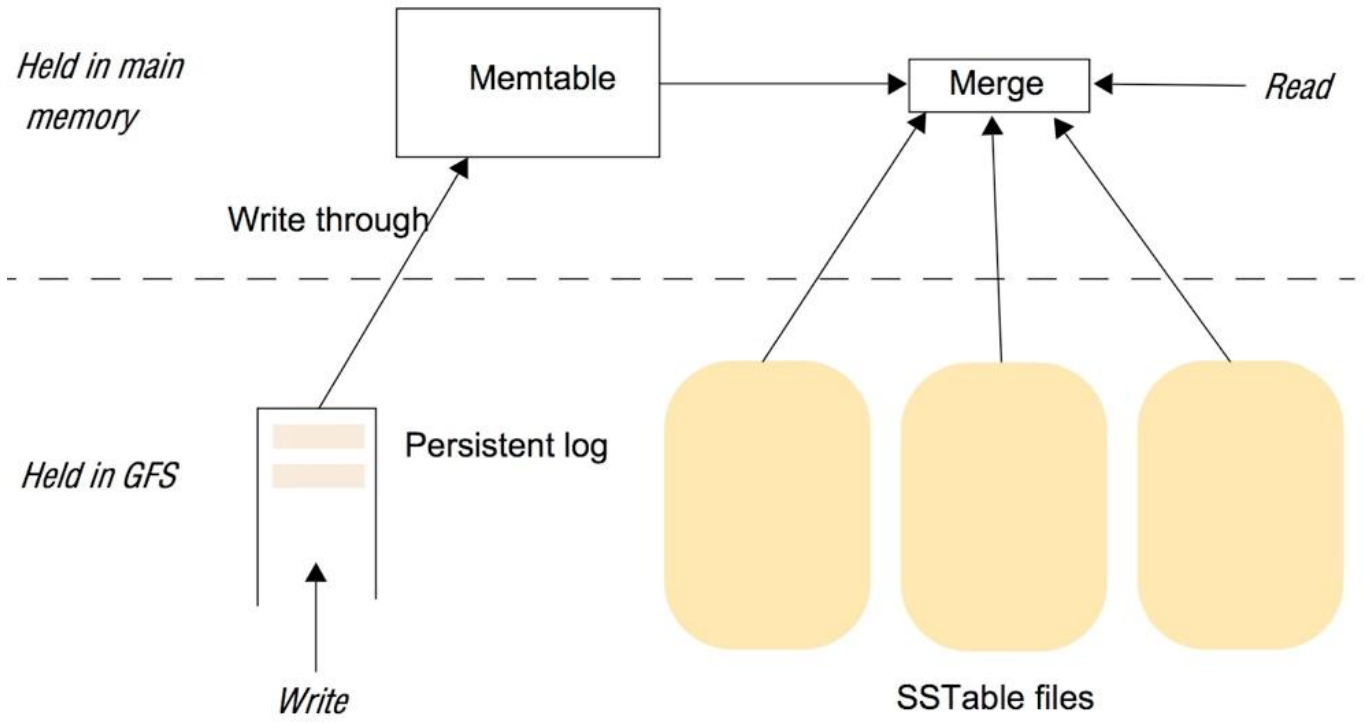
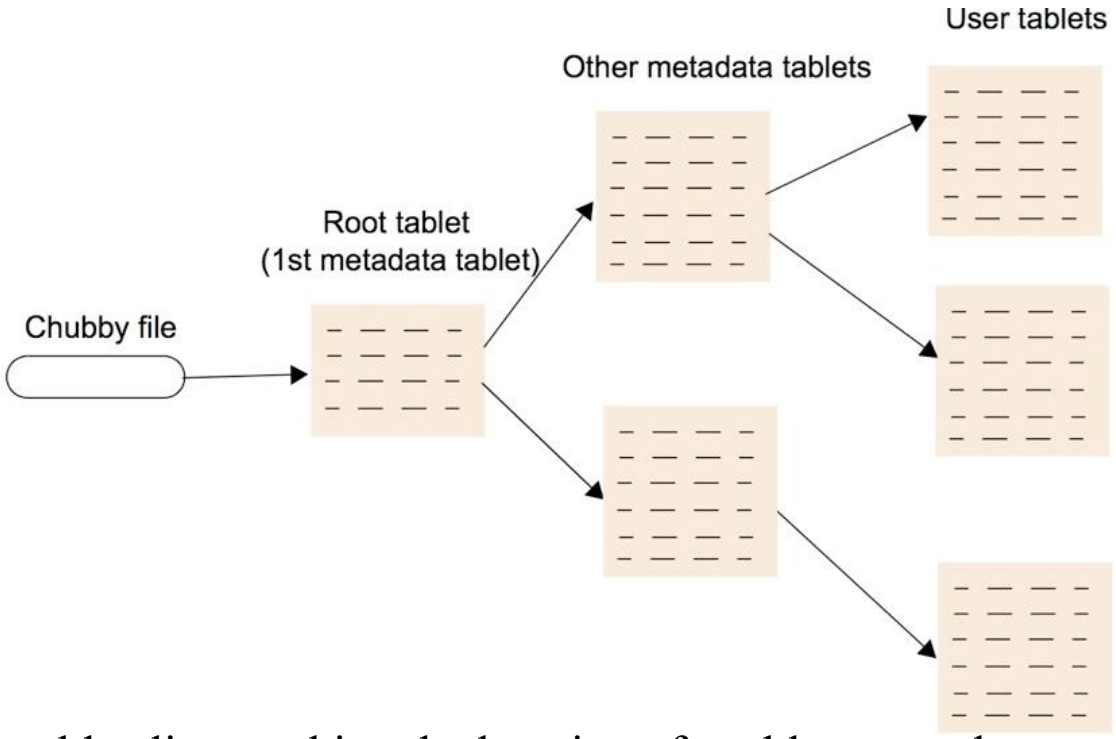


Figure 21.16
The hierarchical indexing scheme adopted by Bigtable



- 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 (containing the root index of the tree structure).
- The root contains metadata about other tablets specifically about other metadata tablets, which in turn contain the location of the actual data tablets.

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

Distributed Computation Services

- It is important to support high performance distributed computation over the large datasets stored in GFS and Bigtable. The Google infrastructure supports distributed computation through MapReduce service and also the higher level Sawzall language.
- Carry out distributed computation by breaking up the data into smaller fragments and carrying out analyses (sorting, searching and constructing inverted indexes) of such fragments in parallel, making use of the physical architecture.
- MapReduce {Dean and Ghemawat 2008} is a simple programming model to support the development of such application, 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.
    - Key principle behind MapReduce is 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 (map function)
        - Combine the intermediary results to produce the final output.(reduce function)

Distributed Computation Services: MapReduce

- For example, search web with words “distributed system book”:
  - Assume map and reduce function 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 the phrase followed by the name of the web document containing this phrase.
  - The reduce function is in this case is trivial, simply emitting the intermediary ressults ready to be collated together into a complete index.

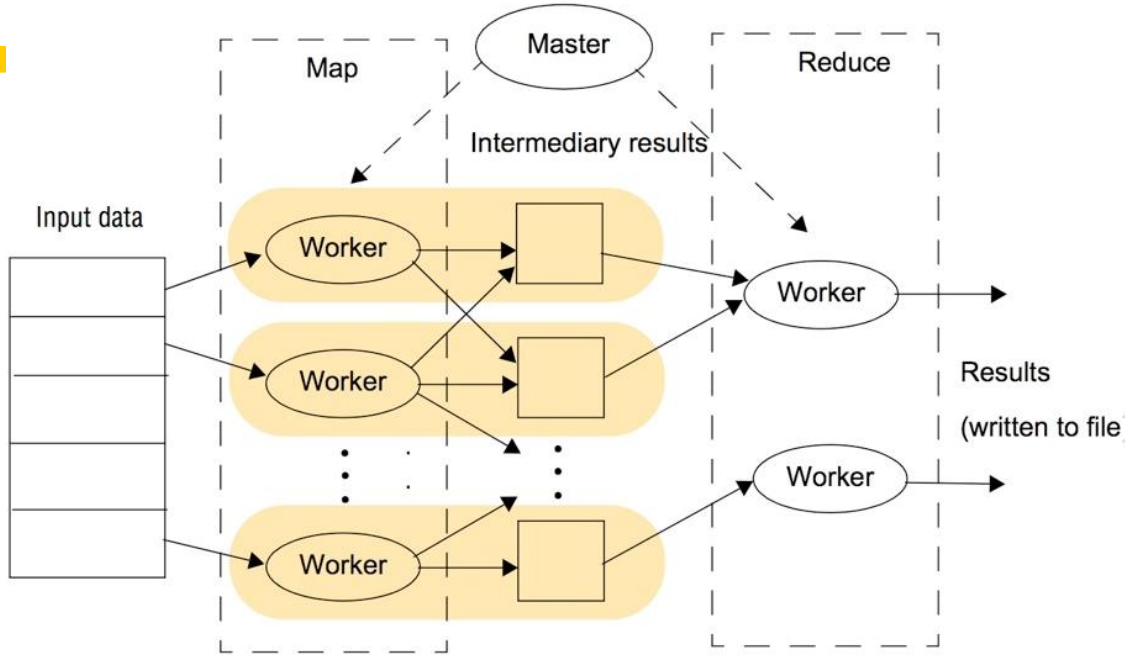
  - The MapReduce implementation is responsible for breaking the data into chunks, creating multiple instances of the map and reduce function, allocating and activating them on available machines in the physical infrastructure, monitoring the computations for any failures and implementing appropriate recovery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

  - 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.

Figure 21.18
Examples of the use of MapReduce

Function	*Initial step*	*Map phase*	*Intermediate step*	*Reduce phase*
Word count	*Partition data into fixed-size chunks for processing*	For each occurrence of word in data partition, emit <*word*, 1>	*Merge/sort all key-value keys according to their intermediary key*	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*		For each entry in the input data, output the key-value pairs to be sorted		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

Figure 21.19
The overall execution of a MapReduce program



- The first stage is to split the input file into M pieces, with each piece being typically 16-64 megabytes in size (no bigger than a single chunk in GFS). The intermediary results is also partitioned into R pieces. So M map and R reduce.
- The library then starts a set of worker machines from the pool available in the cluster with one being designed as the master and other being used for executing map or reduce steps.
- 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.
- The intermediary buffers are periodically written to a file local to the map computation. At this stage, the data are partitioned resulting in R regions. Unusually apply hash function to key then modulo R to the hashed value to produce R partitions.
- 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. The data will be sorted and reduce worker will step through key-value pairs in the partition applying the reduce function to produce an accumulated result set which will be written to an output file. This continues until all keys in the partition are processed.

Figure 21.20
The overall execution of a Sawzall program

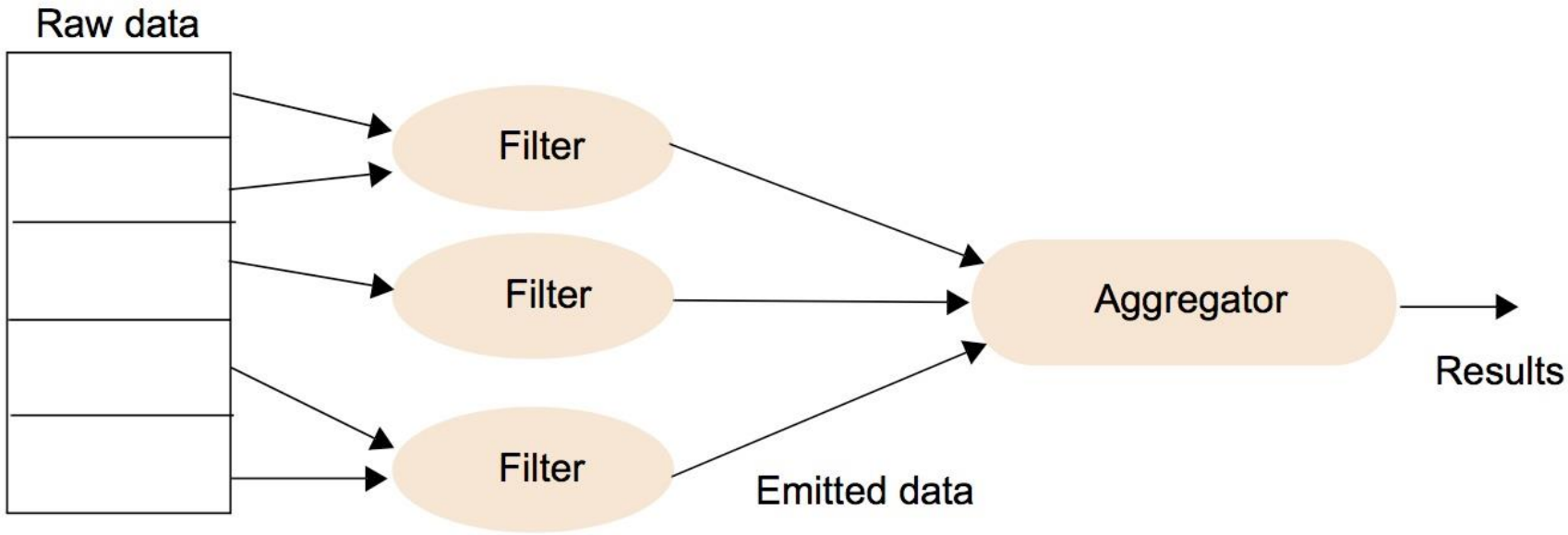


Figure 21.21
Summary of design choices related to distributed computation

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)