A Channel Oriented Aproach to the Restaurant of Gophers

Filipe Colla David, *Student, FCT-UNL* - Victor Ditadi Perdigão, *Student, FCT-UNL*

1. Introduction

A *Restaurant of Gophers* is a simulation of the functioning of a restaurant. It aims to simulate an environment where multiple *Chefs, Waiters* and *Customers* interact concurrently.

2. Architecture

The system architecture consists of four main components:

1. *Customers.*
2. *Chefs.*
3. *Waiters.*
4. Operation.

The *Customers, Chefs* and *Waiters* are also described as Entities in this document, this definition will expanded in Section 3. Entities. *Chefs* and *Waiters* will also be referenced as *Employees*. These entities exchange an *Order* object that will also be further detailed in Section 3. Entities.

These components interact through channels and goroutines to simulate the restaurant's operations.

3. Entities and Objects

Each entity or object is represented by a *Struct*, and each definition can be found in the files named after each entity/object.

3.1. *Order*

The *Order* object represents the object beeing passed around between *Chefs, Waiters* and *Customers*.

- *ID* - An unique ID
- *Name* - Name of the order
- *TimeToPrepare* - The time, in seconds, it takes to prepare this order.
- *Customer* - Pointer to the customer who made this specific order.

3.2. *Customer*

The *Customer* entity represents a customer that will place an order, wait for a determined amount of time and receive the order requested once the *Waiter* delivers it.

Struct definition:

- *ID* - Unique ID.
- *TimeToWait* - The max amount of time a customer is willing to wait before decides the cancel the order (context timeout).
- *Order* - Pointer to a new order.

3.3. Chef

The *Chef* entity represents a chef that will receive an order from a *Customer* and after a pre-determined amount of time, will hand over the order to the *Waiter*.

Struct definition:

- *ID* - Unique ID.
- *Name* - Name of the Chef
- *ActiveOrder* - Pointer to an order, which changes if he finishes the order or if the order is canceled by the customer.

3.4. Waiter

The *Waiter* entity represents a waiter that will receive an order from *Chef* and deliver to a *Customer*.

Struct definition:

- *Waiter* - Unique ID.
- *Name* - Name of the *Waiter*.
- *Order* - Pointer to an order, which changes if he delivers the order or if the order is canceled by the customer.

4. Behaviour

The Operation (*restaurant/operation.go)* component is responsible for opening the restaurant with *operationTimout* as a parameter that defines the ammount time in seconds that the restaurant will operate.

When the restaurant opens (see *func OpenRestaurant* in restaurant/operation.go*)*, channels for each Entity are created, a worker pool of *Chefs* and *Waiters* is initialized, allowing for limited amount of *Employees* for to be available in the system. This behaviour is achieved by only sending a predefined maximum number of *Employees* to their respective buffered channel. These entities will remain the same throughout the runtime of the program, since there's no other function in the code that will

create new objects and place them in this channel. (See *restaurant/chef.go GenerateChefs*).
Once the *Employees* are in the system, a go routine for simulating a line of customer is created (see *restaurant/customer.go func Generate*Customers). This function is responsible for creating a customer and placing it in its respective channel (*customerChannel)*.
At this stage, everything is ready for the start of the operation, this would be equivalent to opening the restaurant to clients. *StartOperation* function uses a common *go* idiom, a *for* loop over a *select*, allowing to choose between different channels to handle different events, these events will be explained in detail in Section 4.1. Events. The flow-chart in Fig.1 describes the how events are handled across the different entities.

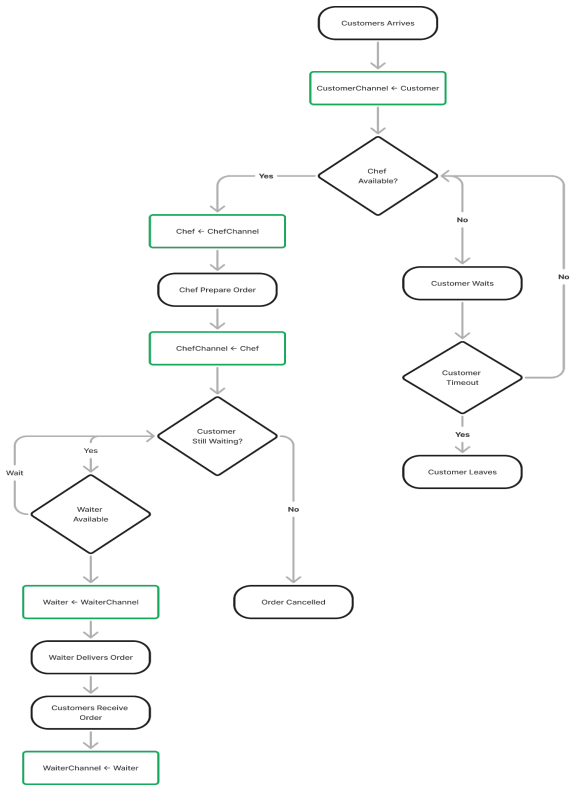

Fig 1. Flow Chart of the Application

4.1. Events

As mentioned in the previous section, there are several events that are triggered by using go Context [https://pkg.go.dev/context] or normal channels [https://go.dev/doc/effective_go#channels].
These contexts are passed down trough the called functions, and handled using the *for select* idiom.

4.1.1. *available{Entity}* or *new{Entity}*

This event, can be *availableWaiter* (see *func DeliveryOperation* in *restaurant/operation.go*), *availableChef* or *newCustomer* (see *func StartOperation* in *restaurant/operation.go*). These cases are selected if there is data to be read from the channels, usually a *Chef* or a that finished preparing an order.

4.1.2. restCtx

This context is created by the *func OpenRestaurant* (see *restaurant/operation.go)*. This is a context with a *timeout*, responsible for closing the restaurant, when selected (when the *timeout* expires), it will force a return on the current go routine.

4.1.3. customerCtx

This context is created with a *timeout* when a *Customer* arrives and *Chef* in *func StartOperation* (see *restaurant/operation.go)* is available to receive the order. When selected (*customer.Ctx.Done())*, whoever *Employee* is handling the order goes back to its respective channel and returns the current *go routine*, this way we simulate the cancellation of the order by the customer. This context can be cancelled, before a *Employee* receives an order or for the *Chef,* once the order is delivered to the waiter. (See *restaurant/waiter.go* in *func ServeOrder* and *restaurant/chef.go* in *func PrepareOrder)*.

5. Statistics and Logging

The system has its Event Streaming, where each event on the application is sent to the events channel.
After the restaurant closes, a function is called to read all events and generate statistics (as a restaurant balance day).
Examples:

- Customers Arrived
- Orders Accepted
- Orders Prepared
- Orders Delivered
- Customers Left
- Customers Served

6. Tests

The tests in this project guarantee that the program behaves as expected. They were implemented using the standard library for testing in go [https://go.dev/doc/tutorial/add-a-test]. In order

to have a controlled environment where was possible to test different scenarios, some mock functions were created to simulate and control some of the parameters to be inserted into the systems. These functions (See *restaurant/operation_test.go*), are modified versions of the *GenerateCustomers,* and *OpenRestaurant* in the *restaurante/operation.go* file, allowing for specific parameters to be used when creating customers such different *TimeToWait* or custom order preparation times (which in a non-test run, are a random number).
The tests are asserted with the metrics referenced in Section [5. Statistics and Logging] that were retrieved during the run of the program with the specific parameters. The test runs made verify that the system work in the following conditions:

- All customers wait enough time to get their orders:
    - All Customers Arrive
    - All Orders were Accepted
    - All Orders were Prepared
    - All Orders were Delivered
    - No Customers Left
    - All Customers were Served
- All customers have a zero wait time associated, meaning that no customer received their orders:
    - All Customers arrived
    - All Orders were Accepted
    - No Order was Prepared
    - No Order was Delivered
    - All Customers Left
    - No Customer was Served
- Just one customer with zero wait time associated:
    - All Customers arrived
    - All Orders were Accepted
    - All Orders were Prepared, except one
    - All Orders were Delivered except one
    - One Customer Left
    - All Customers were Served except one
- More customers than Workers
    - All Customers Arrive
    - All Orders were Accepted
    - All Orders were Prepared
    - All Orders were Delivered
    - No Customers Left
    - All Customers were Served