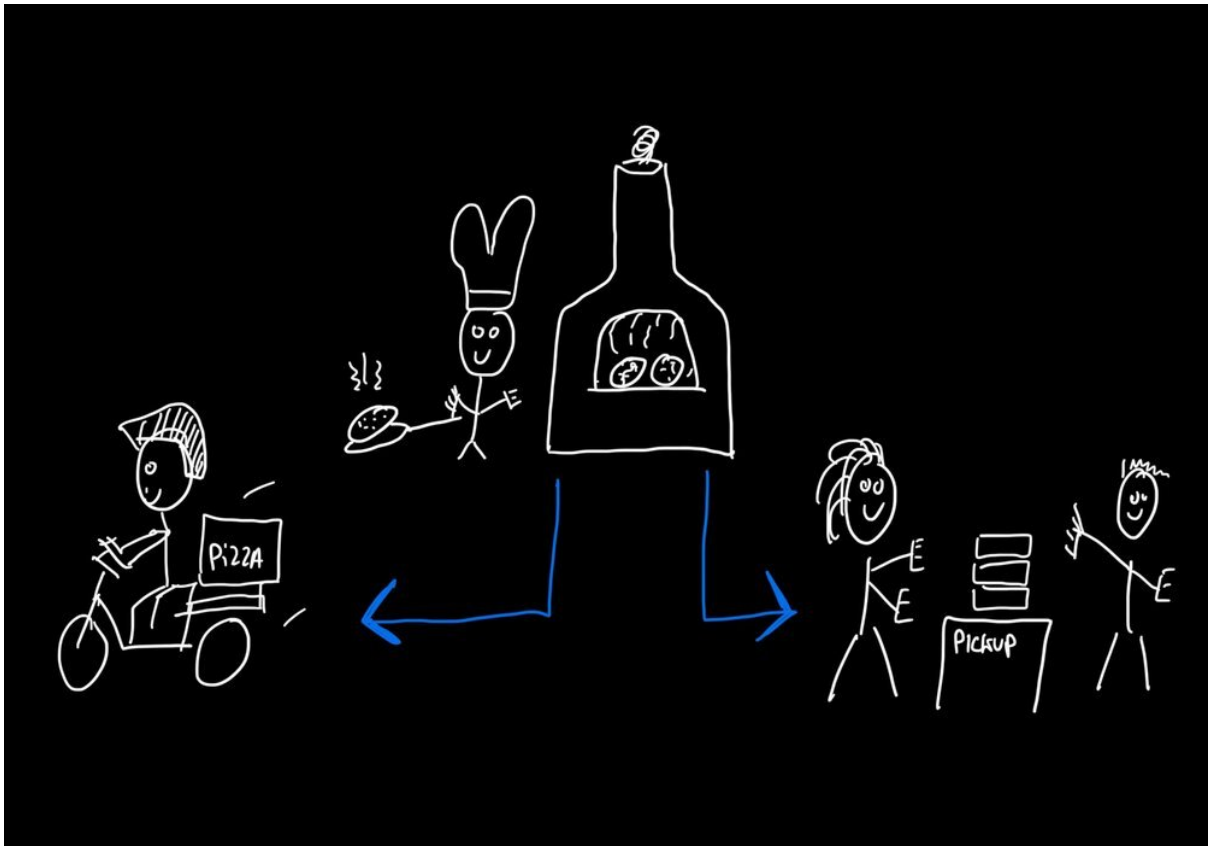
Designing a multi dimensional capacity queue

Managing kitchen-, delivery- and pickup capacity

Willem L. Middelkoop
Dec. 2, 2020



This month I needed to create additional dimensions to a capacity queue mechanism. The food ordering app that I created needed to be able to restrict capacity based on the number of orders, the contents in individual orders and the dispatch type (takeaway/delivery). Read along to find out how I used a Lambda Architecture to do this.

Restricting capacity

Although you may not realise this, one of the most powerful features of the food ordering app that I created is actually to be able to restrict the number of orders. This sounds counterintuitive, but by limiting the number of orders that can be placed, my customers (the restaurants and store owners that use my app) can ensure that orders get prepared and delivered in a good manner.

Three pizza complexity

Say, you want to order three pizzas. The food app then needs to know:

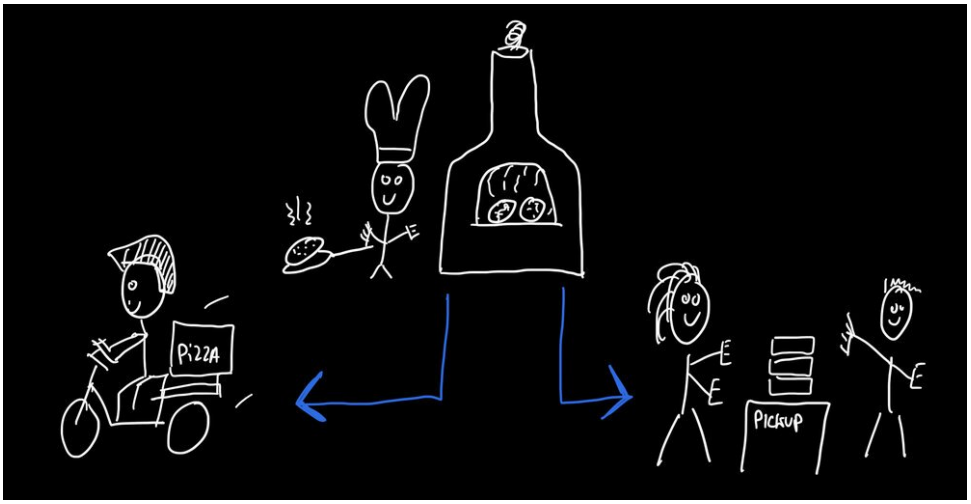
- Do you want to **pick them up** or have them **delivered**?
- At **what time?** when will they need to go in the oven?
- Is is there **enough space available** for *three* pizzas **in the oven** *at that time*?
- If you have chosen pickup: **can we safely receive you** at that time?
- If you have chosen delivery: **is there a delivery boy/girl available** to bring you your pizzas?
- **Yes to all of this?** Then **please pay** while we **keep your pizzas scheduled** in the oven and delivery or pickup. Failed to pay? Second thoughts? Then we "undo" all the scheduling and wait for somebody else to order some pizzas!

Eating pizza during development is how you "live the algorithm"

Three capacity dimensions

Especially around "dinner o'clock" there is a risk that too many people order food at once. While the app can handle the extra traffic thanks to its highly scalable architecture, physical constrains remain, such as:

- **kitchen capacity**: the number of dishes you can prepare at once
- **pickup capacity**: the number of people you can safely receive at once with respect to the COVID-19 restrictions
- **delivery capacity**: the number of orders you can deliver, taking into account traffic, distance and routing



Three capacity constraints: kitchen / delivery / pickup

Distinct yet interdependent

These three different types of constraints are interdependent. For instance, it is possible that one large pickup order can take up all the kitchen capacity, leading to unavailability for delivery, too. Smaller orders could still be accepted to fill up kitchen capacity, while there may no longer be enough capacity available for large orders. This means that you need a mechanism that looks at *all* the metrics combined to determine the availability for new orders, including the contents of an unplaced order (e.g. what's in your shopping cart).

Differences in timespan granularity

It is common to define a physical constraint by using a timespan, for instance *"the kitchen is capable of producing 10 dishes per 15 minutes"*. It becomes challenging when you define other physical constraints using different timespan lengths. While the kitchen could be limited per 15 minutes, the delivery capacity could be using a longer timespan (e.g. 3 deliveries per 30 minutes). Somehow the mechanism should be capable of dealing with these differences in timespan granularity.

Ordering in advance

Adding to the challenge is the option for people to place their orders in advance. People can select a time slot in the future and have the required capacity reserved. Sometimes I see people place their order days in advance to be sure that they get their food delivered on a specific day and time. This means that there are (at least) two times associated with each order: order time and delivery time.

Reserving and releasing capacity

Most business owners require online payments for online orders, this is to ensure that all orders are legit and no capacity (or food) is wasted on no-shows or ghost orders. The problem with online payments is that not all transactions are successful. Sometimes a credit or debit payment is rejected (e.g. due to insufficient funds). Sometimes people simply close the app before completing the order. The system should reserve the required

capacity for a given order, while it allows people to perform the online payment, and it should release the reserved capacity in case the order is abandoned or when the payment fails.

High volume, high speed

Last, but not least, the mechanism should be reliable and performant. Order availability should be determined within mere milliseconds while dealing with heavy concurrency as there are many different clients connected to the food ordering app during rush hour. When somebody places an order, it affects the availability for all other (potential) orders. Not everybody will be connected reliably as some people use the app on phones with poor reception or wifi. The system must work well under varying conditions with many different connected people.

Lambda Architecture

The Lambda Architecture is one way to deal with handling large quantities of data by using both stream processing and batch processing methods. A lambda architecture depends on a data model with an append-only, immutable datasource, often comprised of timestamped events.

Immutable data

This is a very important foundation to the mechanism: all data is timestamped and *never* changes. Instead of mutating existing records, new records are simply added to overwrite the existing records. This allows for high concurrency and distributed algorithms, as synchronisation becomes simpler as one only has to look for new data (instead of both new and updated!).

Realtime / speed layer

The stream processing, or realtime/speed layer, in a lambda architecture is designed to provide super fast answers to realtime data needs. It maintains views and metrics based on realtime processing of events.

Precomputation layer

The batch processing, or precomputation layer, provides an complete and accurate basis of all the data in the system. As processing large quantities of data can take up time to process, the batch processing is not fast enough to do everything instantly. This is why the two layers work in tandem in a lambda architecture.

Implementation

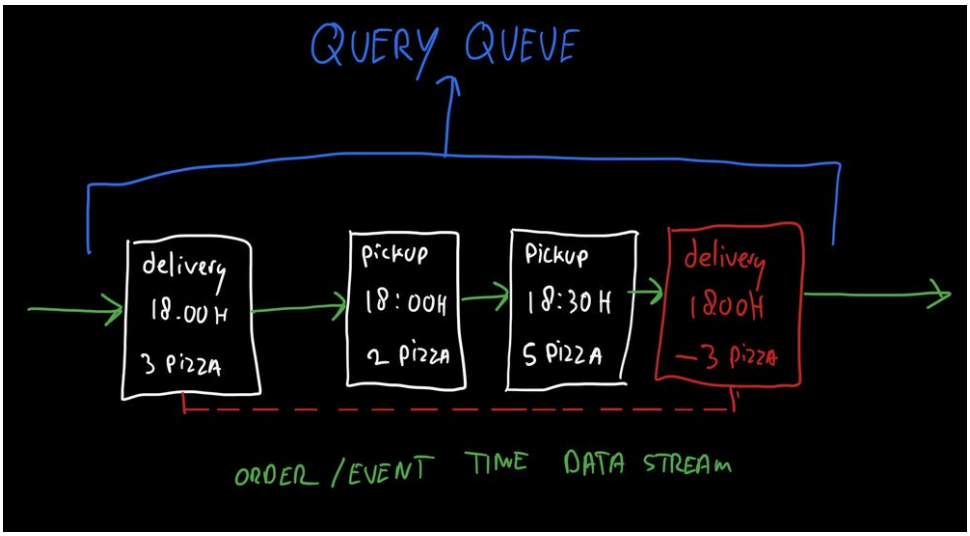
The backend servers that handle food orders from the app work with a data stream of timestamped events. In essence: every order is an event in the system. Each order is stored with its order date/time, dispatch date/time (scheduled delivery or pickup time), customer details and order contents (number of dishes, etc).

Reserving and releasing capacity

If payment fails or if an order is abandoned, a ”negative” order is added to the data stream. It neutralises any effect on capacity constraints by its negative metrics. For example: to cancel an order of three pizzas, a negative order of ”minus three pizzas” is created in the data stream with the same dispatch timestamp. This is an elegant way to make sure capacity is both reserved and released.

Multidimensional realtime metrics

Instead of abstracting capacity into counters, the backend server uses the *real* stream data to anwser queries for realtime metrics. This enables the system to deal with differences in timespan granularity.



An event stream of orders is the heart of the mechanism, queries are fulfilled by looking at this stream: how many pizza's will be ready at 18:00hrs?

The first time a query is received for a given metric, the system loads historic data through a batch mechanism to initialise the model that it can then keep up to date in realtime. This metric is kept up to date by looking at all new orders coming into the system for as long as queries for a given metric come. Once the data is no longer requested the model is unloaded. If it is needed again, the proces repeats.

The great thing of this approach is that most queries are answered within mere milliseconds as they are directly read from RAM memory with no database or disk interaction. This enables ultra high performance on a ”need to have” basis. This works well with the food ordering app as not *all* timespans are equally populair.

The food ordering app repeatingly queries a central capacity server that maintains the queue metrics. It then filters out available time slots by looking at the actual things that a person is selecting for his or her order. Selecting a (likely) available time slot therefore happens client side, reducing the load on the backend. Once an order is placed, the capacity is reserved by the backend by updating the central queue metrics, blocking other people from claiming the same spot in the queue.

Conclusion

Queue mechanisms can be challenging when you deal with multiple metrics that control the speed. You should really take the time to model it right, think of the kind of data you *really* need to perform queue related checks.

Now you know that ordering three pizza's isn't quite as simple! Guaranteeing that they can be served hot and on time is a major technical challenge - in addition of being a good chef. Happy for you that *I* don't focus on the latter, ha!