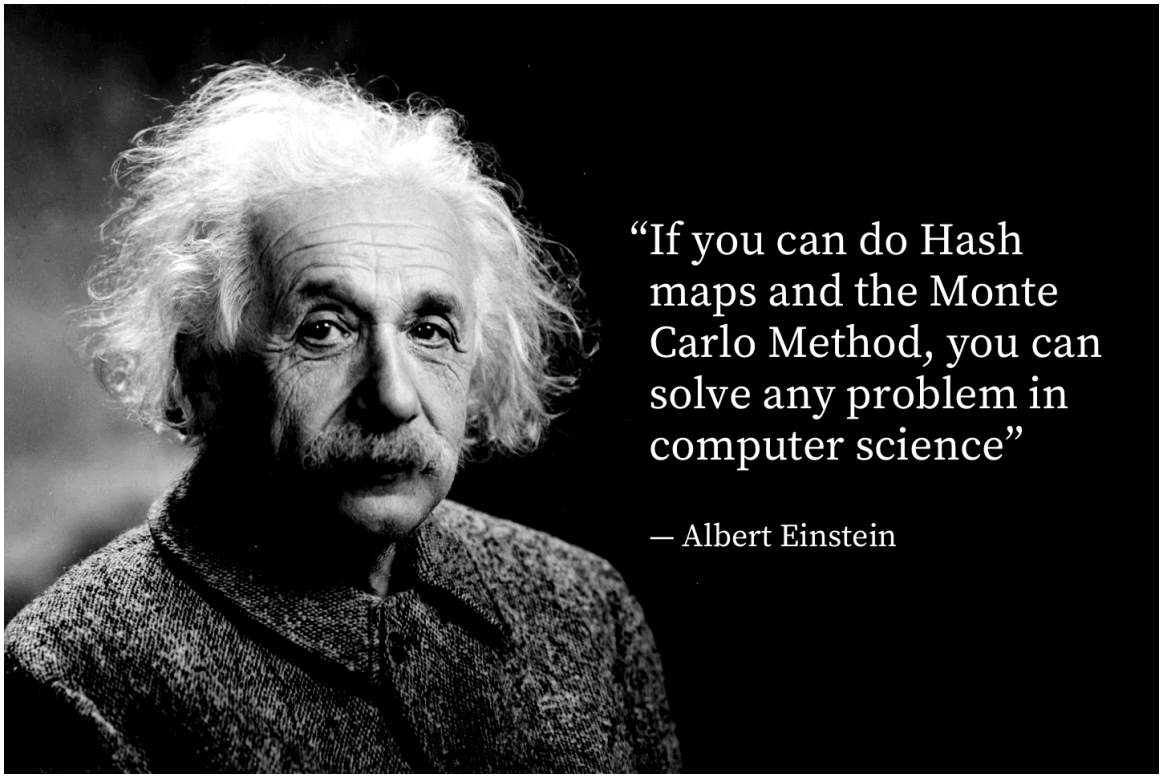
## About Part 4

In this section, we employ a depth-first approach to teach you how to design systems. The systems described in Part 4 are as simple as possible in order to maximize the practical knowledge you can take with you. The inspiration for this section is the famous quote, "If you can do Hash maps and the Monte Carlo Method, you can solve any problem in computer science."




OK, fine, you caught us—Einstein didn't actually say this. However, this quote does come from a competitive programmer who made it to top 24 in the world, so that's got to count for something!

This section takes the idea behind that quote even farther. If you understand the power of immutability, deduplication, enumeration, and how to get yourself unstuck, you can understand how to design any system. Part 4 is a master class in which you'll get to observe the decision-making process of senior engineers as they build systems. Along the way, we'll teach you additional tips and tricks to help you excel during an interview. Once you've finished reading Part 4, you'll possess the knowledge you

need to truly understand system design interviews. And you'll see there was never any reason to be scared of designing systems in the first place.

## Chapter One

Let's get started. The best way to learn is to build, so we'll begin by designing something that is:

- Large enough, and
- Does not have many interdependencies.

**For something large enough, let's look at TikTok and simplify it greatly. Here's one way to do it:**

### 1. Change from videos to pictures.

Now we're designing Instagram instead of TikTok. And, system design-wise, the change is not dramatic.

### 2. What other big element could we put aside for now?

- Social features. Let's drop reactions, comments, content recommendations, chat, etc. For now, users would just be able to upload pictures, group them into albums, and share pictures and/or albums via links.
- By this point we've reduced our TikTok to some early version of Flickr.

### 3. Maybe pictures are too much?

- Simplifying further, we can replace pictures with files. After all, pictures are just large-ish static files, served by some globally distributed cache, maybe a CDN.
- We are now looking at a greatly simplified version of Dropbox.

### 4. Let's take it even further, all the way through TinyURL to Pastebin. How so?

- Taking it to the extreme, and considering our files to be text only, we get TinyURL or Pastebin.

- Logically, TinyURL and Pastebin are almost the same thing. Map short keys into somewhat longer textual blobs. “To redirect or not to redirect” is a minor detail on the product side; it doesn’t affect the design much. After all, the logic of storing the destinations of shortened URLs is more or less the same if the users get to view those URLs, not get redirected to them automatically.

**Let’s now look into Pastebin. It’s probably the easiest one of all the above.**

To keep things simple, imagine we don’t need to edit Pastes; we just need to remove them sometimes.

**This is important. Why?**

Some objects are immutable: once created, they always represent the same value. Other objects are mutable: they have methods that change the value of the object.” (Link to the MIT professor’s mutability/immutability lesson this quote is from.)

## Rule of thumb

System Design problems are almost always easier in the immutable case (compared to the mutable one).

**Here is one way to get unstuck during a system design interview: consider the immutable case. This is a practical way to dumb the problem down. Tackle the dumber problem, and then add in complexity after that.** Considering the immutable case also helps with identifying bottlenecks, as well as with capacity planning.

## How to Get Yourself Unstuck – Tip #1:

To simplify the problem, think about the data in the problem as immutable. You can choose to add mutability back in later after you have an initial working design.

If editing pastes—or changing destinations for shortened URLs—is allowed, there is a whole spectrum of problems to tackle:

- **Concurrency:** What if two people are updating the same data at the same time?
- **Consistency:** How quickly should the updates propagate? In other words, for how long are the users in Australia "allowed" to see the now-outdated content after a user in America has updated it?
- **Reconciliation:** What if the heat in Australia kills drives in a data center there? Or a solar flare occurs and neutrinos fry databases in Sydney? What if there was a network split, and some remote database replicas did not receive an update? We can, and should, implement our "best effort" approach during outages, but it's important to end up in a consistent state—every user across the globe seeing the same data—once the situation stabilizes.

All these questions have answers, but each of these answers is more difficult than solving the simplified problem end-to-end. It's best to step over these parts for this introductory problem.

## Requirements

It helps to align on requirements before designing the system. A conversation with the interviewer flows better if functional and non-functional requirements are introduced.

## Rule of thumb

Functional requirements (FRs) tend to map 1:1 to the functionality the service can expose to the outer world.

## Functional Requirements

- A user can add a Paste: send in the text, get back a short, unique URL, which is its ID now.
- Another user can view a Paste knowing its ID / URL.
- The original author of a Paste can delete it.

- The original author should be able to see the lists of the Pastes they have created (candidates often miss this requirement).
- There may be metadata associated with Pastes (i.e., the programming language, for highlighting / indentation purposes).
- We are deliberately not allowing editing pastes for now, otherwise editing would be here.
- Another big topic to skip for now is analytics, such as how many people viewed a paste, from where, etc. (For the record, this part of the system is generally heavier and more complex compared to the core functionality.)
- Last but not least: monetization. Some users would want to pay to be able to create more Pastes and/or create larger Pastes, and/or receive advanced insights on the popularity of their Pastes.
- If we were to be pedantic, there may be legal requirements to abide by. For example, tracking users who are requested by a court order to be tracked, and flagging / restricting / removing content that should not be there for one reason or another.

## Non-Functional Requirements

- **Availability**: The service should be up and running, ideally, 24/7. We'll get to the "number of nines" later; for now it's just important to mention that the Pastes should be available.
- **Consistency**: Pastes should not change or disappear. They should only stay, or be deleted, by their original owner or by the platform. (Later in Part 4, we will address consistency in the cases of split brain, DB replicas failing, etc.)
- **Durability**: If part of the system and/or the cloud provider and/or the Internet is down, reasonable effort should be put into preserving the contents of the Pastes. (Simply put, we just need to agree that it's not a good idea to store all

Pastes on one drive of one machine, and/or on a single-shard DB and/or in a single geographical location.)

- **Scalability**: Our design should not break under heavy load. (I.e., if our product is suddenly a bit too popular worldwide, we should be able to handle the dramatically increased load more or less well, but drop dead until significant manual intervention takes place.)
- **Latency**: Retrieving a Paste by its ID, as well as other operations, should be relatively fast. (Sub-second would be a decent first approximation; whether we want to get to the 250ms vs. 50ms vs. 5ms here is an open question that we defer for now.)

### Rule of thumb

Similar to DS&A questions, it's easy to make assumptions about the problem. Think of the requirements stage as a fluid conversation between yourself and the interviewer. While you will drive the conversation, you should always check in with them to see if you've missed anything. Many people forget that they can literally ask the interviewer if they've missed any important requirements.

## How to Get Yourself Unstuck – Tip #2:

Not sure if you got all the requirements? Think you're missing something important, but you don't know what? Turn your requirements gathering into a conversation and get the interviewer involved. Ask your interviewer: "Are there any important requirements you have in mind that I've overlooked?" This is totally allowed!

### Estimates

While not strictly necessary, it's worth it to be able to answer questions about the scale of a service in under a minute. This is known in the industry as the "back of the envelope" calculations of load estimates.

### What's the point of doing estimates at all in these problems?

In DS&A questions, we have a DS&A problem as an input, and we output an answer with a target time and space complexity. Your solution is judged on the amount of time and space it takes.

**Equivalently, in system design questions, we have a design question as an input, and we output a working design that uses a specified amount of CPU and will take a specified amount of RAM/Disk storage.** Your solution is judged on the amount of CPU & Disk ("compute and storage") it takes, but there is also emphasis on technical depth and communication as well.

The main reason to do estimates is to justify building a large-scale distributed system rather than just hosting everything on a single computer in your Uncle Louie's basement. Your estimate will prove that it's necessary for you to build a complex distributed system. Keep in mind that some interviewers will want to see a fault-tolerant, resilient design, even if you are able to produce and justify the Uncle Louie solution. You're interviewing for a big company, after all!

Both in an interview and in real-life design, it's useful to be able to estimate orders of magnitude, especially when it comes to users, data volumes, or costs.

An important note: There's no "right" way to do these estimates, but there are "wrong" ways. So, in practice, this is one of the aspects of the conversation where it's more important to focus on being "not wrong" rather than on being "right."

**A back-of-the-envelope estimate here could be along the following lines.**

- The upper bound on the number of developers on the planet is ~1% of the working adult population. So ~1% of ~50% of ~8B, which is roughly 40M users.
- The average number of Pastes per user is far less than one—say, 0.5. While many users would have many Pastes, most would have none. So, before our Pastebin takes off, designing for some ~20M Pastes makes sense.

- We can (and should!) limit Pastes in size. 1 MB sounds reasonable, although most would be much smaller.
- We should also accept that our usage would be far from uniform. Some hackers (in a good sense of this word!) would use Pastebin from their build / deploy scripts, so we should be prepared for usage spikes.

Estimates often can (and should!) be validated against real-life numbers, which are frequently public. Test your estimation skills here:

- https://expandedramblings.com/index.php/pastebin-statistics-and-facts/
- https://en.wikipedia.org/wiki/Pastebin.com

### How to Get Yourself Unstuck – Tip #3:

Some interviewers hate this step and really don't want to see you stumble through 5th-grade math calculations for 15 minutes. Similar to step 2, ask your interviewer if they'd like to see some calculations before jumping in and starting them—you might be able to skip these entirely if the interviewer doesn't care about it! With quite a few system design interviews, you'd be fine as long as you do mention that the system you plan to present will be durable, resilient, and scalable.

### How to Get Yourself Unstuck – Tip #4:

Sometimes it's difficult to know what to calculate during this part of the interview. If you've already confirmed that the interviewer wants to see calculations as mentioned in Tip #3, then follow these rough guides to get the basic estimates for any system.

### Storage Estimation:

Storage = daily data used by 1 user * DAU count * length of time to store data

## Bandwidth Estimation:

Bandwidth per second = (daily data used by 1 user * DAU count ) / total seconds in a day

Also, there are roughly 100K seconds in a day, which is five orders of magnitude. If your API gateway expects to see a billion requests on a busy day, that's approximately 10K requests per second, as 9 zeroes minus 5 zeroes is 4 zeroes. The true figure is ~15% larger, as 100K / (60 * 60 * 24) is around 1.15.

## Let's start designing!

Intuitive things first:

1. We need to figure out how to store Pastes. It might occur to the astute reader that all we're doing is sharing files (aka the "Pastes") between users. So if we are just sharing files, why not just have the "pasting" action simply copy the file with a "Paste" to a remote folder?
2. Folder sharing properties are a little niche and OS specific. Instead of dealing with them for each OS, perhaps we can access a folder from a tiny Python script?
3. Pastes always have those funky unique names in their URL. How do we generate those names for Pastes?

The shared folders analogy, from (1), immediately takes us about half way to designing a document store: a non-relational object persistence layer. It's broadly acceptable to refer to it these days as a NoSQL database, even though along many axes it is nearly the exact opposite. It would not support transactions or joins, for example. On the other hand, for the purposes of designing Pastebin, neither joins nor transactions are required. Plus, we want the benefits of pagination. This allows us to leverage the strong sides of document stores / NoSQL databases, while not

suffering from any downsides of them. The strong sides include features such as replication, automatic failover, and out-of-the-box horizontal scaling.

### Remember

The takeaway message here is that document stores truly are not very different from reliable shared folders, optimized to be used by software through APIs calls (2), not by humans via a command line or via drag and drop. They also abstract away the underlying OS details (as well as cloud / containerization / etc.), allowing their user to focus on the business logic of their application instead.

The unique names generation task, (3), is a well-known problem by itself. It is sometimes referred to as KGS, the Key Generation Service. We will tackle this problem later in Part 4; for now it will suffice to say that as long as the keys can be long enough, it is straightforward to reduce the risks of collision to zero, or to an infinitesimally small number.

### More Real Interview Stuff

The previous paragraphs outlined the key concepts that will help you reason about the problem. Depending on the interviewer, different parts of the problem might become more or less important with respect to getting an excellent score.

In this section we will cover some of those parts, from first principles, and loosely in the order of likelihood that you'll be asked about them.

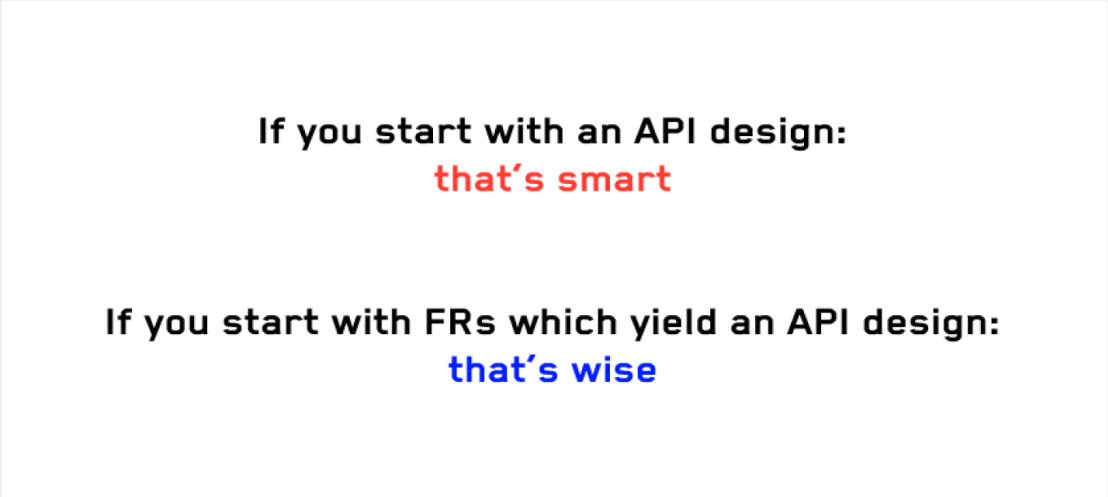
### API

For instance, one of the earlier identified functional requirements for Pastebin was that "another user can view a Paste knowing its ID / URL." Therefore, the API should have a GET endpoint similar to getPaste(pasteID)

## Rule of thumb

If the actions covered by functional requirements are generally one-off (such as “no streaming”) and affect a single instance of data (such as “no batching”), a RESTful API is usually a good choice.

We, the designers of a system, rejoice and perform a ritual dance in our heads, as we discover that each functional requirement yields an action that can be viewed as a request-response pair, so that each individual request-response pair accesses a single logical data point. This is a great use case for a RESTful API, which eliminates a large chunk of what needs to be thought of. This is doubly true in interviews.




## When not to use a REST API

### The cases where you wouldn’t want to use a REST API are:

- When either real-time responses (aka push notifications) or streaming are required. This is common in messaging or task queue problems.
- Or when one request can conflict with many others, which is a common occurrence in hotel booking or Ticketmaster problems, where a large number of users could be competing for the same resource.

REST is a large topic that deserves its own discussion, but right now the important thing to know is that in a RESTful API the server exposes Resources, and their list, as well as their Properties, can be viewed and changed. With Pastebin, the resources

are the Pastes, and the standard CRUD (Create + Read + Update + Delete) is a good way to begin reasoning about them. The problem is far simpler in the immutable case, of course, as we only need to create, read, and delete.

If multiple data points can be affected by one request, some batching is in order. This is not hard, and this can be generally worked around by introducing a shared entity that stores "batches" on the server side, which allows the client to interface with a single "batch," while the server manipulates multiple data entries at the same time.

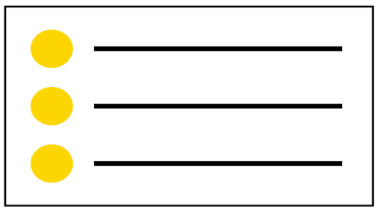
And the problem becomes far more difficult—making the RESTful approach to the API suboptimal—when real-time push updates to the client are essential. For example, when designing a messenger, the client should see a new message sent to them quickly enough, and the idea of polling the server (i.e., repeatedly asking some "are we there yet?" question) is worth taking a closer look at. We will tackle both batching and real-time push updates later in Part 4.

### An important part of the API that people tend to miss has to do with bulk GETs

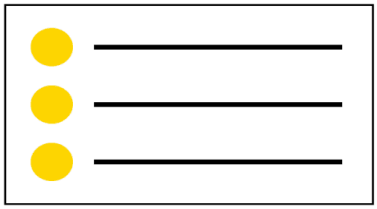
"Bulk GETS" means: retrieving a list of multiple objects. This may require pagination, on the API level, as the list may be long. Product- and user-level thinking are key here.

For the case of Pastebin, it's probably natural to retrieve Pastes sorted in reverse chronological order, which makes it simple to design a relatively straightforward pagination technique: "return the most recent N=20 elements, among the ones that are created no before T." T can be unspecified for the first call; when looking for the next page, the creation time of the last entry seen on the current page becomes the next T.

Pagination 1/2



Get Top 3



Get “Next” 3



New entry added between 1st & 2nd cells

Fail

## Pagination 2/2

Page 1 is sorted by date, which we have labeled as "d"

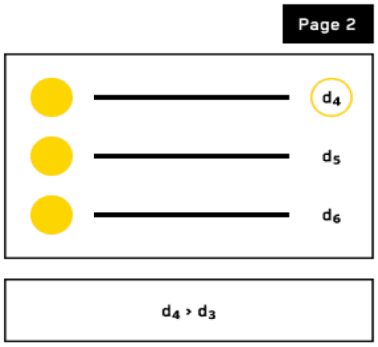

---

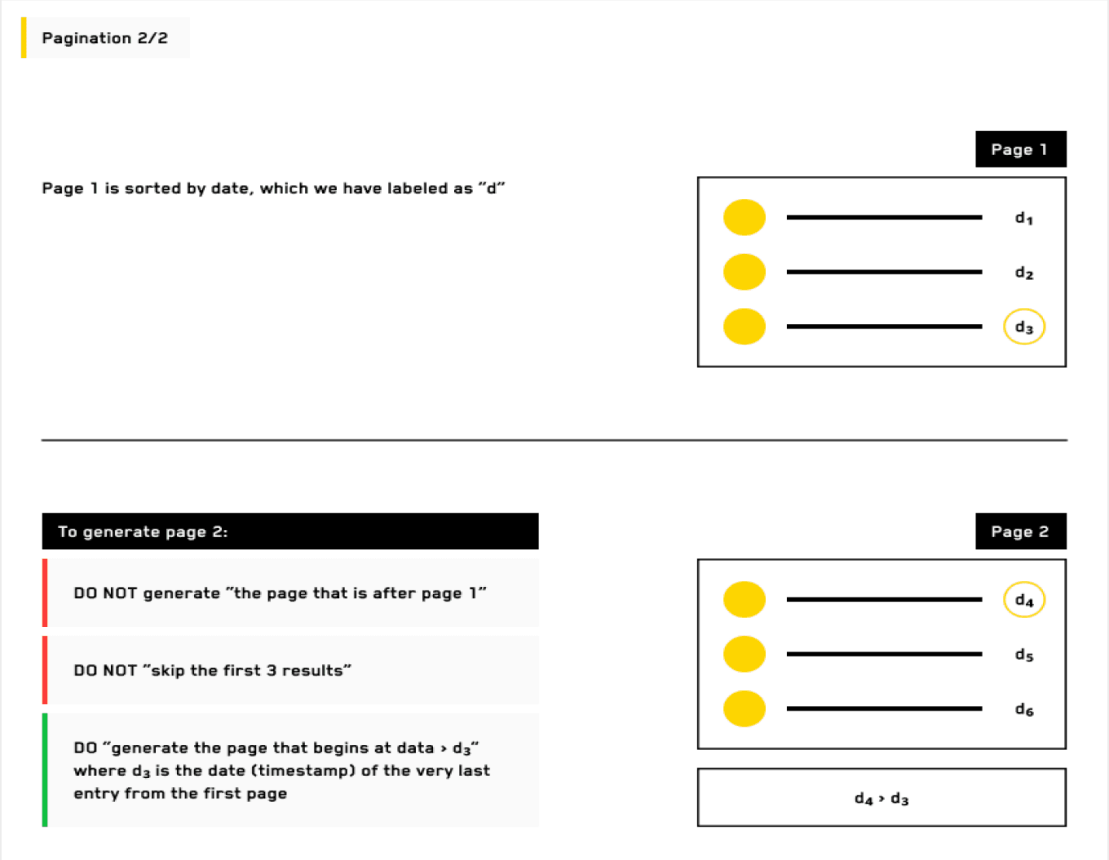
**To generate page 2:**

DO NOT generate "the page that is after page 1"

DO NOT "skip the first 3 results"

DO "generate the page that begins at data > $d_3$" where $d_3$ is the date (timestamp) of the very last entry from the first page






## Rule of thumb

The above design may be less "obvious" than assigning indexes; however, it is guaranteed to be consistent. And since we've determined consistency to be an important requirement for the way we're designing Pastebin, this is a tradeoff we will happily make.

## What are we not covering, but could be asked about for this Pastebin problem?

- Caching. How to make stuff load faster? How to load our servers less?
- How critical is deletion? How important is it to remove everything quickly (say, it's private data, by mistake)?

- Private sharing? How to authenticate? How to authorize? Is it possible to change sharing settings of a paste on the fly?
- Sharing analytics? / Telemetry.
- GDPR and other regulations to observe.
- Authentication. By the way, ****If you are asked an Authentication question in the context of the Pastebin problem, congratulations. This means you have likely nailed everything else the interviewer had to ask you about.

## Chapter Two

Time to look into mutability.

## Remember

As we talked about in Chapter One, most system design problems are trivial when reduced to an immutable case.

**A good way to dive into the depths of system design is to attack a problem that is all about mutability.**

In other words, we need to look into a problem where the objects can be changed. The perfect problem to illustrate this point would focus on mutability and nothing else. In search of such a problem, let's cover two warm-up ones first.

### Warm-up Problem One: Deduplication

The problem statement for deduplication is deceptively simple:

- The only way to call the service is: "Did you see X before?"
- The service answers with is true / false.
- The first time the service is called for X, it returns false.
- If the service is called with X again, it should return true.

Without loss of generality (WLOG), let's assume the X-es are strings of some reasonable length, say, under two hundred bytes. (If you want to learn more about the math-y term "WLOG," you can do that here.)

## Why is this important?

The short answer is entropy.

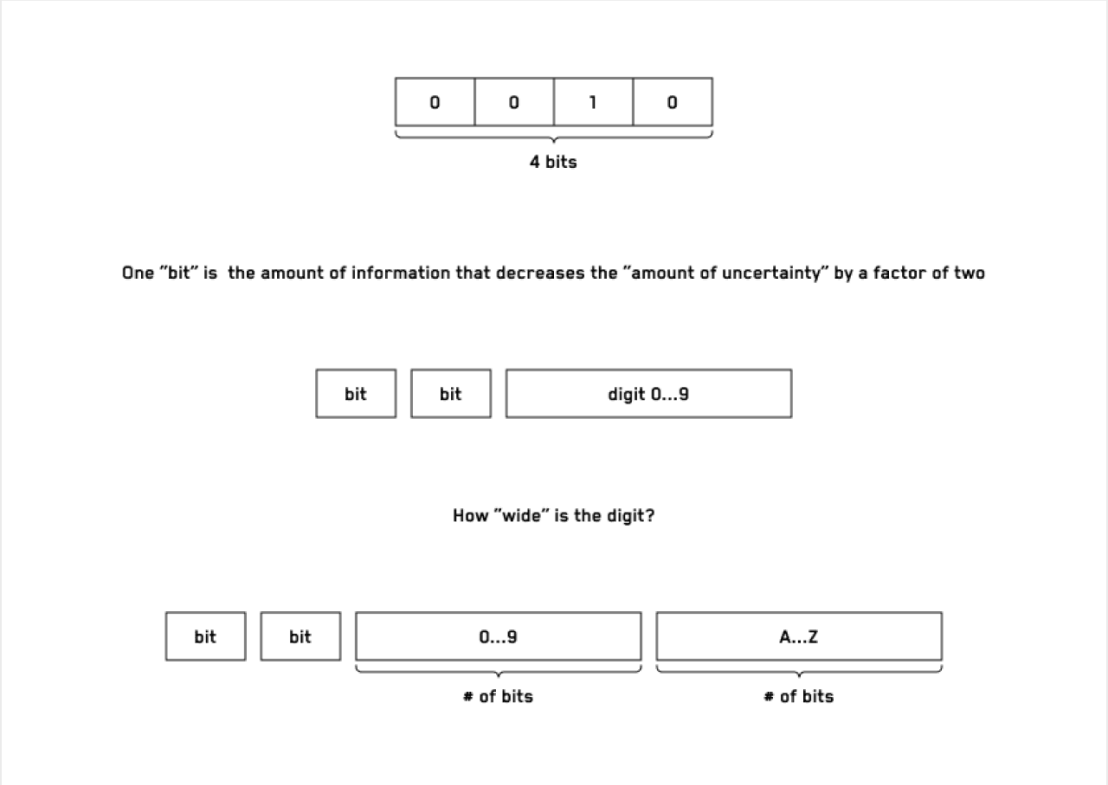
For most practical use cases, 128-bit numbers would be sufficient as well, as the probability of a hash collision within such a massive key space is negligible.

128 bits are 16 bytes. If we limit ourselves to Latin characters and digits, reducing each "character" from 1 bit (2 options), to log_2(36) bits (36 options, 26 characters + 10 digits), we will see that ~24.76 (i.e., 25) alphanumeric characters carry about the same amount of information as 128 "full" bits do.

### We can go a bit deeper to explain this point about characters and digits and bits.

- The idea is that the "amount of information" can be viewed as the width of the bar.
- One "bit" is defined as the amount of information that reduces uncertainty by a factor of two. Which is 0 or 1.
- Now, how much information is there in a digit from 0 to 9, of which there are 10? Well, one bit can differentiate between two options, two bits can differentiate between four, three bits between eight, and four between sixteen. So a decimal digit contains between three and four bits, and the exact answer is the binary log of 10, log_2(10), or log(10)/log(2).
- Similarly it's log_2(26) for a Latin letter, or log_2(36) for an alphanumeric, if we ignore the case.
- Now, with the "width of the bar" analogy, it's just about splitting the full "128 bits wide" bar into slices of log_2(36) units wide; that's approximately 24.76 units,

which we round up to 25, and conclude that a string or twenty-five random alphanumeric characters, such as AB23X-56QR1-22BFF-98KJH-GR8GG, contains about 128 bits of data.




The solution would not change much if we allowed our strings to be a kilobyte or two larger. If we go into megabytes, though, storing raw input blobs would become remarkably ineffective, and a solution that involves hashing the inputs manually, behind the scenes, would be a better one. (NOTE: you can read more about blobs here.)

In computer science terms (or, rather, in "algorithms and data structures" terms), the problem of deduplication is effectively about building a largely distributed append-only hash set.

## Remember

While system design interviews are not the same as algorithms and data structures ones, never forget that, ultimately, it's all about writing code that works! The

difference is this: doing vs. directing. In a coding interview, you're literally writing the code. In a system design interview, your goal is to assist those writing the code by providing architecture / direction / guidance.

### Let's put together some requirements first.

Keep in mind, **we work on requirements not because the script of the interview contains such a section, but because they help us to better reason about our solution.**

## Consistency

- The deduplication service should never return false to what it has returned false before.

## Availability

- The deduplication service should be distributed, and some nodes failing should not affect the end users.

### Persistence and Durability

- We should be able to migrate the service to a new location / new set of machines altogether, without losing any vital data about which X-es has our system seen before.

What the above effectively tells us is:

- The solution should be distributed. The total amount of data to store would not be huge. It may even fit in memory. Therefore, we will only be looking at a few machines—not hundreds.

## Tip

Memory is fast, disk is slow. In this problem, since every record is a small number of bytes, we may be able to fit everything in memory.

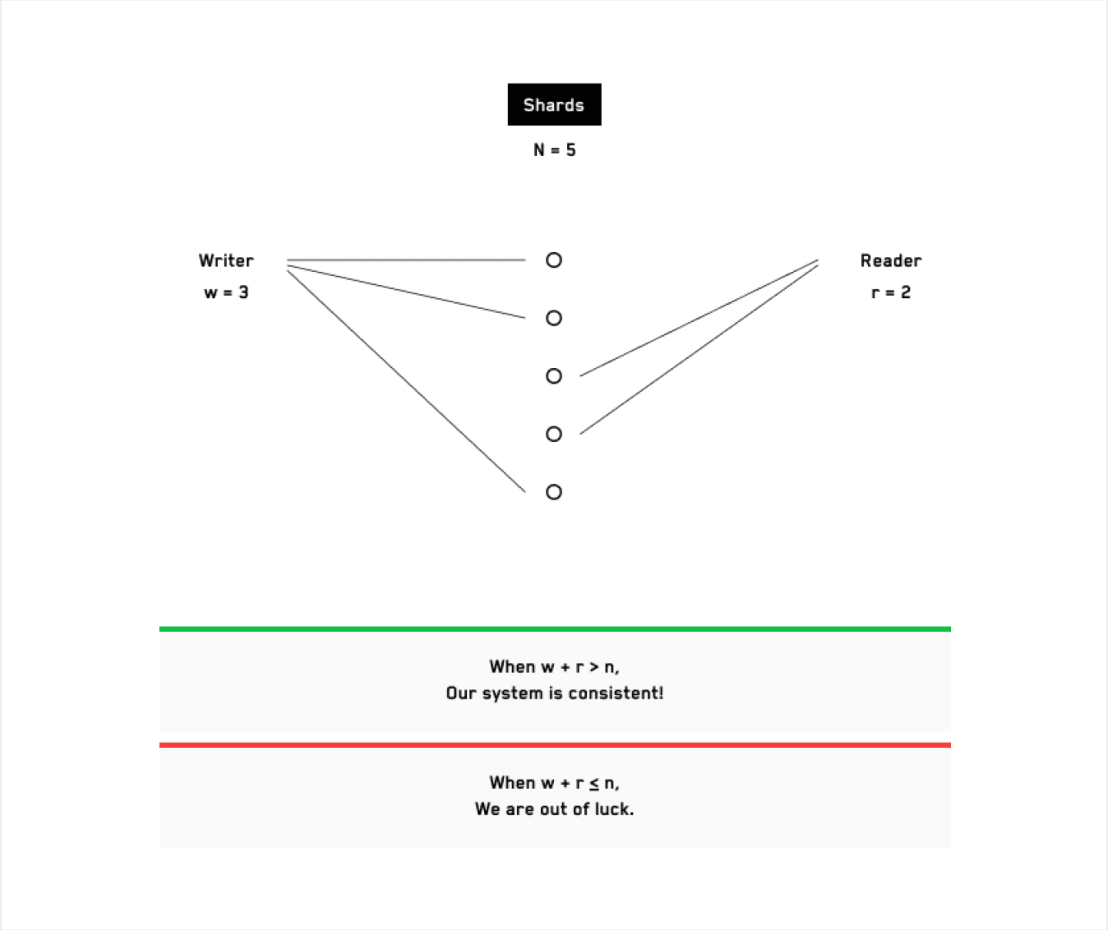
- The solution should be resilient. The main reason we need more than one machine is to ensure uninterrupted operation of the system if one of them is unavailable.

### Now would be a good time to introduce the simple idea of R + W > N.

This is a very simple trick, and while it is not often used directly in production systems, it's incredibly useful to know when learning about system design.

The idea is this:

- If we have **N** machines in total,
- We consider a write successful when the write has succeeded on **W** machines,
- And a read is successful if at least one of **R** queried machines contains the data. Of course, if more than one machine has the data, the response from all of the machines that have this data should be identical,
- Then we need **R + W** to be greater than N for our system to be consistent.




Keep in mind, we are still looking at a semi-immutable problem of deduplication: **we can only "get to see" every possible input X; we can not "unsee" an X.**

For a positive example, consider N = 5, W = 3, R = 3. What this means for our deduplication solution is:

- There are N=five servers in total.
- When we "write," i.e., when we see an X we know is new, we:
    - Take a note that "X was seen" on at least W=three servers (you can think of choosing them randomly now), and
    - Consider the write successful only after these three servers have confirmed this write as successful.
- When we "read," i.e., when we need to see if a given X was seen before or not:

- We ask R=three servers whether they have seen this X,
- If at least one of them responds with a “yes,” we know the answer is “yes,”
- And if and only if all three servers respond with a “no,” we know definitively that this X was not seen before.
- These R=three machines can still be chosen randomly.

### Tip

Take a moment to internalize why the above would work when R + W is greater than N, and why it may fail if R + W is equal to or less than N.

### Of course, in practice many more tricks can be used to improve the solution. (These ideas, while important, are out of scope of this trivial example problem):

- The machines will not be chosen randomly, but according to some **consistent hashing** algorithm.
- Some machines can be chosen as **leaders** to minimize overall load on the system. **Leader elections** would have to be used in order for such a fleet to function in an uninterrupted fashion without human intervention if the leader machine is suddenly down.
- Instead of sending a request to only R=three machines and waiting for all the responses forever, the system can query three machines right away, and maybe a fourth one if some 100ms later it has only received two answers.
- The data structures to keep the knowledge of which X-es have been seen by each machine can be probabilistic, such as the **Bloom filters**, thus reducing the RAM needed and/or disk used.

The redundancy factor is (N-W) for writes and (N-R) for reads; as long as there are enough machines to pick W of them to write to we can write, and as long as there are enough machines to pick R to read from we can read. The important part is that the write machines and the read machines intersect by at least one, so W+R should be strictly greater than N. This is because if W+R is equal to N, or is less than N, there's no guarantee that a set of W randomly picked machines and a set of R randomly picked machines will intersect. And if they do not intersect then the read will miss the data that was just written!

### Tip

Always pay attention to solutions that keep working “just fine” if some nodes are down. Since things are going to go wrong, it’s best to have systems that can deal with that without issue, so you don’t have to!

Last but not least: The above solution would use each machine’s RAM or disk capacity at its (W / N) fraction. In other words:

- If one machine can only handle the key space of up to a billion X-es,
- And you need to handle the key space of five billion X-es,
    - That is, five times that amount that one machine can handle.
- Then you need (W / N) to be at most 0.2 (or at most 20%).
    - Always good to allow for some extra room, so, say, we plan for ~16%, not 20%.
- You can go with W = 1, N = 6, but then R would have to be the whole 6. It's unwise to design a distributed system with R=N though. If just one machine dies, we can not longer have the total R of them to respond, and thus our reads will be stalled.

    - In other words, if you only write to one machine, you need all machines to be up and running while you are reading. Slow, and not safe with respect to machine failures.
- You can go with W = 2, N = 12, R = 12.
    - This is acceptable since R+W, which is 12+2=14, is greater than N, which is 12. The problem persists however: losing even one machine makes writes impossible, since, with R=N, if only 11 of 12 machines are operational it is impossible to read from "any" distinct 12 machines.
- Or you can go with some W = 4, N = 24, R = 22.
    - This is also acceptable since R+W is greater than N. And now, for reads to work, we can lose one or two machines safely, since both (N-1)=23 and (N-2)=22 is at least R.

Obviously, if you assign parts of the key space (i.e., different values of X) to N machines intelligently, not randomly, you can achieve better RAM and disk utilization. But the example is still quite illustrative: **we now know that without any custom knowledge on how to design a system, we can build a resilient, highly available, multi-node deduplication engine, from first principles!**

Before we declare ourselves done with the warm-up problem, let's consider its extension.

### Warm-up Problem Two: The Enumeration Problem

This problem statement is very similar to the above:

- The only way to call the service is: "Did you see X before?"
- The service responds with an integer: The unique, 1-based, index of this X.
- From this moment on, for this X the very same index is returned.
- The returned indexes should not have gaps. That is, they should go as 1, 2, 3, etc.

At first glance, enumeration is identical to deduplication, but instead of a hash set, we need a hash map.

But this first glance is deceptive. This enumeration problem is **substantially** harder than the deduplication one.

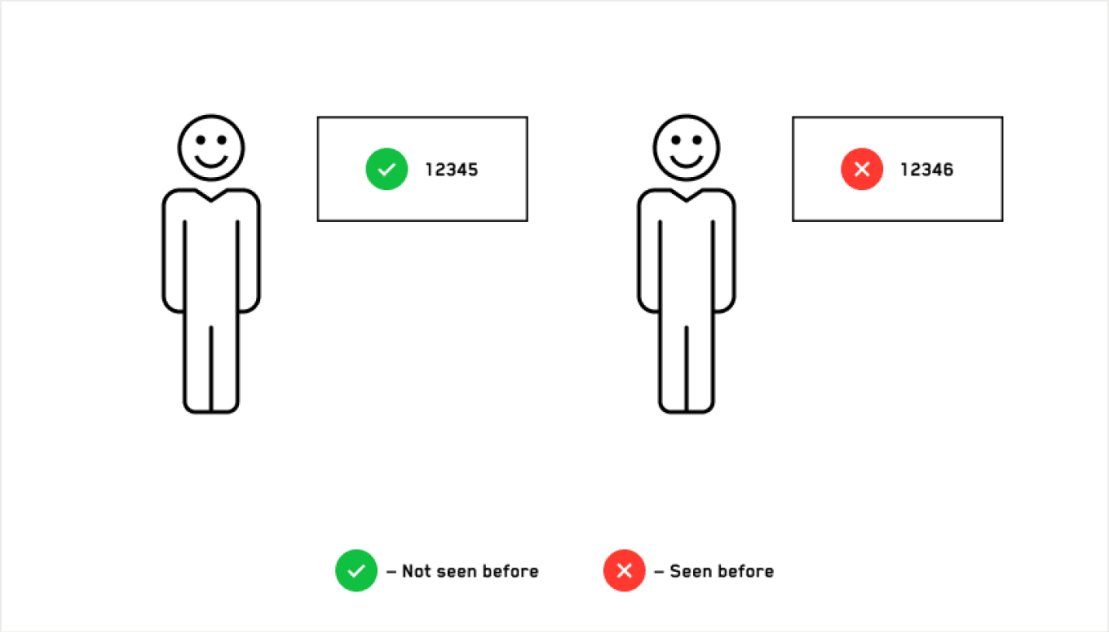
The challenge is in assigning IDs in a fixed order (as in: 1, 2, 3, and so on). In order to stamp the new, unique index, **all** other calls should wait, as this **very next unique index** "distributed variable" is a **shared resource.**

### A real-life analogy to help you understand enumeration

### A real-life analogy might help.

- Imagine the service is an office building with many clerks, and each request to it is a human being walking inside this building with a piece of paper. This paper is the content we are going to deduplicate.
- This human walks out with a green "check" or a red "X" stamp on this paper, representing the "not seen before" or "seen before" responses. In the case of enumeration, the "stamp" reads "ID assigned: _", and the number is filled into the "" part.
- Now, for the deduplication problem, the "user" can be routed to a specific floor right away. Or to a specific room on a specific floor.
- Or a clerk at the entrance (the "load balancer") can tie another piece of paper to the one the user walked in with. This additional piece of paper would list three room numbers. The user would need to go to those three rooms, in any order, and collect some intermediate paperwork. Then, upon exiting the building, another clerk (effectively, the same load balancer "on the way out," i.e., in the "reverse proxy" mode) ultimately gives the user a red or green stamp.
- This works just fine, as the work can be parallelized. But what if the user in the Enumeration problem were to "get a green stamp," (i.e., if this "X" was not

seen before)? Now, the very next available unique index should be immediately reserved and allocated for this particular X. Every single clerk, in every single room or every single floor, must be somehow aware that, should they see X again, not only they have seen X, but a fixed number N was assigned to this X.




In the case of enumeration, the very same (R + W + 1) > N trick would not work, at least directly.

## Because

- In the duplication case, each machine can be considered **independent**: If it dies between requests, or even while serving a request, the other machines kick in, seamlessly, and the user does not notice anything.
- In the enumeration case, a machine dying mid-request **loses part of the shared state**, most notably the “last” (or the “next”) index to use.

## Shared state is the root of all problems

## Tip

The more massive the shared state has to be, the more difficult it is to solve the problem in a generic fashion. And the enumeration problem is a perfect illustration of why exactly it is so difficult.

When attacking the Enumeration problem, the concept of a **read-write ratio** comes into play.

If there are only 1,000,000 distinct X-es in the world, eventually—and rather quickly—all of them will become seen. Since we don't need to invalidate indexes, each particular node of our service could "memorize by heart" the indexes for these 1,000,000 X-es, after which the problem can be treated as an immutable, read-only one. **And, as you know by now, immutable problems are easy to solve.**

Exhausting the set of unique inputs is quite a rare case. But it does happen often that the vast majority of inputs (say, 99.9%) would be the ones that have been already seen. Since inquiring about the already seen X is an immutable operation, this would result in a 999:1 read-to-write ratio (which is safe to round to 1000:1, for all intents and purposes).

## Remember (from the two warm-up problems):

- Deduplication is the simplest case of a mutable problem.
- Enumeration looks like a trivial extension, but it is far, far trickier.
- The CAP theorem is a bitch blast, and it is important to understand the boundaries of what's possible (and to ask clarification questions on the functional requirements step!).

### Real problem: Unique ID Generation, aka the Key Generation Service (KGS)

The problem of generating unique IDs serves as an excellent real-life illustration for the ideas we've discussed so far. Unlike the two warm-ups above, Unique ID Generation is a real problem. And it is commonly used as an interview question, including this mock interview hosted by interviewing.io.

## Requirements

We will, of course, be solving the problem in a distributed setting. In other words, the solution should be highly **available** and **consistent**. This means that we will be using several nodes that communicate with each other. This way, if some nodes die and/or become unresponsive, temporarily or permanently, the system remains:

- **Highly available** — it keeps functioning, and
- **Consistent** — it keeps satisfying its functional and non-functional requirements.

Pragmatically speaking, the problem is quite straightforward, as long as we answer three important clarification questions. All three fit solidly in the Functional Requirements (FRs) bucket.

## Remember

During a system design interview, we focus on functional requirements not to please our interviewers, but to set the table for ourselves and to stack the deck of the upcoming story in our favor!

### The three FR questions are:

1. Is it a strict requirement for the IDs to not to repeat?
2. How large is the ID space?
3. How many unique IDs per second should we generate (and what is the acceptable latency)?

Let's dive deeper into why these are the important questions.

**Question #1, about whether the IDs are allowed to repeat at all, is quite binary.** Either generating a non-unique ID is absolutely unacceptable, or it is allowed once in a blue moon.

To answer this from a practical standpoint, consider the cost of generating a non-unique ID for the business. If this cost is bounded by a reasonably small amount, then, in practice, it is not entirely unacceptable to have a non-unique ID from time to time.

Consider Amazon, AirBnb, or Expedia order ID generation. What can go wrong if the ID is not unique? A customer might have a ruined experience. What's the cost of this for the business? Probably in the order of dozens or hundreds of dollars; this cost is not likely to be above, say, $10,000. Thus, if the likelihood of a generated ID to not be unique is such that a duplicate may emerge once in ~10 or ~50 years, the "daily" cost of this "imperfect" implementation is less than what the company will pay for organic sweetener in the office. As a result, "fixing" this "problem" may not be worth it at all.

A strong user- and product-facing candidate should be able to articulate the above.

With this said, if the IDs are allowed to be not unique once in a while, the problem is sort of non-existent: generate a random 128-bit number, and you're good for ages. In fact, for most practical intents and purposes, even a random 64-bit number would do.

**Back-of-the-envelope math, which, unlike most SD interviews, actually is important here!**

- We're going to do a math deep dive. You don't have to follow all of it, but this deep dive is worth including so that you know what the most stringent interviewers may expect. If it gets too math-y, you can always skip it and come back to it later.
- A useful number to remember is that 232 is a bit over four billion.

- Thus, 264 is four billion squared (i.e., a bit over sixteen billion billion). In reality, the first two digits would be “one eight,” not “one six,” because 232 is indeed a tad greater than four billion. It doesn’t really matter though. **264 is a one, followed by one high digit, and then followed by 18 zeros. We know it’s 18 followed by 18 zeros**, but no interviewer would grill you if you say 264 is 16 followed by eighteen zeros; it’s the order of magnitude that matters.
- On the other hand, **we know there are approximately 105 seconds in a day**. (The true answer is 86400, ~15% lower.)
- The next number we need is covered by question #3 above: How many unique IDs per second do we need to generate? **Say, for the back of our envelope, it’s 1,000.**
- **For how long would we need to keep generating 1,000 random 64-bits unique IDs per second to have a duplicate?**
  - This sounds like a probabilities and statistics question, and indeed it is!
  - But we are preparing for an interview, and, in fact, this level of math is something you may well benefit from knowing. Besides, this is the level of probabilities and statistics that you may need in a DSA interview, so there’s no harm in repeating it here.
  - Let’s say that we generate K random numbers, each an integer from a range from 1 to N. The question that is relatively straightforward to answer is: What’s the probability that all these K numbers are distinct?
  - Obviously, if $K \geq N$, there will be at least one repetition, so we are looking for the case where K is less than N.
  - **Since each random number is unique, the total number of ways to “draw” K numbers, each of which is from the range from 1 to N, is $N^K$**. This should be intuitive. If N is 10, we’re drawing decimal digits; with N = 10 and K = 1 we can draw one digit, for 10 different options; with N = 10 and K = 2 we’re drawing

two-digit numbers, which offers 100 options; and with three digits we're looking at 1000 options to choose from, etc.

- **By the way, this is another reason why computer scientists count from 0, not from 1.**
    - If K is (N - 1), then from all (N ^ K) ways to draw K numbers there are N! (N factorial, 1 * 2 * … * (N-1) * N) ways that are "good" for us. This requires an explanation. First, for all (N-1) numbers to be distinct, there should be exactly one number "not picked." So it will be N times something, as this one number that is not picked can be any of N possible numbers. Second, for the remaining (N-1) numbers, we will be looking at a random permutation of them, and the number of permutations is a factorial, in this case (N-1)!. Multiplying (N-1)! by N we get N!.
    - **Expand this for an exact formula, and the one below for a way to approximate it.**
    - For K = (N - 1), the probability of no repetitions will be (N!) / (NK). This number can be approximated intuitively (it's the product of N/N, (N-1)/N, (N-2)/N, all the way down to (1/N)), or using the Stirling's formula.
    - For a large N, it would be a damn small probability, by the way.
    - Hint. This follows from what you you ought to know: that O(log(N!)) = O(N*logN)

Now, for a K that is much smaller than N, of a total of N^K possible random sequences, the number of "good" ones is **C(N, K)** ("N choose K," as we necessarily need to see K distinct numbers, and we can choose any K) **multiplied by K!** ("K factorial," as we can see these K distinct numbers in any order).

And N choose K is computed as N! divided by (N-K)! * K!. Google "binomial coefficients" for further reading. For this particular problem, as any order works,

we need to multiply this number by K!, effectively, removing K! from the denominator. The formula then becomes (N! / ((N-K)!)).

**The intuitive meaning of this formula is to multiply K numbers, the first of which is N, the second is (N-1), the third — (N-2), etc.** For K = N, as 0! = 1, the above formula gives N!, which is identical to our logic above. Same goes for all numbers but one, K = 1, as the one "remaining" number is uniquely identifiable. Effectively, the "multiplied by one" piece at the very end is omitted as redundant. If K = 2, we are leaving two numbers aside, and, thus, the "multiplied by two" piece goes away as well; for K = 3 the "multiplied by three" part is gone, and so on. As "N factorial" multiplies all the numbers from 1 to N inclusive, to remove the "tail part," that is "from 1 to Q," so that the smallest number we get to when "multiplying down" is (N - Q + 1), we just divide N! by (Q!). Given that we need to "multiply down" K numbers, from N downwards, the last operand would be (N - K + 1), which means the denominator for N! will be (N-K)!.

Therefore, the probability of no repetitions ("of no collisions" would be a better way to phrase this) when choosing K numbers randomly, each one of N possible options, is (N!/((N-K)!) / (N^K), which is N! / ((N-K)! * (N^K)).

If the above sounds intriguing and fascinating, good further reading might be the page about the Birthday Paradox on Brilliant.org.

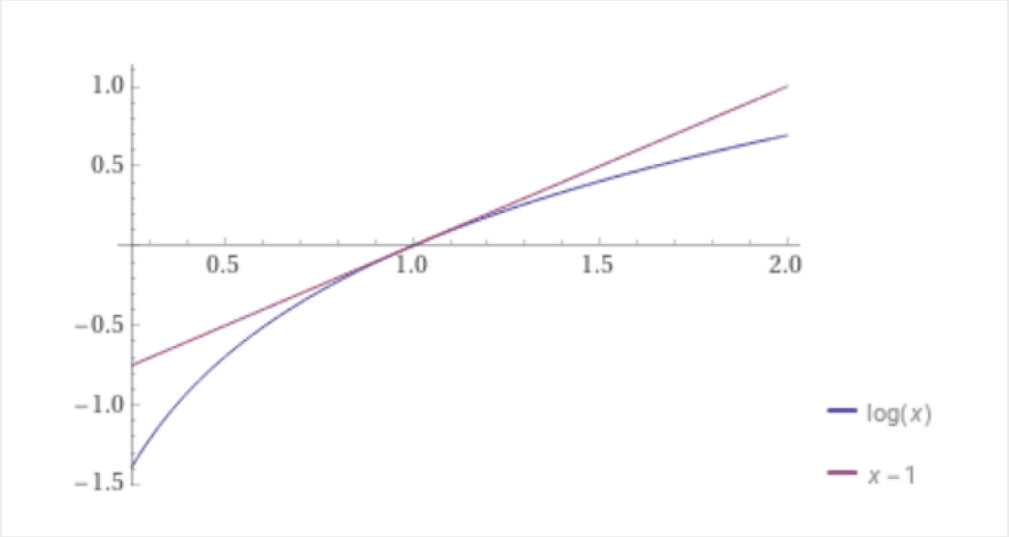
**To wrap up this section, say that we want a 99.999% ("five nines") probability of no collisions. This is identical to wanting a 0.001% ("10^(-5)") chance of at least one collision.**

1 - 10^(-5) = (N! / ((N-K)! / K!)) / N^K

This is the exact formula to compute, should our goal be to get an exact answer. What we need though is an approximation, which only requires a subset of the above logic to get to.

- **Approximately for how long would we need to keep generating 1,000 random 64-bits unique IDs per second to have a duplicate?**
  - The probability of a collision on the first drawn number is zero.
  - The probability of a collision on the second drawn number is 1 / (264).
  - The probability of a collision on the third drawn number is 2 / (264). And so on.
  - We need to keep multiplying these numbers until we hit some target threshold. Intuitively, let's aim for "at least five nines" (i.e., 99.999%) of **not** hitting a collision. This is the same as having at most 0.001% probability of hitting it!
  - The trick is to replace multiplication by addition. A trick that mathematicians know (AND ONE THEY DO NOT WANT YOU TO HEAR ABOUT IT! JK, they do!) is that you can get **a good approximation when multiplying numbers close to one by adding up their one's complements.** Simply put, (1 - small_number_one) * (1 - small_number_two) is very close to (1 - (small_number_one + small_number_two)).
  - **This trick is explained in the Birthday Paradox example, and the TL;DR is:**
    - (1-A) * (1-B) = exp(log((1-A) * (1-B))
    - exp(log((1-A) * (1-B)) = exp(log(1-A) + log(1-B))
    - For a small X, log(1-X) = -X, thus:
    - exp(log(1-A) + log(1-B)) is approximately exp(log(1-(A+B)).
    - And therefore (1-A) * (1-B) is approximately (1-(A+B)), QED.
- **So, we're looking for a K such that the sum of (1+2+3+…+K) / 2^64 is at least P=0.001% = 10^(-5).**
- (1+2+3+…+K) = K*(K-1)/2, loosely ((K^2)/2), and it should be equal to (2^64) * (P=10^(-5)).

- As shown in the Brilliant.org link, for P=0.5 the [approximate] answer is sqrt(2^64 * 2 * 0.5) = sqrt(2^64) = 2^32. For P=10^(-5), the [approximate] answer is will be sqrt(2^64 * 2 * 10^(-5)) = 2^32 * (1/223).
- Intuitively, **as our target threshold probability decreases X times, as long as it remains small, the number of numbers to generate on average before the first collision decreases sqrt(X) times.**
- **NOTE: We've actually tested this approximation, and we have the code handy. If you're curious, let us know.**
    - TL;DR: For small probabilities of collision (i.e., for several nines of no collision, 99.9%, 99.99%, 99.999%, etc.), it gives remarkably accurate answers:
    - For range = 18446744073709551616 (log_2(range) = 64.0), the desired P of collision = 0.000010 (-log_10(P of collision) = 5.0), need to draw 19207723 numbers. The approximation for the number to draw is 19207677.7, and this number is 0.0% off the actual number.</code>
    - For larger probabilities, the approximation does not work as well, but it is still quite accurate. For P=50% the approximation of 2^32 is 17.7% lower than the computed number; for P=10% it's 2.6% off; and for P=1% it's 0.3% off.
    - This is consistent with the theory, as the approximation that log(1-X) = -X is more accurate for small X-es, and the formula results in more negative gets away from the actual number as X increases, see this WolframAlpha plot.




- **To internalize and memorize this:** We know that for P=0.5 (”fifty-fifty whether there will be at least one duplicate”) the approximate number of numbers to draw from the 2^64 range is 2^32. So for P=0.25, which is half of P=0.5, the target number of numbers to draw becomes 2^32 / sqrt(2). For P=0.005, half a percent, we need to divide 2^32 by a square root of 100, which is ten; thus, to reach a half a percent chance of drawing a duplicate, we need to draw not four billion numbers but four hundred million numbers.
- **That’s how back-of-the-envelope math is done in the realm of probability and statistics.**

- To recap the above math-y deep dive, among other things the Birthday Paradox example shows that for a 50/50 chance of collision in a 64-bit space, the number of numbers to draw is approximately a square root of 2^64. The square root of 2^64 is 2^32, which we know to be around four billion.
- At 1,000 requests per second, ~10^8 requests per day, we get to 4 billion, 4 * 10^9, in about six weeks. And 50% is a **huge** chance for a collision!
- For a 10^(-5) chance of collision, equal to 99.999% chance of no collisions, the six weeks interval is to be reduced by a factor of ~223 (one over the square root of (0.00001/0.5)), which is slightly under five hours. **Not cool.**

- In other words, if we generate an order of 1,000 “random” IDs per second, and if our IDs are randomly generated 64-bit numbers, we can only expect to run for under 1.5 months until we have a coin-flip chance of having at least one collision.
- And if we’d like to be five nines (99.999%) confident that we encounter no collisions, at 1,000 random IDs per second we can only run for ~five hours until the solution is no longer “safe” to use.
- (This also assumes our random numbers generator truly is random, which is not necessarily true. Not that it’s that important, but your interviewer would appreciate you making this remark here.)
- Of course, doubling the key space, from 64 bits to 128 bits, multiplies this number by the square root of four billion, which is 65,536, 2^16. 65K times six weeks is over seven thousand years, and two hundred twenty-three times less is 32 years.
- If this topic interests you, Google announcing its first SHA1 collision, in 2017, is a good read.
- Last but not least: Remember that **bits, or, more specifically, entropy, follow simple rules of combinatorics**. 2^64 is sixty-four binary digits, zeros or ones. Each Latin letter, of which there are 26, is log_2(26), approximately 4.7 bits, which makes it 28.3 Latin letters to make it to 64 bits.
- **Five blocks of five random Latin letters, i.e., HRVEN-UTABR-GATBR-KTNBE-UTAHZ, contains a bit more entropy than 2^64**. Which means that, yes, you carry a 50% chance of witnessing a collision among these within six weeks of generating them at one thousand times per second rate. Obligatory XKCD: Password Strength.
- And many airline, or AirBnb, booking numbers, which are six Latin letters each, only cover a bit over 300 million unique values. Either these companies have

solved their Unique ID Generation problem, or they are virtually guaranteed to have collisions. Now you know this for a fact, from first principles!

With the above deep dive being said—and you should somehow indicate to your interviewer that you understand it!—let's approach the Unique ID generation problem under the assumption that **the IDs should be guaranteed to be unique.**

**Question #2** effectively boils down to "64 bits vs. 128 bits." 128 bits, as shown above, are a lot. For the record, a standard UUID is of 128 bits, although a quick Googling shows it "only" contains 122 bits of entropy.

And **question #3** is only important in conjunction with question #2, as what truly matters is **the rate at which the key space is being exhausted**. Simply put, **the very problem is almost identical if we add 10 more bits to the IDs and generate the IDs 1,000 times faster.**

### Tip

Though it is rare, sometimes system design interviews are indeed about doing math with estimates. A good interviewer will test your intuition on the above, and human intuition is notoriously unreliable when it comes to very small and very large numbers in general and to probabilities in particular. So if you skipped the above math-y deep dive, at least read up on the Birthday Paradox.

### Solution

So, effectively, the true problem is:

- **Do we try to utilize the key space effectively?**
- **Or are we allowed to use a larger key space?**

Really, that's it.

For the unique key generation problem, just do a bit of trivial (*) math.

*(*) Professors tend to refer to anything that's semi-straightforward to follow as trivial. Thankfully, our interviewers are not that picky. Although the above is relatively standard math to someone who is fluent in probability and statistics, and, chances are, your interviewer may well be—system design interviews are often conducted by more senior people, and more senior people work with data more often than average.*

**If** we need to utilize the key space effectively, we're fully in the realm of distributed consensus, etc. **Hard problem.**

If we can be loose, we just need to employ a few simple tricks to minimize our risks. **Easy problem.**

### Solution to the Easy Problem

First of all, if you are looking at the easy problem, it should be noted during the interview that **just generating the IDs randomly would solve it for all intents and purposes.**

Go ahead and explain this to your interviewer. Explain that even if each collision costs the company $10,000, it is still not worth an extra minute of your time, as an engineer employed by this company, to be solving this problem.

Because this is The Correct Answer, if this unique ID generation problem presents itself in the real life setting.

*(Not to mention that any large company that employs you expects to make considerable money from you; for top IT giants this figure is over one million dollars per year per engineer, although things did turn south since 2020.)*

Of course, after you cover this, the interviewer will ask you to actually solve the problem, in a scientifically and engineering-ly precise way, in an "academic" setting. This is what the next section is about.

But before you get too fancy, it won't hurt to suggest a few trivial improvements. In short, having a lot of key space allows for plenty of "cheap" tricks.

For example, each node of your now-distributed service can just assign the IDs in the 1, 2, 3, … fashion. You can then prepend each generated ID with something that contains:

### a) The ID of the machine (node/server) that has generated this ID.

So that the IDs generated by different machines will never intersect.

### b) The timestamp when this particular machine has started assigning IDs.

So that when this machine restarts, it does not issue the same IDs. Even HH:MM:SS would do as the timestamp, BTW.

If you have a large key space (say, 128 bits), you can easily allocate ~32 bits for the ID of the machine (an IPv4 address) and another ~32 bits for the Unix timestamp, in seconds, when this particular machine has started its "session." This leaves you with plenty of indexes per each node: 2^64, to be precise, as you've "used up" 64 bits of 128.

And if one of your machines **suddenly** runs out of whatever range you have allocated for it, you can "just" restart that machine. It will get itself a new prefix and continue afresh. Of course, the restart itself is not necessary, as the machine can just (automatically) change the value of that variable it uses internally.

### Solution to the Hard Problem

The above solution should be good enough to pass the interview about Unique ID generation. Before we get to solve the Hard version of it, let's add a few more assumptions. They do, by the way, qualify for **Non-Functional Requirements**. Here they are:

**a) The IDs generated should not just be sequential.**

This is for security / cryptography reasons. It's not cool when your client—a prospective attacker, always—can just decrement the ID they have just received from the system by one, and plausibly expect that ID to also be a valid order/booking ID.

**b) The API is such that clients can receive IDs in bulk.**

If we truly want to scale to tens of thousands, or even millions, of new IDs generated per second, requesting each individual ID in its own HTTP, or gRPC, call is an overkill. A strong candidate would do well by noting this.

For (a), a simple solution comes from number theory. Security specialists know it. Just perform some relatively trivial mathematical function before returning the ID to the user, while keeping this function **reversible**, so that when F(x) = y, some G(y) = x. A Modular Multiplicative Inverse might do the job just fine.

And (b) is exactly where the solution to the broader problem comes into play!

Imagine that you have designed a system where the "users" can "request" IDs in bulk. (We will explain how to do it in a minute.) The trick is that **each node of your system is effectively performing the same task: it needs to request a large "bulk" of IDs from some "central authority," and then distribute these large bulks in smaller quantities to its "retail customers."**

It's as simple and as genius as that.

To arrive at a "proper" solution, the "large" bulks should not be too large. Because, if a node dies, it is impossible to tell how much of its latest bulk it has "distributed" so far. In other words, when a node dies, some part of a key space will be wasted.

Moreover, if the node does not die, but terminates gracefully, it may then "return" the "unused excess" of its bulk to the system, which sounds very fair. But it may present a challenge to the "bulk provider service" to deal with "partial" bulks. It may

well be better to deliberately discard the “rest of the bulk,” even if the node that has previously requested this bulk is terminating gracefully.

Simply put, the bulk size can be calibrated such that each node requests a new “bulk” to “distribute” approximately every several seconds or maybe several dozen seconds. **That’s the sweet spot**. Make this number (this bulk size or this time window—they can now be used interchangeably) too tight, and the “central authority” would need to handle more load than needed. Make it too loose, and, especially if machines do die often (or need to be migrated often), a non-negligible portion of the key space would be wasted unnecessarily.

Now, to the gist of the problem. **The very “central authority”**:

1. Can not truly be “central,” but should be “distributed,” and
2. Actually deals with very little traffic, as each node literally talks to it once every several seconds or dozen seconds.

So, in a true System Design fashion, use some standard distributed component there, FFS!

- It could be Google Spanner, because, with this traffic, it will be cheap.
- It could be anything with internal Leader Elections implementation, such as:
    - ZooKeeper,
    - or even Kubernetes/Consul itself!
- It could be any distributed DB, as long as it supports strongly serializable transactions over a reasonable number of replicas.
    - Literally, both MySQL and Postgres would do, as long as you have a quorum of some N = 5 or 7 or 11 nodes. (We usually use an odd number of nodes for leader elections, so that they are impossible to break into two equally sized parts).

- Kafka is another option, as you would literally be publishing single-digit messages per minute into one topic of one partition!

Even if each “call” to “reserve” another “bulk” takes a second or two to settle—because the request would have to travel across the globe, possibly more than once, to guarantee strong consistency—it’s perfectly fine with your design, as long as each node requests a new “bulk” those few seconds in advance, before it runs out of its “current” “bulk.”

That’s it for ID generation. Relatively simple.

## Remember

Any design for Unique ID generation that attempts to store Every Single Issued ID is a **bad idea**.

## Why is it a bad idea?

- Because, as we know by now, if the total number of IDs to issue is small, well, there is no problem.
- And if the total number of IDs to issue is large, the waste on the DB resources will be colossal, to say the least.

More junior, and/or less experienced engineers often make this mistake. As they say: “Forewarned is forearmed.” At least, store “bulks” only, not each issued ID. Better yet: only store the “latest issued bulk ID” or the “next available bulk ID.” That’s it—relatively simple still.

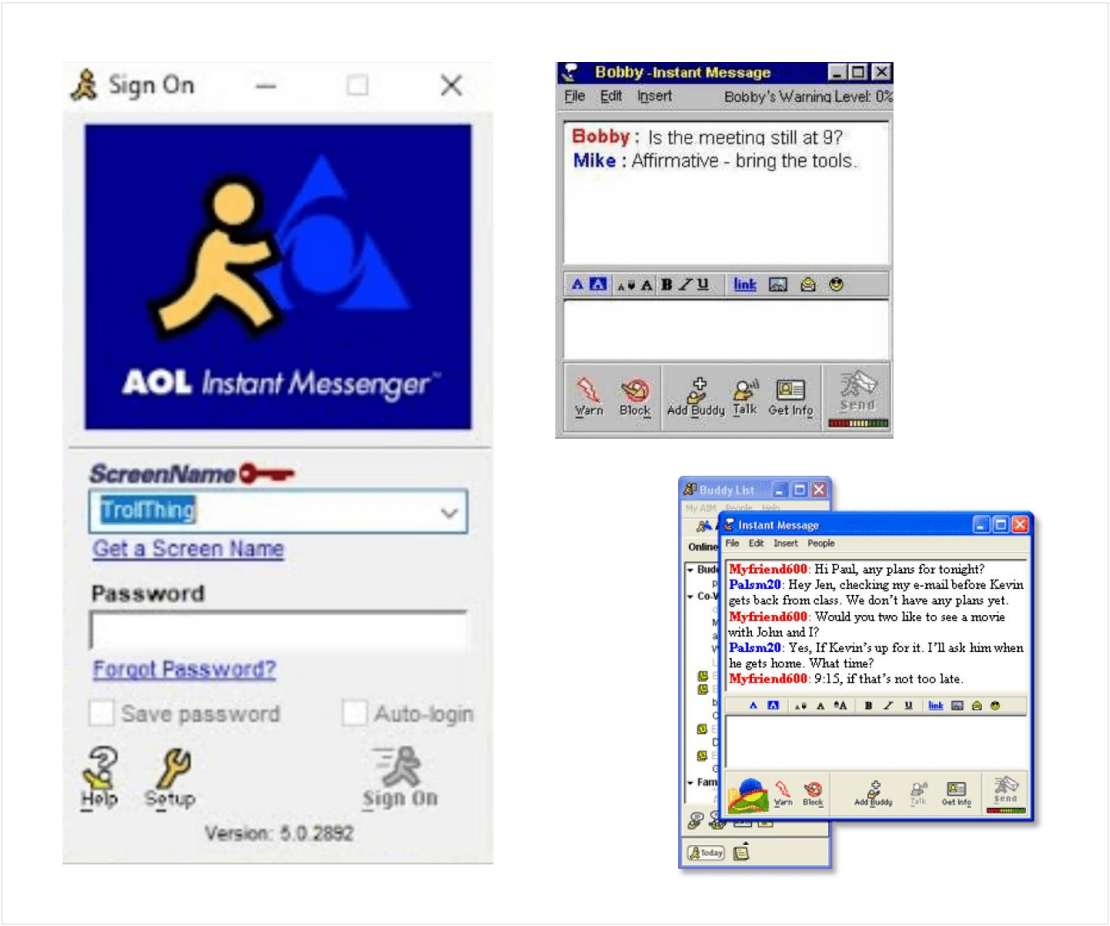
# Chapter Three

All right, we now know enough about things like mutability, deduplication, and basic requirement gathering skills to be dangerous. Let's now tackle a larger problem—not too complicated, but it'll stretch us a little bit.

## Design AOL Instant Messenger

This is almost certainly a system design question you haven't encountered, and depending on when you were born, it might not even be an application you've seen before! No worries, we can think of this as one of the simplest chat apps imaginable. Keeping it simple allows us to design any chat app from first principles and provides a solid structure for us to work with, regardless of what chat app we are asked to design.

For the younger readers who aren't familiar with the AOL Instant Messenger (AIM) application before, here are a few snapshots so you can get a sense of the basic functionality of the application.




## Requirements

Yikes! Those font choices clearly weren't vetted by a UX designer! OK, so clearly the app has a few specific functionalities. A requirements list may look something like this:

## Functional Requirements

- Ability to sign up for AOL Instant Messenger
- Ability to generate a screen name
- Ability to authenticate yourself with a username and password
- Ability to save username/password securely & auto-login
- Ability to add a "buddy" to talk to on AIM
- Ability to select a buddy and send chat messages to them in real-time
- Ability to block, warn, and delete buddy

Wow, that's still quite a large list for such a simple application. For now let's ignore the authentication pieces—while they are important, they don't really make up the core of this application. Let's spend some time focusing on the actual messaging parts and assume we can log in already.

As we did in the previous chapter, we should discuss the scale of our app in terms of functional requirements. Again, this isn't just strictly necessary for an interview, it's also a useful tool to help frame what we actually care about in the design of the system.

### Non-Functional Requirements

Ranking the order of importance with functional and non-functional requirements is silly because a single requirement not being filled will lead to a bad system.

### Tip

Still, for any given design problem, there is usually at least one particularly relevant requirement. This means that there's likely to be one requirement which is the linchpin of the system.

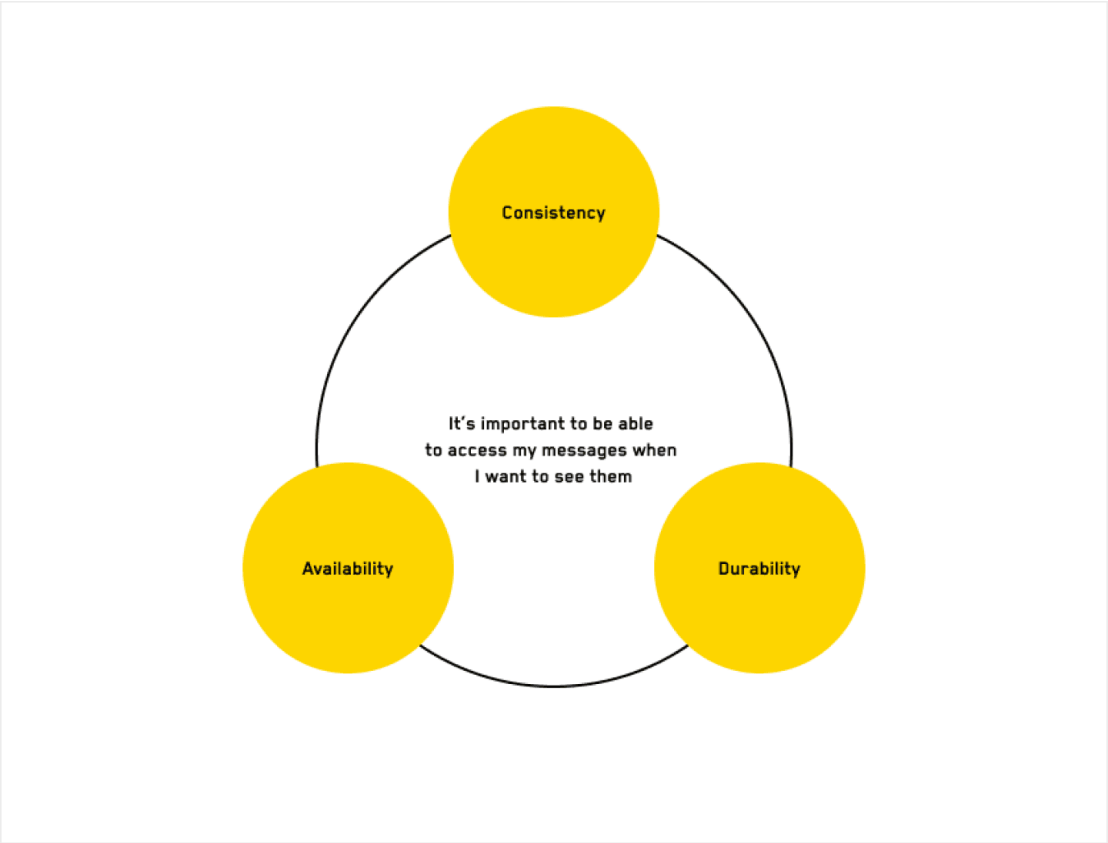
### Question

Generally, what do you think are the most critical parts to an app that allow you to talk to other people? Seriously, think about it. I'll be here when you get back.

Did you think about it? I mean it! Think for a moment about this. What are things that are just "givens" about a chat app? What have you come to expect when it comes to these types of apps?

OK, so you might have phrased it differently, but if you thought something like, "It's important to get messages in a *timely manner*," or maybe "it's important to always be able to send a message when you want to," or possibly even "It's important to be able to access my messages when I want to see them," then fantastic—you're absolutely right.

In plain english, we've just named our most important non-functional requirements: **Consistency**, **Availability**, and **Durability**.




## Availability

Have you ever tried logging into a financial website like a bank or retirement account and run into a message saying the system wasn't available at the moment and to "try again later"? If you had a chat app that did that, would you keep using it? I think not!

Although chatting with any particular user will depend on them being online, we want the actual ability to chat with them to always be possible. Therefore, this system should be Highly Available

### What does Highly Available even mean though…?

### Remember

Did you know there is an actual availability metric that is used within the industry? Agreements between product teams and their clients, typically called Service Level Objectives (SLOs) (also called “Service Level Agreements” or SLAs”), give a percentage of time that their system is guaranteed to be **available** for use. This metric is typically measured in 9’s. A typical target for a highly available system might be 99.9% (“three 9’s of availability”), and this target would mean that the developers guarantee their app would be down and unavailable for use less than 9 hours out of an entire year. That’s less than 2 minutes a day.

### Warning

There is an exception to the above “Tip” about SLOs. Although it’s true that you don’t need to worry about SLO’s for your own service, it is NOT true for any dependencies you have. You need to know the characteristics of those dependencies to be able to design for the right problem. **Our rule is to Never Trust a Third Party Service**. Any time a 3rd-party service is mentioned in a system design interview, ask the interviewer how trustworthy the 3rd-party service is. You can say, ”What is the SLO for this server?” or “Is this server reliable?”

### How to Get Yourself Unstuck – Tip #5:

Consistency is important because the order in which distributed messages are sent matters. If you send your friend a diatribe about how Vim is superior to Emacs (why are we even still arguing this? □), and your long-winded chat

messages are received out of order, then your friend will be totally lost and won't comprehend your persuasive brilliance.

Common ways to ensure a system is highly available include:

- Diverting traffic to other servers seamlessly without losing data by employing tools like **load balancers, reverse proxies, and DNS**
- Investing in redundancy by leveraging **database replication, regions, and availability zones**
- Protecting the system from failures and atypical behaviors by utilizing **circuit breakers, rate limiting, load shedding, and retries**
- Detecting system failures with **monitoring tools** and techniques like **chaos engineering**

A plethora of techniques exist for increasing an application's availability (i.e., to get progressively more 9's after the decimal point), but we will start with a small handful just to give you the big picture.

## Durability

We should be able to migrate the service to a new location / new set of machines altogether, without losing any vital data. Systems achieve this by having multiple copies of the data. Three major strategies for this include:

- **Backups:** these are straightforward and what you probably have implemented on your own machine right now (as a software person you do backup your personal machine, right? ). We can do full backups or potentially just differential backups to avoid making entire copies of the data since this is expensive.
- **Redundant Array of Independent Disks (RAID):** Storing data across multiple disks rather than just on a single disk allows us to avoid data corruption and even aids in data recovery.

- **Data Replication:** this idea is different from backups and involves quickly replicating data to other servers in other unrelated systems for extra redundancy; that way, if an entire system goes down, you'll still have copies of your data on another system.

## Consistency

Consistency is important because the order in which distributed messages are sent matters. If you send your friend a diatribe about how Vim is superior to Emacs (why are we even still arguing this?), and your long-winded chat messages are received out of order, then your friend will be totally lost and won't comprehend your persuasive brilliance.

### Wait… why could the messages I sent be delivered out of order?

Great question!

If you send 3 messages to your friend in rapid succession, they have to go across the internet and reach our system's servers. There are billions of routes that network packets can take from you to the AOL server. Depending on other internet traffic, it's entirely possible that one message gets delayed and another gets to our servers faster than a message sent earlier. Networks are complicated, but this is a key consideration to keep in mind. Order of the data being received is not guaranteed when sending data to and from any point online.

### How to Get Yourself Unstuck – Tip #6:

Ask yourself, "Would it be fine if the data in my system was occasionally wrong for a split second or so?" If the answer is yes, then you probably want **eventual consistency**. If the answer is no, then you're looking for a strong type of consistency called **linearizability**.

Let's practice the above tip to be sure you've got the hang of it.

- You're designing a meticulously timed nuclear launch sequence for the military. Is it OK if the data you're sending to different systems is sometimes off for a bit? If you said "no" (meaning we do need linearizability), then you're correct. If you said "yes," then please don't be a software developer for the military. 
- You're designing a chat app. While messages need to show up in a timely manner, is it acceptable for them to be a split second slower on arrival from when they were actually sent? Is it fine if there are momentary split-second discrepancies between the app on your iPad and the one on your iPhone? If you said, "Yes, because eventual consistency is good enough here," then you understand the concept. Bravo. Let's move on!

### How to Get Yourself Unstuck – Tip #7:

Don't confuse consistency terms. Besides being common terms talked about in system design, **what do all three of these terms have in common**?

- ACID
- CAP Theorem
- BASE

All three terms talk about **consistency**! The 'C' in both ACID and CAP stands for Consistency, and the 'E' in BASE stands for Eventual Consistency.

What makes matters worse is that the term means something different in each context. Be sure to separate these ideas in your head before you talk about consistency in an interview!

- ACID consistency discusses transaction guarantees within the context of database constraints.
- BASE eventual consistency discusses guarantees around objects being updated and what will be returned by all nodes when the same info is queried after an update.
- CAP consistency is about the tradeoffs in distributed systems between partitioned networks, nodes always being up to date with the latest values, and the system always being available.

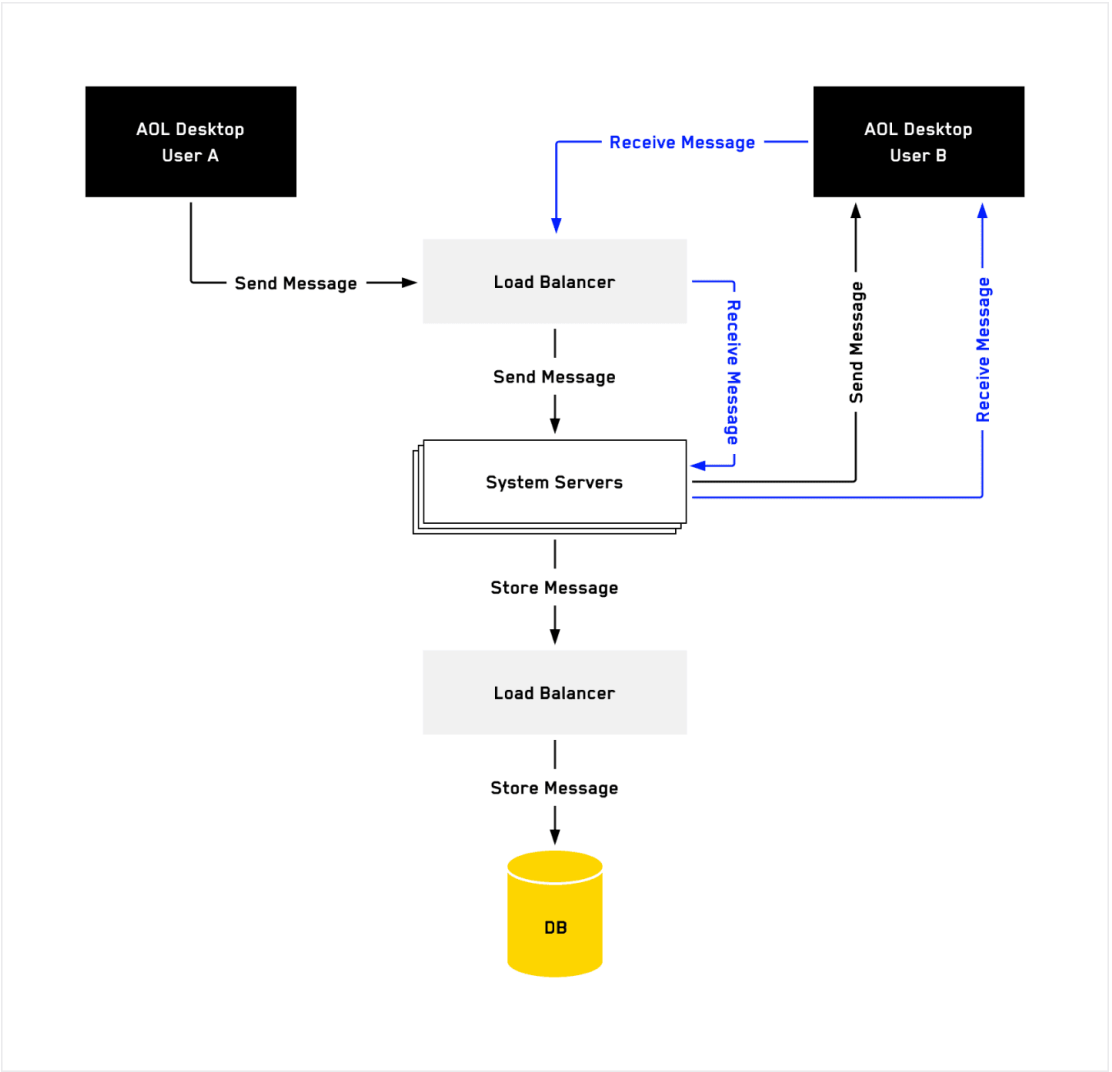
## Solution Design

Awesome, now that we've discussed the requirements, much of our work has already been done. Let's talk about what the system actually looks like. 

### How to Get Yourself Unstuck – Tip #8:

It's common to try to detail every part of the system's design like you see people do on YouTube. Realistically, these videos are scripted, and the drawings are fast-forwarded. In a real interview, you won't have time to actually detail every part of the system, and that's OK! It's expected that you'll abstract away pieces that aren't particularly relevant. It's a good practice to call out what you're abstracting, but just focus on the general data flow of the system.

In our diagram we need to show how data is getting from one client to another client and what it does in the system. The diagram can start simple and then evolve if the interviewer wants you to "zoom in" and discuss specific parts. In our example though, the diagram could be as simple as this:




## Data flow:

- Our first user (AOL Desktop User A) sends a message to our servers, and we have it hit a load balancer to avoid slamming individual servers.
- We have a fleet of servers waiting to work. Upon receiving a message, they store the data in the database, which is also behind a load balancer.
- The database piece is left intentionally simple, but we can deep dive into it if the interviewer wants us to. This alone can be an hour-long discussion, so we recommend that you keep this simple and call out that you're handwaving this for now so that you can finish the rest of the system first.
- The second user (AOL Desktop User B) can get data by accessing our servers (through the load balancer) and then getting sent data back directly.

Now that we have an idea of how the system has data flowing through it, we might want to discuss one other critical piece. The sending of the data makes sense from the diagram alone, but how does that second user know it's time to receive data? Are they checking the service every few seconds to see if something new is present for them? Is there some way we can alert the AOL user that they have a new message? The app is real-time, so how do we ensure that messages are delivered as promptly as possible? These questions are answered by knowing a bit about key ways computers can interact with other computers. There are three major types of ways computers talk to one another: Long Polling, Short Polling, and WebSockets.

These can be best explained through an analogy.

## Short Polling, Long Polling, and WebSockets are we there yet?

### *Short Polling*

Remember when you were younger and you'd always ask the question, "Are we there yet?" Repeatedly asking this same question every few minutes is a good example of short polling. Over a short period of time, we are constantly asking, "Are we there yet? Are we there yet? Are we there yet?"

This is short polling in a nutshell. We repeatedly bombard the servers with the question, "Are there new messages for me?" Just as this was a bad idea to do in your parents' car when you were a child, it's a bad strategy in basically every system design structure. It's annoying to the servers (causes extra processing power) and doesn't scale (can you imagine the amount of resources wasted by 1 million users all asking our system this every few milliseconds?).

### *Long Polling*

Did you ever have a forgetful aunt or uncle? Imagine that you're a little older now and driving on a roadtrip with that person. When you both get in the car, you ask them to tell you when you all reach a particular spot on the trip. This could be fine for a shorter trip because they'd probably remember to tell you. But on a really long

trip, they might forget to tell you when you reached that particular spot, perhaps because the car ride took too long and they'd simply forgotten you ever even asked them to tell you.

This, again, is an analogy for long polling. Our client reaches out and asks the server to tell us when something new has updated for us, which helps us successfully avoid the waste of resources. But this fails when the time between responses can be particularly long (think more than a few minutes). So long polling can be good when we are expecting data repeatedly, but it's not great when it could be a while before we get the data.

### *WebSockets*

Finally, let's imagine a car ride with our best friend. We ask them to tell us once we've come to a particular spot on our journey. They say, "Sure, I'll tell you when we get there," and then we patiently wait for them to tell us. They aren't forgetful, so we trust them to let us know no matter the length of the car trip. This, in essence, is WebSockets.

A key part that distinguishes WebSockets from long polling is that with WebSockets we can have arbitrary lengths of time pass without needing to worry about the connection timing out and the server "forgetting" us. We also have two-way communication, so the client can talk to the server, but the server can also directly talk to the client (whereas in long polling the server can "forget" how to talk to them).

For a chat app, out of these three choices, a case could be made for either long polling or WebSockets, but WebSockets is clearly a better design choice since we don't know how long it'll be between messages being sent and arriving.

### Conclusion

This is the core of every chat app, and hopefully it provides you with a good description of how we can model future problems. Start with requirements, decide

what matters most for the application with non-functional requirements, show the flow of data in your app with a diagram (maybe a couple!), and then iron out the details.

Though we certainly could go deeper into any part of the design and unpack things further, this represents a good overview of how the first major chat apps were designed and shows the basic model that all chat apps since that time have been built on.

### What did we leave out?

- A few things that can exist in modern day chat apps include:
    - Authentication
    - GIF/Video messages/Link previews
    - That little “read” status you see in your text messages
    - Group chat capabilities (common in apps like WeChat and Facebook Messenger)
- Other areas we could dive deeper into but hand-waved for simplicity:
    - DB selection. If you search online for chat app system design questions, you’ll see that people have strong opinions on databases as if it’s the only way to design the system. In reality, the topic is nuanced and generally doesn’t have a correct answer. You can refresh on databases by using the Table of Contents (on the left) to navigate to the section called "12 fundamental (technical) system design concepts" (found in Part 2).
    - Load balancing. It’s in the diagram, but we don’t really discuss how it works in any detail.

- Caching: It makes sense to cache recent messages (perhaps the last 20ish) so that when a user opens their app they can see the messages without having to wait for them to load.
- Data partitioning: If we choose to partition data, then how can we split the data?

## Chapter Four

### The fourth and final problem we will cover in Part 4 is what is commonly referred to as “Design Ticketmaster”

There are two major parts to this problem when it is used as an interview question:

1. Consistency under concurrency. Ticketmaster, when seen in this light, is just an extreme version of a service used to book hotels or airline tickets. There just are vastly more seats in a popular show than in a hotel or in an airplane, so the system should handle far greater concurrency, and not crack under pressure.
2. Real-time updates. Savvy candidates, who crush the first part, are then tortured further by the second one: handle people holding on, while being put onto an “online waitlist” by the service. Granted, an online waitlist is not the best product decision for Ticketmaster (”You are in line, your number is 12,345, and if all these people before you refuse to complete their transactions, you’ll be given your chance”). Nonetheless, it is a separate sub-problem that enables probing the candidate’s skills in quite a few more unique ways.

Let’s focus on the first sub-problem. The real-time part is a special case that deserves dedicated attention.

### How to Get Yourself Unstuck – Tip #9:

As usual, we begin from requirements. In fact, it's best to postulate the problem right in the form of requirements! This way we also develop a habit of thinking of system design problems from the grounds of how to solve them, as asking the right questions is at least half of solving them.

Don't worry if you don't know in detail what the Ticketmaster problem is about. In fact, for any problem, if you don't fully understand its statement, jump straight to functional requirements, and clarify them—with your interviewer or with your peers—until they are crystal clear!

## Requirements

### Functional Requirements

- The user can book tickets for a specific show.
- The user can book more than one ticket at the same time.
- The user may book specific seats or, at least, request "N seats together."
- Once the user has selected the seat(s), they are given some amount of time (say, 10 minutes) to pay for them.
- During this period of time, the seats are considered reserved for this user's session.
- Other users should see these seats as temporarilyy unavailable.
- They may still return to the pool of available seats if the user who's "holding" them decides to abort the transaction, or if the user does not pay for the seats in time.
- Also, while most seats are numbered, many shows offer "zones," where N tickets can be sold to a given zone, with no differentiation between these individual tickets.

    - I.e., most seats are individual seats, one person each, but a venue may offer, say, a “dance floor,” to which one hundred tickets can be sold.

Generally speaking, this is what the Ticketmaster problem is about: concurrency, as many users, effectively, can and do “compete” for a small pool of seats available. Of course, the read-to-write ratio is quite high, as most users (more precisely, most user sessions) that look at some concert hall’s available seats map do not immediately proceed to pick seats and purchase them.

The important part is that the Ticketmaster problem does not generally have to deal with user flows outside the very ticket bookings. What goes out of scope of the Ticketmaster problem:

- Searching for shows (location/venue, date/time, etc.).
- Interacting with custom APIs (be it the show-booking APIs and/or the payment APIs).
- Frontend stuff (rendering the venue seat map, supporting web/mobile/tablet native experiences).
- Online ticket validation (i.e., the QR code from the email sent, which can be used as a ticket to enter).

To be frank, if you are asked this problem in an interview, the nice-to-have parts are great to mention, but your best bet is to dive deep into the concurrency-heavy aspect.

### Non-Functional Requirements

- **Consistency!**
    - Don’t sell the same seat twice, of course.
    - Return seats to the pool quickly.
- **Responsiveness**

    - The seat map should reflect the actual occupancy map at all times, with low latency.
    - Keep in mind that the read-to-write ratio can be quite high.
- **Availability**
    - The system should serve end-user requests correctly, even as individual nodes, responsible for the above, die or misbehave.
    - Do not lose the data on the tickets already sold, as well as the currently open purchase sessions.
    - In practice, a good solution here is eventual consistency. That is, if stuff breaks, then:
    - The users can still use the service, in read-only, somewhat stale, mode.
    - They may see the “state of the world” as it was for some moment in the past (say, a minute ago).
    - However, as the user tries to mutate the state, they get an error along the lines of “please come back later.”
    - This “temporary staleness and read-only-ness” is a standard way of preferring consistency over availability when the service has a high read-to-write ratio (i.e., when it is plausible to expect that most users would not even notice this “silent downtime,” during which the system is indeed not up and running, but stale and showing some outdated snapshot.

## Solution

### Here's our proposed solution

The reason Ticketmaster is included in this guide is simple: **Ticketmaster is about the perfect way to illustrate the value of ACID transactions.**

Yes,

- that ACID that is the opposite of BASE,
- that ACID that is famously guaranteed by relational databases, such as Postgres or Spanner,
- that ACID that is the easiest to articulate differentiator between a SQL and a NoSQL database.

So, with no further delay, let's focus on data consistency in the Ticketmaster problem, in the context of high concurrency.

First things first:

- **There are hardly any dependencies between shows.** Even if they share a venue, and even if some two shows take place on the same day, in the vast majority of cases it's safe to assume they are not connected bookings-wise.
- Thus, the system we are building will be **horizontally scalable**: its capacity can grow, linearly, and almost indefinitely, as we keep adding more and more nodes. Simply put, in an extreme case, we can dedicate a separate (replicated!) DB and a separate set of nodes (three will be more than enough) exclusively to handle some important show.
- The major problem is very close to maintaining **transactional guarantees** when it comes to **multiple actors attempting to mutate the same resource at the same time.**
- The above should also be designed such that individual node failures do not cause data loss and, ideally, do not bring the whole system down.

## How to Get Yourself Unstuck – Tip #10:

The best way to reason about the value of consistency is to think of *what could possibly go wrong*.

## What could possibly go wrong in the Ticketmaster system?

**These things could go wrong:**

- Obvious failures: ultimate inconsistencies that result in terrible user experience.
    - "I booked a ticket but it was not issued."
    - "I booked a ticket, but someone else also has a ticket booked for the same seat."
- "Nice to avoid" problems: the functionality that does not render the service useless but hurts users.
    - "I was told during (or even after!) making the payment that the seats that were reserved for me are no longer available" (bad UX).
    - "I was charged twice, as I was made to re-enter my payment details" (support can fix this).
    - The service is showing the seats as unavailable, even though when my friend opens the seat map (or if I open an incognito window) the seats seemingly are available, and the option to book them is there, too.

To be frank, a good design takes care of all these problems, both the obvious failure modes and the "nice to avoid" ones, and it does so naturally, with no extra work on the developer side. The key to proper architecture here, which is what system design interviews are about, is to leverage RDMBS (relational database management systems) transactions.

## SQL Schema Design

As a matter of fact, Ticketmaster is one of the few system design problems where designing SQL tables schema is an essential part of the interview.

For most problems, SQL tables schema is a bit like the API spec. For most problems, SQL tables schema is important, but not too important. If you have more important topics to cover, go on; and if you need to fill time with something not too valuable and information-dense, do that with the SQL schema overview or an API spec.

### Tip

With Ticketmaster, since the gist of the problem is in managing concurrency, an SQL-first design is what interviewers tend to like a lot.

Strictly speaking, there are several points of potential conflict / concurrency / race condition. All of them can be handled nicely with an RDBMS, a SQL database, as a source of truth. These points are:

1. Each unoccupied **seat** can only be "locked" into one order at a time.
2. Each **order** can only be completed (paid for) within a specific time interval.
3. Corner case: If the payment API took too long, the seat lock may have been released by the time the payment went through, in which case the seat may already be claimed as part of another order, and the order should be refunded.
4. Availability + consistency: If the Ticketmaster server goes down between (1) and (2), or between (2) and (3), the freshly restarted server should be able to pick up the state of the system from the point where the first server dropped the ball.

Here, "order" (as in the "shopping cart in progress") is a good way to refer to the user session of completing the payment for a set of seats they have selected.

### Exactly once seats to orders

(1) is the canonical case of a SQL transaction: **UPDATE IF**. SysDesign-wise, this statement alone is sufficient. Refer to your favorite SQL tutorial for more details. In practice, at mid-level / senior interviews, your interviewer will not judge you harshly if you make it clear you understand that an SQL engine can handle this concurrency issue for you; it is not critical to know the exact syntax.

### Timestamps instead of cleanup jobs

A neat trick for (2) is to avoid timers and overall timed or scheduled (”cron”) “cleanup” jobs when you don’t need them. Instead, just write the “lock expiration timestamp” into the respective column. The “lock expiration timestamp” is simply the current time, at the moment of transaction, plus the fixed delta (5 or 10 minutes).

You probably want to make it 5.5 or 10.5 minutes, not just five minutes sharp or ten minutes sharp, to be nice to your users; final seconds ticking down is a negative user experience, and the payments API may also take several seconds to respond.

In this design, the “is seat available” condition, on the SQL level, is not just “seat is not booked yet, and seat is not part of an open order,” but “seat is not booked yet, there is no order that is not yet expired of which this seat is part of.” The last condition may be easier to understand if it’s phrased as “there is no active order created in the past five/ten minutes that contains this seat,” but it’s a good habit to add the expiration time while writing to the DB, not subtract it while querying.

### Keeping the payments subsystem at bay

The corner case (3) is very similar to (2).

We just give the payments API some “time window” within which it is supposed to respond. We can’t wait forever, although this really is the corner case.

Most large-scale products would have several payment providers to choose from, hidden behind some API gateway, so that a different team would be responsible for making sure the response SLA from this payment provider is right.

### CAP, anyone?

And for (4), a savvy reader, as well as a savvy interviewer, would immediately ask: So, if you claim to have both availability and consistency, you're sacrificing partition tolerance, right? Yes, this is absolutely true from the standpoint of the CAP theorem. Such an argument is rather theoretical though. In practice, it is a **good thing** that our system scales horizontally… as it gives us the indulgence to ignore the CAP theorem altogether!

### Outlaw idea

If your system shards horizontally, just ignore the "P" part of CAP while designing, and focus 100% on "C" and "A" parts. Just mention that your data has "no inner dependencies," that you plan to "replicate your DBs in a leader-follower fashion," and your service will be in great shape in production.

Of course, our databases have replicas, and, broadly speaking, network split is a case of one or several of our nodes becoming unavailable.

But when our nodes become unavailable, we just take them out of rotation and spin up the same service from the backup DB shard.

Admittedly, the above is far more difficult than it sounds. But the very argument holds true. Without loss of generality, consider the "one show one DB" design. This DB would be leader-follower replicated, likely to one or two replicas. If a DB is down, we can, automagically, take one of these replicas, promote it to a leader, and restart the system, as if nothing happened.

This would require synchronous DB replication, so that whatever tickets we have already sold are correctly marked as such. But we want this anyway! Otherwise, moving from one node to another would be a cumbersome process that requires

reconciliation, collating data from different databases (seats, completed payments, issued tickets, etc.)

When outlining the above, don't forget to mention that it is important for the times on these DB instances to be somewhat close to each other. No need to be sub-sub-second synchronized; in practice, it's enough to be within several seconds, give or take. As long as we're conservative with timeouts, and as long as the "flipping" from an unhealthy node+DB pair to a healthy one takes more than these few seconds, no inconsistencies will be introduced.

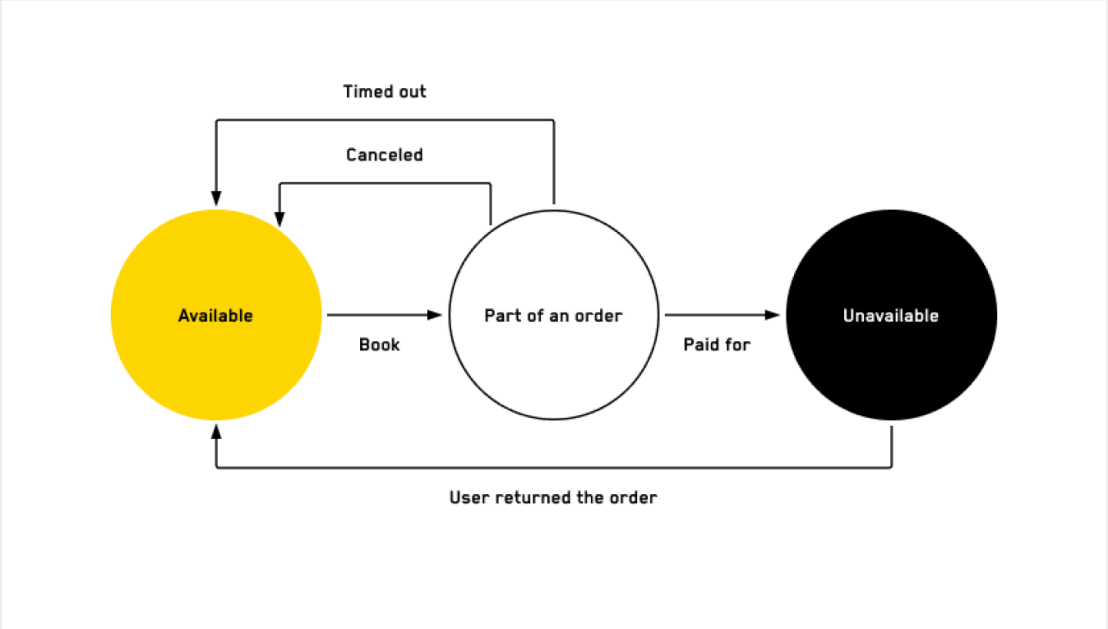
That's about it.

## The Real-Time Part

Rendering the seat map, which the users see when they land on the page of a particular show, is just about maintaining a materialized view of the database.

If we are talking about a super-popular show, with thousands and thousands of visitors, we may just re-generate a snapshot of this view every second or so and serve it off some passive storage (or even from a CDN with a quick-to-expire caching policy).

And, especially if your interviewer is picky, you can describe a pub-sub solution, so that each and every viewer of the show page gets instantaneous updates while a seat, or several seats, goes through its lifecycle journey. This journey can be visualized as a simple Markov process:




There are five arrows on this diagram:

- **“Booked”**: When a seat goes from “green” to “yellow,” from available to “someone’s trying to book it now.”
- **“Canceled”**: The person who was “holding” the order to book this seat (possibly, among others) has explicitly clicked “cancel.”
- **“Timed out”**: The person who was “holding” the order to book this seat (possibly, among others) did not complete the payment in time, and, thus, the seat(s) is/are back to the pool of the available ones.
- **“Paid for”**: The order, which this seat was part of, was paid for, and the seat goes from “yellow” to “gray”—no longer available for purchase.
- **“User returned the order”**: The person, who previously had successfully booked and paid for this seat, has refunded their order, which returns the seat to the pool of seats available for purchase.

Each of these five orders can yield an event that makes it into the PubSub bus. Each person, who is looking at the web page with the seats map, can subscribe to the updates to this bus, and the updates would be streamed to them, in a “push, not pull” way, likely via a WebSocket. Thus, each person would be looking at an accurate

picture of the seat map, colored into green, yellow, and gray, for “available,” “part of an order,” and “unavailable,” respectively.

## Bonus

To earn bonus points from your interviewer, consider…

For bonus points, consider the show for which most orders are bulk orders (e.g., if an average number of seats per order is not one point something, but over ten, or even over a hundred seats). In this formulation of a problem, it’s probably better for the PubSub bus to operate not on individual seats level, but on orders level. The very mapping of order ID → list of seats in this order can then be exposed via a simple RESTful API. To maintain low latency of updates, as the Web page with the seat map is open, the code can request these lists of seats that correspond to every order that is currently “yellow.” This way, when each of the { “canceled,” “timed out,” “paid for” } events arrives, the client-side code knows right away the status for which group of seats should be updated.

For more bonus points (entering the Staff level territory), the contents of the above PubSub bus can be viewed as the source of truth, and the overall architecture can be based on this very log of journaled events. This design pattern is increasingly popular these days, and its official name is Event Sourcing. The beauty of this pattern is that as long as the event log (the “journal”) can be replayed (is “replayable”), we can think of materialized views of the available seat map at a certain point in time as a snapshot of this log at certain index (or “offset”) in this log. For instance, we may have a snapshot as of midnight, November the 1st, corresponding to the first one thousand events. The next snapshot can be generated an hour later, or another thousand events later, whichever arrives sooner. This way, the client, when loading the page, can retrieve the latest snapshot from some cold storage, and then request to subscribe to updates from the offset corresponding to these snapshots.

Snapshots are immutable, and, as we covered above, immutable data is quite easy to deal with. And the "delta," which the client should receive and amend to the snapshot, is guaranteed to only contain a small number of records—because if the number of records is large, a newer snapshot would have been already available. This Event Sourcing paradigm can also be viewed as an extension of the leader-followed DB replication, as the "DB state," replicated from a leader to the follower(s) is merely the contents of this replayable event log (aka the journal).

## Part 4: Outro

*Fin*. This is the end, my friend.

If you read all 4 parts, congratulations are in order! You learned high-level ideas to strategically approach system design interviews, the 15 fundamental system design concepts, a 3-step framework to solve any system design problem, and you watched us design simple systems from scratch. Plus, you picked up a bunch of useful tips and tricks along the way.

We want to thank you for reading this guide and for entrusting us with your precious time. We hope that you agree that you'll be a better interview candidate and a better engineer because you took the time to experience this guide. Here's a final tip: Give yourself some more time: for the ideas to integrate, to take the concepts further, and to practice on your own. Thank you for joining us on this journey!

Here is a delightful picture of the team that made the video Two Ex-Google System Design Experts Compete: Who will design the better system?

**At this point we want to do three things: Tell you what's on the horizon for us, give you a way to interact with us, and provide you some additional resources in case you're still hungry to learn more.**

## What's next for the team that made this guide

You! And hearing what you thought of this guide. What elements were the most useful/actionable? Which parts could benefit from additional examples or clarity? Once we grok that, we'll know exactly what to do. The community might give us harsh critical feedback, shout from the rooftops that they'd love to see us make more stuff like this, or (ideally) some combination of both.

Some possibilities we've discussed:

- Make this an ongoing thing. Maybe it's a newsletter, maybe it's a meetup, maybe it's neither of those formats, but the idea would be to ensure that we consistently release stuff like this.
- Make a V2 of this guide that takes your feedback into account.
- Make a system design guide for other levels (if we did this, most likely we'd make a guide for an "above L5" or "senior plus" audience).
- Make cool live system design events (though we would want to hear from you about what kind of events you'd most like to see).

Other things we'd like to hear about:

- People who want to make content with us (could be written, video, or both)