

System Design for Interviews

Juliane Marubayashi

2022-08-01

Contents

1	Basics of System Design	3
1.1	Monolithic architecture	3
1.1.1	Downsides	3
1.2	Microservice architecture	4
1.2.1	Benefits of Microservice	4
1.3	Monolithic vs Microservice	4
1.3.1	Latency	4
1.3.2	Backward Compatibility	4
1.3.3	Logging	5
1.3.4	Service Discovery	5
1.3.5	Other smaller costs	5
1.4	Conclusion	5
1.5	Questions	5
2	Interservice Communication	5
2.1	Analysing performance	6
2.2	Modes of communication	6
2.3	Synchronous	6
2.3.1	Disadvantages	7
2.4	Asynchronous	7
2.5	Best of both worlds	7
2.6	Message Queue	7
2.6.1	Advantages	8
2.7	Building queues	8
3	Protocols for communication	8
3.1	HTTP	9
3.1.1	WebSocket	10
3.1.2	Conclusion	11

Attention before reading: this document is not original. It is deeply based on the resources. Thus, I don't have any credit on the biggest part of this text.

1 Basics of System Design

1.1 Monolithic architecture

In the beginning of the internet, usually one single **codebase**. Usually a codebase is maintained by a version control system, such as GitHub and GitLab.

The **problem** with this **monolithic** approach is that it was:

- Hard to maintain;
- Hard to improve;
- On top of all, there were multiple people working with the same code. Which is a receipt to disaster.

To solve this issue, we started using more **efficient** design patterns and various OOP concepts to streamline the code and make it more modular.

Once I went in a discussion of why is OOP important? Do we really need to use it? Does it make the code more complicated? The point is that, probably the teams that created a certain package used OOP to avoid the monolithic approach and make it easier to subdivide tasks. The code on this way would be more modular and it would be easier to split tasks.

Nowadays, with Web 2.0, since a website supports many types of services, using OOPS and design patterns are the new way of building monolithic apps.

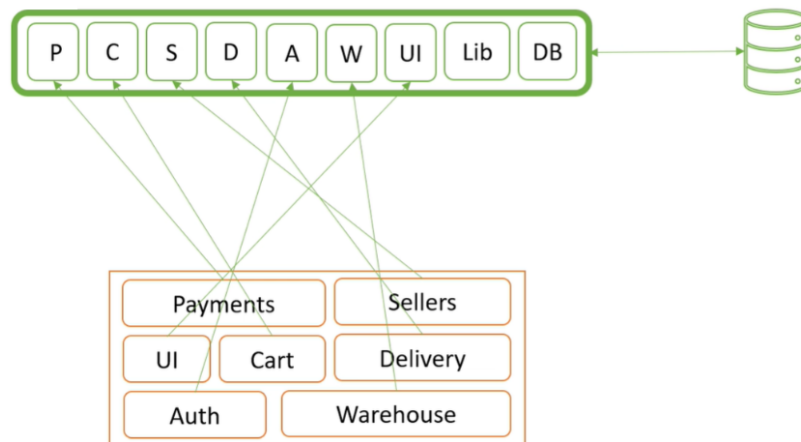


Figure 1: Monolithic app with OOPS

1.1.1 Downsides

As the team grows, some problems start to come up with this approach as well:

- Another big defect of the monolithic approach is that we are bound to the technology stack. So if we build a code with Java and Spring, but we need to create a Machine Learning feature in Python. How can you do this? The solution would be creating a standalone app to handle this feature. There isn't really a problem with creating a standalone feature, but you are defeating the meaning and purpose of a monolithic app.
- Another problem is that it's really easy to break things with this type of architecture, because you are never gonna be entirely up-to-date of what each person did in the code. Thus, if you change some logic, you might break someone's else feature. Sure there're some test codes and the second SOLID principle, which is open-closed. But sometimes, even these are not enough.

- Another major issue is **scalability**. Let's suppose it's Black Friday: the sellings are done constantly and you might be afraid that the system crashes. At this moment is necessary to scale the payments and the cart module. The problem with that, is that the notification and the Warehouse doesn't need to be modify, but once you changes the Payments and the Cart modules, you gonna need to deploy the code again and at this moment the Notification and the Warehouse will be idle. A big enterprise such as Amazon would loose a lot of money.
- The bigger the code is, more it takes to be deployed.
- If there's a leak in some part of the system, it's very likely that another part be also leaked. Even a small leak in the Warehouse might compromise another module.

A person started to discuss with me that, the more a code grows, more it becomes difficult to understand. Mainly when it is done with OOP. Actually, this is not a problem of using OOP or not, it's a problem of monolithic approaches.

The solution of all this problems is to use a Microservice architecture.

1.2 Microservice architecture

The main idea is to break down the application into logical components, so that this components becomes a service on their own. The microservice is the only source of truth of a feature group. Now each service will communicate with each other via a set of API calls like REST APIs or Remote Procedure calls.

1.2.1 Benefits of Microservice

- Now we are not bounded to a single technology stack;
- One team can do your work well without worrying about break someone's else feature;
- One microservice can be scaled without affecting another entire module;
- Iterating deploying and testing is a lot easier.
- The engineers will be working on smaller code bases.

1.3 Monolithic vs Microservice

Microservices have a lot of advantages, but it doesn't mean that we should always use Microservices in one application.

1.3.1 Latency

- Of course functions are faster than API calls, thus the Monolithic architecture is more efficient when we talk about communication between modules/ services. But this delay is not noticed by the user, but in critical services, this might be one aspect to be considered.
- With network calls comes the possibility of network fail. So it increases the complexity of the error handling and retries.

1.3.2 Backward Compatibility

- If a service makes a change for a mandatory parameter, another service that depends on this one, might break. As an personal opinion, this is not a big issue, because if a service creates a new version on every release, than the depending service will not crash. Besides, the same thing might happen in a Monolithic approach: someone might change the parameters of a function, the difference is that the IDE might recognize the error in compilation time.
- To solve this problem some tests are necessary.

1.3.3 Logging

- If you need to trace logs for a user in monoliths, it is easy to do so as they are all in the same place. But in microservices, for each request from the user there will be multiple service calls.
- A way would be storing in each service its individual log. But tracing exactly what happened becomes more difficult.
- A better approach would be to store all the logs in just one central place where they can be queried. This is a cost, since you need to build a **Log Aggregation System** or buy a third-part system license.

1.3.4 Service Discovery

- If you are calling multiple services, we need a way to identify where they are hosted.
- We could of course hard code this, but that makes it little difficult to change, especially if you have a lot of microservices.

1.3.5 Other smaller costs

- We also need a configuration manager system to push configurations into all the nodes of a particular service.
- Even though auto scaling is a benefit of microservices, doing it in a controlled manner needs to be done by a system as well.
- In monoliths, since all the code and data are in one place, access control is straightforward. With multiple microservices, access control, security policies, etc., will need to be done for each service repeatedly.
- There needs to be a managing infrastructure for all the services that need to asynchronously talk to each other, something like a centralized Kafka deployment, which becomes an additional cost.

So, there're a lot of components just running to support the microservice architecture. For small applications, **the cost of maintanning this architecture is higher than creating the application itself**. However, thinking in a real world scenario, there would be hundrer of services working together. The cost of creating the support is negligible compared to the development of the architecture.

1.4 Conclusion

If you are working on small business or small team and have limited set of features, monolithic might be a better approach. However, if you are working on a huge product with hundreds of microservices, the maintenance cost of microservices will be worthwhile.

1.5 Questions

- What is a monolithic architecture?
- What are the disadvantages of a monolithic architecture?
- What is a microservice architecture?
- What are the advantages of a microservice architecture?
- Compare the monolithic and microservice architecture.
- When is it better to use a monolithic architecture? And microservice?
- Why is it interesting to use OOP?

2 Interservice Communication

This section is reserved to explain how microservices can communicate between each other.

2.1 Analysing performance

: When analysing these kind of systems we need to consider the following components:

- The latency; [1]
- The complexity; [2]
- The scenarios of crashing; [3]
- Maintenance; [4]
- Scalability; [5]
- Stability; [6]
- Cost. [7]

2.2 Modes of communication

There're two modes of communication:

- **Synchronous:** When the requester waits for the response.
- **Asynchronous:** It's the fire and forget approach. It launches the request and it doesn't wait the response. Eventually it will arrive.

2.3 Synchronous

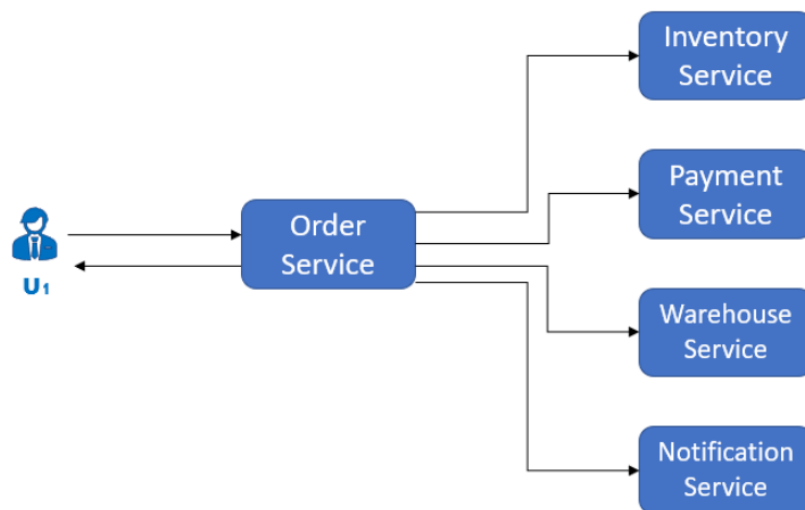


Figure 2: Synchronous scenario

Consider the occasion where the user **U1** wants to buy something from Amazon.

The request will be send the **Order service** that will control the operation. First it will ask the **Inventory Service** if the product is available. In case of yes it will call the **Payment service** to perform the payment. Then a request will be sent to the **Warehouse service** and if everything is ok the **Notification service** will send an email message to the user, saying that the product was bought with success or not.

This is the best scenario that can occur, but what if the **Payment service** fails? We should of course stop the processing. But should we stop it when the **Notification service** fails?

2.3.1 Disadvantages

The point is that the synchronous approach has some disadvantages:

- The code can become very **complex**, since we need to handle all the possibilities of failure scenarios. [3]
- Due to the **complexity**, it needs high maintenance. [4]
- It has a very high latency as the user doesn't get notified until a request has been sent and received from the other services. [1]
- The system is tightly coupled, and any failure will have cascading effects across the board. [2]

2.4 Asynchronous

Now let's analyse the same scenario, but with a asynchronous approach.

Creating a full asynchronous approach here would be a big mess! Imagine: we send a request to the **Inventory service**, barely the request is sent we start to send another request to the **Payment service** and so on. But what if the payment service is unavailable and it takes a while to send this response? Will the user receive a notification saying that the purchase was done with success when it actually wasn't. At this specific part of the code we can use a synchronous communication and in the warehouse service we can use asynchronous, since we are just notifying the warehouse about the purchase.

2.5 Best of both worlds

The hybrid approach is what we have just talked about. Is using the asynchronous and synchronous approach in one single communication system.

The mandatory requests are done with synchronous communication and the rest can be done with asynchronous: we send an synchronous message to the **Inventory service** subtracting the quantity of products to be bought. If the service returns an OK response we proceed by making a request to the **Payment service**. If the service respond with an OK, than we can make an asynchronous communication with the **Warehouse service** and **Notification service**. Otherwise, we just needed to revert the alterations made in the **Inventory**.

This seems ok by now, but what if the **Warehouse service** fails? We would loose the details of the purchase. That's why in this case we would use the **Message Queue**.

2.6 Message Queue

The message queues are **highly tolerant to faults**. It has some **Publishers** and some **Subscribers**. The publishers publish messages of a certain topic and the subscribers reads the message that they're subscribed to.

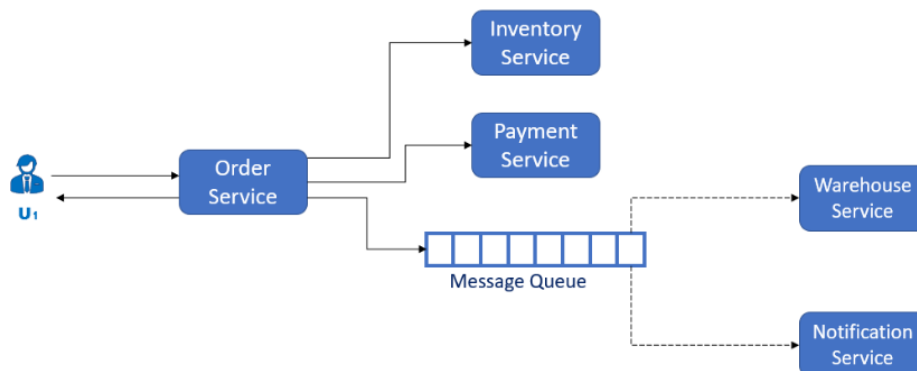


Figure 3: Message queue

Now instead of `Order service` make a purely asynchronous call to the `Notification` and `Warehouse` services, it will send a message to the queue. If one of the system is down, the messages will remain on the queue until the service is back up and ready to receive messages again. On this way, none of the data will get lost.

2.6.1 Advantages

- **Increase stability** [6]: Even if a system fails, it will not lose the messages. On this way the system becomes more stable.
- **Enhanced scaling** [5]: When the number of request increases the asynchronous services can continue to work in their own pace. Only the synchronous services need to be scaled by adding more hardware. *My personal opinion is that if the number of requests increase exponentially, then the services that depends on a queue, might also have to be scaled, since the queue can increase to a size, where memory may start to be discarded.*
- **Reduced cost** [1] and [7]: By using queues, not all the services need to perform synchronous communication. Also, at some level it's not necessary to add extra hardware to the other services. In some aspects this is a win for the wallet.
- **Reduced complexity** [2]: Since we don't need to handle cascade effects, the code complexity is reduced.

2.7 Building queues

There're some software that provides this type of communication by queues:

- Kafka
- RabbitMQ
- ActiveMQ
- ZeroMQ

3 Protocols for communication

This section is separated to explain how clients can communicate with the services.

Let's first take a look on an e-commerce platform design:

Say our user searches for a hat. Now the client will send a request to the server to search for hats and the server will respond with search results.

If a user wants to buy a hat, the client will send a request to the server to add the hat to the user's cart, and the server will respond with a confirmation.

If the user is happy with the items in their cart, on their action, the client will send the request to the server to place the order, and again, the server will respond with a confirmation.

In a real-world scenario, rather than talking about one specific server, the client's request will be sent to a data center, where any server can pick the request. However, irrespective of which server receives the request, the response will be the same. Based on this flow, we can draw the following conclusions about this architecture:

- It is client-driven. Only on the user's button click will the client send the requests to the server, and the server will only respond to these requests.
- It's a **request response model**. For every request the server will respond with some information or simple confirmation.
- There are occasional requests from clients, only one request every few seconds based on the user's actions i.e from the client-side it is a **low throughput system**.
- It is a **stateless** system, i.e irrespective to which server is responding the answer will be always the same.

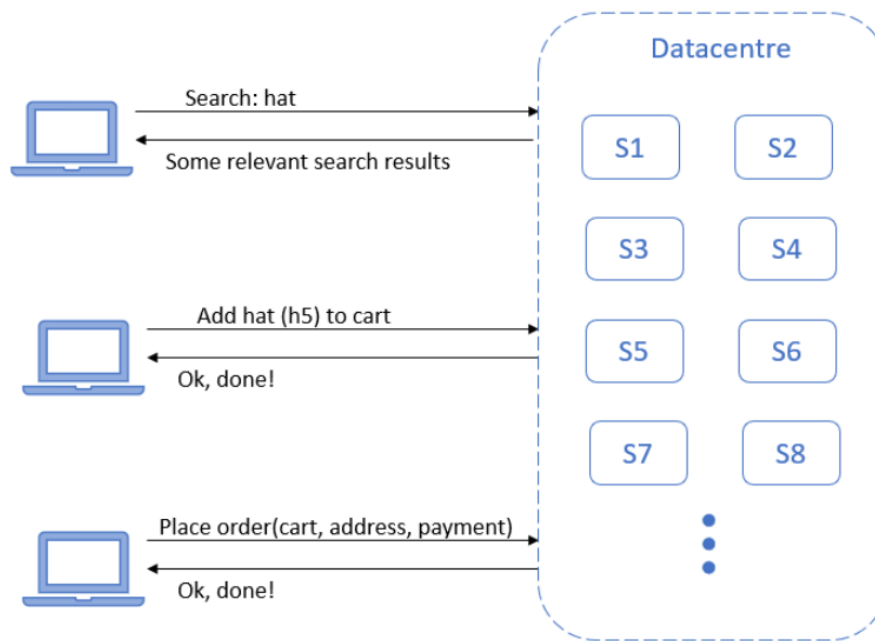


Figure 4: e-commerce platform design

3.1 HTTP

These requirements make this a perfect use case for HTTP(s) protocol. Although these days, most architectures on HTTP have moved to HTTPS, which is a more secure version of HTTP as it prevents man-in-the-middle attacks.

Now, when we are using HTTP, REST is usually the best API standard to follow as it is very widely used and very user friendly.

Let us look at an example for a REST request and response:

```
Request:
Method: GET
URL: https://www.twitter.com/user/{id}

Response:
Status: 200 OK
Headers: <...>
Body: {
  "userId": 1,
  "Email": "someone@example.com"
}
```

The client makes a request to twitter.com over HTTPS to get information about a user with an id. In response, the server sends a success status code along with the user's user id and email. As you can see, REST API standard is pretty much self-documenting, which adds to its user friendliness.

Now let us look at an example of a chat application.

We know that the **HTTP request is a client-driven protocol**, so the server cannot initiate any contact with the client. It can only respond to the client upon receiving a request. So when U1 sends a message to U2 via chat server, U2 doesn't receive the message until it asks the server to share any pending messages. **This leads to a delay when receiving message. It also makes the notification service impossible.**

A naive solution would be U2 sending repeated requests to the server in the hope of receiving a message. But this consume too much resources and **puts a huge load on the chat server as it will receive a huge number of requests from all of its clients!**

Another solution wouldl **polling**. The server could wait a few minutes before responding to U2,

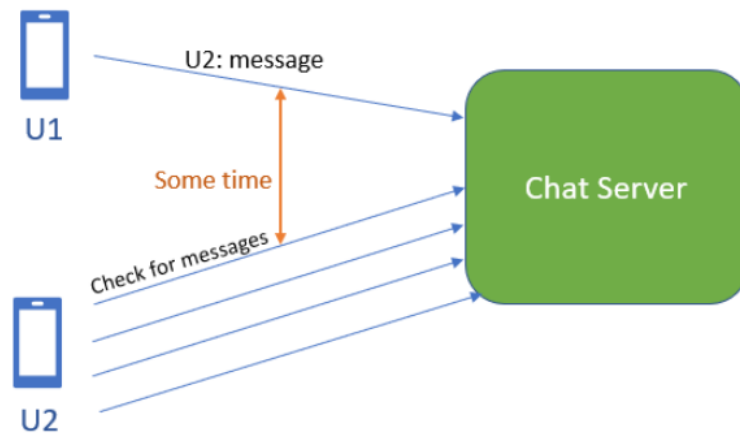


Figure 5: Chat app

thus increasing the chance of a message being received before responding. But it's also not a very good solution.

The best approach would be if the server could send a notification to the user every time there is a message. For this, we use a protocol called WebSocket.

3.1.1 WebSocket

A WebSocket connection **is a persistent connection**. It is also a **bidirectional protocol**, where communication can be initiated by the client or the server as long as there is an open connection. **It is optimized for high-frequency communication**.

Let's look at how our chat application would work in the case of WebSocket protocol.

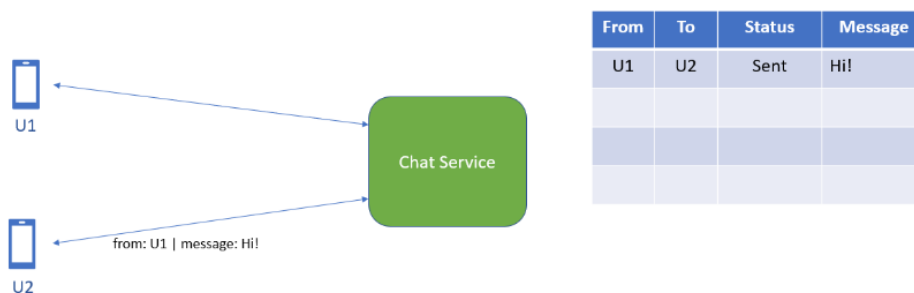


Figure 6: Chat app with websocket

First, U1 and U2 will establish HTTP connections with the chat server, which are then upgraded to a WebSocket connection. When U1 sends a message for U2 via the chat server, it will store the message along with its status, **RECEIVED**, let's say.

The chat server, if it has an open connection with U2, will then send the message to U2 and update the status to **SENT**. If U2 was not online and there was no open connection between U2 and the server, the messages will be saved until U2 comes online and requests the server to send all pending messages. The server will send all messages with the status **RECEIVED** and update the status to **SENT**.

As you can see, with this approach we have:

- Reduced the latency, since the server can simply send the messages over an open connection
- Saved on CPU and bandwidth, as the client doesn't need to unnecessarily send requests to the server and the server is not under unnecessary load.
- Provided better user experience.

Even with the benefits, there is a high cost to using WebSockets; that is the cost of maintaining a persistent connection with millions of users.

3.1.2 Conclusion

So how do we decide whether to use HTTP or WebSocket? Do we always go for Websocket then? Well, not really, as **WebSocket is much more expensive than HTTP**. We can safely say, if the communication between client and server is at a lower throughput on the client-side, HTTP is the way to go. If the communication is always client-driven, WebSocket is not needed. Also, if you are on a tight budget, HTTP may be the better choice.

On the other hand, if the communication from the client is at a higher throughput, WebSocket may be a better option. If the communication can be driven by both client and server, WebSocket is the way to go. Although here comes the tradeoff between cost and performance. We must decide if the optimization is really worth the huge cost of maintaining persistent connections with so many users.