Questions for Midterm #1

Jeff Chase
CPS 512, Fall 2016
Duke University
Department of Computer Science

October 1, 2016

Some of you have asked about how to study for the midterm. So far this semester we have covered key aspects of the state of the art in technology for building big services (reliable, scalable, available, etc.). At this point we have covered the basics of large-scale services at a level that is broad but not deep. We have studied a number of techniques, protocols, systems, and properties to illustrate the principles:

- Reliable RPC in Birrell/Nelson, classic NFS and RIFL/RAMCloud;
- Lease-based cache consistency in NFSv4 and Chubby;
- Leased sharding in GFS and NFSv4/pNFS;
- Single-shot transactions in Birrell/Wobber;
- Client/server transactions as used in Thor and RAMCloud, with 2PL (or OCC) concurrency control and 2PC for distributed commit;
- Primary/backup replication as used in Chubby and RAMCloud, based on RSM Consensus (Paxos, VR, RAFT);
- Coordination service abstractions as implemented in Chubby and Zookeeper, and their use;
- Consistency and availability properties in relation to these various techniques.

That’s a lot to absorb. At this point you should exercise your ability to think about the various techniques and protocols in a holistic/integrated way that focuses on how they fit together.

Putting it all together

Consider an application service on a cluster of servers in a datacenter. What techniques do we use to make it scalable and reliable (available)? What does the application code have to do differently to use these techniques, vs. say, a service implementation that can run on only a single server? How effective are these techniques and what are their costs? How should we quantify these terms “scalable”, “available”, and “cost” to make these questions concrete? What metrics do we use?
To organize your thinking for a deep drive, review the sequence of slides about the architecture of your “next big megaservice”, based on a classic multi-tier structure. See Figure 1. This structure mirrors the “ship in a bottle” distributed service of the “rings” Scala/Akka code and the group service you built for lab #1. But here we have tricked it out with other techniques from the state of the art, e.g., transactions. RAMCloud/RIFL/RAFT is an example of a state-of-the-art approach to datacenters of the future that incorporate all of these elements.

Below are some questions based on this architecture. These are a little more open-ended than I would use on an exam, but if you can puzzle through them all and work through the various interactions and tradeoffs you should be in pretty good shape. You can look at the relevant slides for each question and reflect on how the issues discussed in the slides bear on the answer.

For questions like the ones below, the answer might be “it depends”. In that case, your answer should say what it depends on. In other words: if you need to make additional assumptions to answer a question, just go ahead and make them and state them. You should strive to develop a simple mental model of real-world systems that helps you reason about them. The assumptions that you make show me what your mental model is.

Consider an arbitrary application service \( A \) built with this architecture. Here are some simple but reasonable assumptions you could make about \( A \):

- \( A \) is a three-tier service with presentation, application, and storage tiers. Incoming requests from end-users are distributed randomly among the front-end (presentation-tier) servers.
- Each user request for \( A \) results in a single call to the application tier. A single server running the application code for \( A \) can handle up to \( T \)
requests per second, on average, for a typical request stream on A: the peak throughput of a single server is $T$.

- Service A stores its state as (key, value) pairs in a K/V store. The K/V storage servers can handle up to $S$ requests per second, on average, for typical value sizes and a typical requests. The processing for a request to the application tier accesses $k$ keys chosen at random on average.

- Servers have a probability $p$ of being down at any time. The uptime is then $1 - p$. For example, suppose each server fails randomly once every 1000 time units, and takes one time unit to recover, on average. Then $p = .001$, and servers for A have 99.9% uptime.

These assumptions might not be true in practice. Also, there are lots of factors they don’t take into account, for which more assumptions might be needed to analyze the structure. But they make a good starting point.

**Sharding and caching**

Suppose first that the storage tier uses sharding but no replication, and no transactions, and that requests are distributed randomly across servers in the application tier.

- What does it mean for the service A to be “scalable”? If A is “ideally scalable”, what is the peak request throughput on $N$ servers?

- What impact does sharding have on throughput? Does dividing the (key, value) pairs evenly among the storage servers make the storage tier “scalable”? Why or why not?

- What impact does caching have on the service? Suppose the (key, value) accesses are 100% reads, and each application server keeps an in-memory cache of (key, value) pairs that it has read. What is the hit rate in the caches? What other assumptions do you need to answer this question?

- How do these caches affect the throughput of the storage tier? How do they affect the throughput of the application tier? What impact do they have on the number of servers required to handle the incoming request load at some given rate?

- What is the impact of the caches on the latency or response time of end-user requests? How would you characterize the response time to answer this question? What other assumptions do you need to answer this question?
Consistent caches

Now suppose that the operations on the key-value store are 10% writes and only 90% reads. Suppose further that the application tier caches are kept consistent using reader/writer leases on individual keys.

- What impact does this have on cache hit rate? What other assumptions do you need to answer this question?
- What if an application server fails? Would such a failure be visible to users? How should the service design hide such failures from users?
- What if the storage tier API supports transactions (like RAMCloud/RIFL), and the application tier uses them. Explain how transactions could hide such failures from users.
- You should understand how transactions work together with sharding and caching in a system like this. Why is a two-phase commit needed (2PC)? Who acts as the coordinator for 2PC? When are writes propagated to the storage servers?
- We can presume that the system invalidates cached copies of a key’s value when a write for a key commits. Assume that any running transaction that has read or written the key from a stale cached copy is restarted. How do transactions impact the effectiveness of the caches? The throughput of user requests? The latency or response time of end-user requests?

Application-tier sharding

Now suppose further that the requests originating from any given user access shards selected from some fixed subset of the service data, say 10% of the data: each user is “interested” in only 10% of the data.

- How would you leverage this property to improve the cache hit rates? Propose a design.
- How does a front-tier (presentation) server choose the application server for a request in your design?
- What impact does your design have on hit rate, service throughput, etc.?

Replication

Now consider replication. Why do we want it? How much does it help? What are its costs?
• What is the impact of sharding on availability? Consider the storage tier under the assumptions above. What is the impact on availability if a storage server fails? What is the probability that a shard needed for some given client request is unavailable?

• Suppose that each data shard is replicated on three storage servers, using Consensus replication. What is the probability that a shard needed for some given client request is unavailable? How about with a replication degree of five?

• What is the impact of this replication on the throughput of the storage tier? What is its impact on throughput of the application tier? What is its impact on the number of servers required to handle the incoming request load at some given rate?

• What is the impact of this replication on the latency of end-user requests?

• With this replication, what should an application server do if an RPC to a storage server times out? Outline the steps it might take to resolve the issue.

• Explain how fail-over could cause two different servers in the storage tier to each receive the same request \( R \), or each issue a reply to \( R \). Explain how to preserve execute-exactly-once-eventually semantics in this situation (e.g., using mechanisms from RIFL).

• With all of these assumptions, how would your characterize the behavior of application service \( A \) with respect to consistency and availability (CAP)? Explain why it cannot be 100% available. What could go wrong? Under what conditions would it deny service to a user?

We also discussed consistency and the relationship with availability. We have defined a notion of an atomic data object—an “object” (e.g., a server/service/module) that exposes an API and ensures linearizability of all calls to that API.

Here are some questions about consistency issues with respect to our model of the three-tier application service \( A \):

• What is the relationship between linearizability and transactional serializability? Does the serializability of transactions at the storage tier guarantee linearizability of operations on (key, value) pairs sent to the storage tier? Is linearizability at the storage tier a sufficient condition for serializability of transactions? Is it a necessary condition?

• Are read-write leases sufficient to ensure linearizability of operations on (key, value) pairs at the client API for the K/V service (in the client library)? Are they necessary?
What effect does Chubby’s session lease mechanism have on the number of lease renewal requests in an implementation of a consistent cache using read-write leases?

How does Chubby control the overhead of the session-lease mechanism under load? What impact does this have on availability?

Could the service A as a whole be viewed as an atomic object? Does it or could it meet the linearizability requirement for an atomic object at the user API?

We talked about how sharding complicates the implementation of consistent (e.g., linearizable) services with operations that involve multiple shards. Consider the problem of the rename operation in the file abstraction (NFS); rename is absent from the key-value storage abstraction and from Chubby, both of which are designed to be scalable/reliable. Do transactions solve this problem? Is transaction serializability at the data tier sufficient to ensure linearizability of operations at the application tier? Is it necessary?