Distributed Systems: Concepts and Design

Edition 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair

Chapter 21 Exercise Solutions

21.1 To what extent is Google now a cloud provider company? Refer to the definition from Chapter 1, repeated in Section 21.2 above.

21.1 Ans.

The definition of cloud services in this book is as follows: "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". Looking at Google, the company offers a wide range of Internet-based services to the public from the initial search service through applications such as Gmail, Google Docs, Google Calendar, Google Earth, Google Maps, and the Google App Engine. All are implemented on their extensive physical infrastructure. Google also provides a range of services internally including GFS, Chubby, Bigtable and MapReduce. These are certainly all cloud services, running in the Internet and aiming to largely or totally dispense with local services, for example, Google Docs replacing word processing on local machines.

The services cover both software as a service, in the case of Gmail, Google Docs, Google Calendar, Google Earth and Google Maps from the list above and platform as a service, in the case of the Google App Engine. The internal services are also an example of a private cloud offering platform as a service facilities.

In conclusion, Google can now be viewed as an extensive provided of cloud services offering both software and platform as a service, but not yet operating in the domain of infrastructure as a service.

21.2 The key requirements for the Google infrastructure are scalability, reliability, performance and openness. Provide three example of where these requirements might be in conflict and discuss how Google deals with these potential conflicts.

21.2 Ans.

The most obvious conflict from the list above is between reliability and performance, as introducing measures to increase reliability will inevitably carry an overhead. For example, replication is often used to increase reliability (and also is used extensively in Google) and this does require additional algorithms to maintain consistency of replicas. The key is to implement reliability in a manner that minimizes the impact on performance, for example to have relaxed consistency between replicas and this strategy can be seen in GFS, for example (where they also implement policies optimized for the style traffic they encounter). Given that GFS also underpins Bigtable and MapReduce, this can have significant impact on the overall performance of the architecture. In addition, the separation between control and data in GFS and Bigtable allows control to be implemented in masters, where optimal placement strategies can be implemented, then the master gets out of the way in terms of delivering good throughput in terms of data access.

There is also a similar potential conflict between reliability and scalability. The above strategies also help with scalability by reducing the overhead of consistency management. Equally, Google makes use of the extensive physical infrastructure to ensure that services can scale.

There is also a potential conflict between openness and performance. In particular, the temptation in supporting openness is to provide generic (and hence potentially bloated) interfaces that are suitable for a wide range of applications and services. This trend has certainly been seen in a number of middleware implementations such as CORBA. Google's philosophy is however to offer minimal interfaces that do one job very well. Perhaps the best example is the underlying communications mechanism, protocol buffers, that, in

comparison to other communication paradigms, arguably lacks sophistication but provides a lightweight, fast and sufficiently general purpose service to meet their communication needs (along with the alternative publish-subscribe service which again is quite minimal).

21.3 A specification of a *Person* structure in XML was presented in Chapter 4 (Figure 4.12). Rewrite this specification using protocol buffers.

21.3 Ans.

The *Person* structure in protocol buffers would simply look like:

```
message Person {
    optional string id = 1;
    required string name = 2;
    required string place = 3;
    required int32 year = 4;
}
```

The *id* field could be left out as, strictly speaking, it is not an intrinsic part of the person structure (as in the CORDA CDR version for example). Note that this specification is not exactly equivalent to the XML version as it is not self-describing. It is possible to include such descriptions in protocol buffers as described in Section 21.4.1.

21.4 Discuss the extent to which the RPC style supported by protocol buffers enhances extensibility (especially the design decision to have one argument and one result).

21.4 Ans.

The key argument for having a single argument and a single result is that interfaces defined in terms of operations which take data in and produce data out are more likely to be stable than more general interfaces defined in RMI, for example, which will have arbitrary parameters including object references). This also shares similarities with the REST philosophy as discussed in Section 9.2, where the complexity is pushed towards the data and interfaces are simple. Both features are potentially supportive of more extensible interfaces (although this is debated in the community).

21.5 Explain why the Google infrastructure supports three separate data storage facilities. Why does Google not just adopt a commercial distributed database instead of utilizing the three separate services?

21.5 Ans.

As mentioned above in the answer to Exercise 21.2, the Google philosophy is to provide minimal services that do one task efficiently. Google therefore provide three minimal services all doing their own job well, which together provide coverage of their storage needs:

- GFS offers storage of very large datasets optimized for the styles of reads and writes encountered in their environment;
- Chubby offers efficient access to very small data items, together with the services for distributed consensus;
- Bigtable supports access to semi-structured data (building on top of GFS and Chubby).

Google would not want to use a commercial distributed database for three reasons:

- they prefer to own and be responsible for their entire code base;
- distributed databases are not tailored towards their own operating environment, for example dealing with massive datasets;
- distributed databases, for example relational databases, offered rich functionality that goes beyond what is necessary in the Google environment (for example, full relational operators) and this also comes at a price in terms of performance.

21.6 Both GFS and Bigtable make the same core design choice – to have a single master. What are the repercussions of a failure of this single master in each case?

21.6 Ans.

This decision was made, particularly in GFS, to simplify the design process and allow Google to get a production system in operation quickly. Although this is a single point of failure, failures can be detected and the master restored. This is supported by the fact that key metadata is stored in a persistent operations log and this is replicated on several machines, enabling relatively rapid recovery. Other data can be restored by polling the chunkservers. There will be a short delay from failure occurring to being detected through to the master being restored and this will disable operations that require the master, but not ongoing transfers as they bypass the master due to the separation of control and data (note that in Bigtable, fewer operations go through the master - see Section 21.5.3).

21.7 In Section 21.5.2, we compared the cache consistency approach in Chubby with NFS, concluding that NFS offers much weaker semantics in terms of seeing different versions of files on different nodes. Perform a similar comparison between the cache consistency approaches adopted in Chubby and AFS.

21.7 Ans.

In Chubby, clients cache data and, when a mutation occurs, this operation is blocked until all other cached entries are invalidated. Caches are also not updated directly but rather updates go through the server. This ensures deterministic results, that is clients either see valid data, or receive an error. In AFS, when a server receives a request to update a file, a callback is sent to every node with a cached copy and this invalidates this copy. On a subsequent open, a client checks the cached entry and, if marked invalid, it must retrieve a new copy from the server. These two schemes are very similar and both offer more deterministic guarantees than NFS. The key difference is that in AFS a callback message may be lost and this can result in clients accessing an old copy of the file.

21.8 As described in Section 21.5.2, the implementation of Paxos depends on the generation of increasing and globally unique sequence numbers. This section also described a possible implementation. Describe an alternative approach to implementing such sequence numbers.

21.8 Ans.

The solution in Section 21.5.2 is simple, efficient and works. Other solutions are equally possible. What is required is that the sequence numbers are unique and monotonically increasing. Another way of achieving this is to form a sequence number from a combination of a timestamp and an identifier uniquely identifying the associated node.

21.9 Figure 21.18 lists a number of possible applications of MapReduce. Describe one other possible application and sketch out how this would be implemented in MapReduce, providing in particular outline implementations of the *map* and *reduce* functions.

21.9 Ans.

There are a large number of possible answer to this question, but we provide one example, that is summing the elements of a large array.

The MapReduce framework will take this large array and split it into chunks passing the chunks on to parallel instances of the map operation.

This map operation will take its portion of the array and sum the elements in this portion, emitting the intermediary result. This result is a key-value pair. In this case, the key is not significant and will (say) be given the same value across all instances of map, say 1.

The reduce function then takes all intermediary results and produces the sum of all the values as the final result. In this case, only one instance of the reduce operation is required, seeking all intermediary results with the given key (1), and calculating the sum.

21.10 Provide an example of a distributed computation that would be difficult to implement in MapReduce, giving full reasons for your answer.

21.10 Ans.

MapReduce is designed to work well with applications that fit the pattern of taking large datasets, performing the same operation (the map operation) on different chunks of this data, producing a set of intermediary results, sorting these intermediary results (by intermediary key), then combining these intermediary results into a single value or values, again by carrying out a number of parallel executions of the same operation (the reduce operation). This is a flexible paradigm that matches many styles of distributed computation, but not all.

Consider for example a computation that follows a pipeline pattern where the results of one computation feed directly into another and so on, along a pipeline (for example as in the class Sieve of Eratosthenes algorithm). This is fundamentally a different programming pattern that would not work well in MapReduce. There are many other examples of distributed computation that do not fall into the pattern supported by MapReduce.