

Projet réseau PEI-2 2022-2023



Lundi 21 novembre 2022

Table des matières

Introduction.....	3
Description et déroulement	4
Récupération des commandes envoyées par le serveur UDP.....	5
Premier pas avec une base de données.....	6
Dialoguer avec une base de données.....	7
Concaténation des programmes	10
Capacités de production et choix du poste de production	13
Statistiques et IHM (<i>Interface Humain-Machine</i>)	18
Code optionnel	21
Conclusion	22

Introduction

L'objectif de ce projet est de faire un programme gérant les commandes que reçoit une pizzeria. Le projet est structuré de la manière suivante :

- Un serveur centralise les commandes des clients. Lorsqu'un client passe une commande, cette commande est envoyée à travers tout le réseau en UDP. Nous connaissons l'adresse IP et le port depuis lequel les commandes sont envoyées.
- Une base de données se trouve sur le réseau et nous connaissons l'adresse IP et le port de communication. Pour chaque commande reçue, nous devons lire les informations concernant la pizza commandée et savoir si la commande peut être prête avant l'heure où le client veut être livré.
- L'usine de production dispose de postes de production, nous devons donc distribuer les différentes commandes à travers les postes de commandes en prenant en compte les spécificités de ces postes.
- Enfin, nous devons être en mesure de présenter une interface web permettant de suivre l'avancement des différentes commandes.

A la fin de ce projet, nous disposerons d'un programme autonome permettant de gérer les commandes des clients ainsi que les postes de productions, tout en affichant des informations sur l'avancement de la production.

Dans la suite de ce compte-rendu, nous déroulerons nos réflexions et la création des différentes fonctions dans l'ordre chronologique. Il est important de noter que ce compte-rendu ne présente que les grandes lignes de la création du programme et que nous ne rentreront pas dans les détails techniques précises du programme.

Description et déroulement

Pour simuler des commandes réalisées par plusieurs clients, un serveur UDP envoie des trames comportant des informations concernant ces commandes. Nous connaissons l'adresse IP de ce serveur ainsi que le port depuis lequel les données seront envoyées. Une première étape consiste donc à récupérer les informations envoyées par ce serveur UDP. Nous serons en mesure de récupérer :

- la date et l'heure de la commande,
- l'identifiant du client,
- le nom de la pizza,
- la taille de la pizza,
- la quantité,
- l'heure souhaitée pour la livraison.

Nous disposons d'une base de données comportant des informations sur les pizzas, les clients et la chaîne de production. A partir des informations envoyées par le serveur UDP, nous devons être en mesure de savoir si la commande est faisable avant l'heure de livraisons souhaitée par le client. Pour cela, nous disposons, dans la table comportant les informations sur les pizzas, du temps de production de chaque pizza. Dans la table concernant les clients, nous disposons du temps qu'il faut pour leur livrer les pizzas (la distance en minutes). En utilisant ces informations, nous sommes en mesure de savoir si le temps de préparation et de livraison dépasse l'heure à laquelle le client veut être servi. Si c'est le cas, la commande est refusée.

Après cela, nous devons répartir les pizzas sur les postes disponibles et en mesure de préparer les pizzas. C'est-à-dire que nous devons respecter les restrictions de taille et de type de pizza que peut préparer le poste. Ensuite il nous faudra mettre à jour les postes. En effet les pizzas qui auront fini d'être préparées devront être retirées du poste sur lequel elles se trouvent afin de libérer la place qu'elles occupaient.

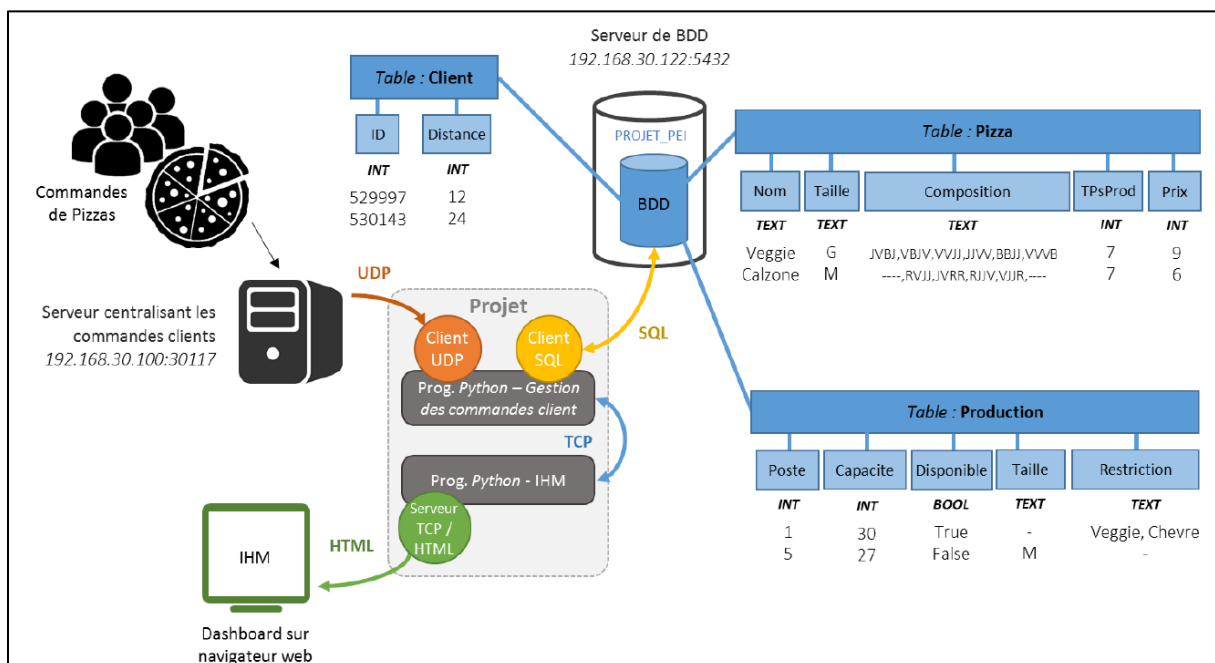


Figure 1: schéma représentant la situation générale.

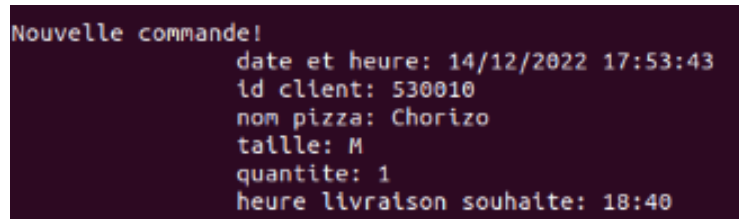
Récupération des commandes envoyées par le serveur UDP

Pour récupérer les informations envoyées par UDP, nous utilisons la bibliothèque *socket* nous permettant de gérer l'échange de paquets en Python. On crée un objet *sock* de type *socket*. Cet objet va traiter des messages devant être envoyés à une adresse IPv4, donc nous mettons dans les paramètres constituant notre objet *sock* le paramètre *socket.AF_INET*. Ensuite nous lui indiquons le paramètre *socket.SOCK_DGRAM* pour indiquer un échange de données par UDP¹. Nous assignons aux variables *HOST* et *PORT* respectivement l'adresse IP du serveur UDP et le port à partir duquel les données seront envoyées. A l'aide de la fonction *bind*, nous « écoutons » les données provenant de n'importe quelle adresse IP en lui indiquant en premier paramètre un espace. Nous lui indiquons en second paramètre le port d'où viennent les informations, soit 30117.

Maintenant que nous avons un port écoutant toutes les trames UDP circulant sur le réseau, nous souhaitons, par sécurité, ne lire que les trames provenant de l'adresse IP du serveur envoyant les commandes de pizzas (dont on connaît l'adresse IP). Cela se fait avec une simple boucle conditionnelle lisant le premier élément du tableau renvoyé par la bibliothèque *socket* lorsque nous recevons des données. Ce premier élément correspond à l'adresse IP de l'expéditeur des données, il nous suffit donc de la comparer à celle du serveur dont nous souhaitons recevoir les données.

A partir de maintenant, nous sommes en mesure de lire les données envoyées par le serveur. Ces données se trouvent sous forme d'une ligne dont chaque information est séparée par une virgule. Pour pouvoir récupérer les informations individuellement, nous utilisons la fonction *split* de Python. Cette commande nous permet de récupérer chaque élément en les séparant selon un caractère que nous avons spécifié en paramètre, dans notre cas la virgule. En utilisant cette fonction, nous sommes en mesure de créer un tableau dont chaque élément correspond aux informations.

Nous pouvons donc afficher toutes les informations reçues en utilisant leur indice dans le tableau.



```
Nouvelle commande!  
date et heure: 14/12/2022 17:53:43  
id client: 530010  
nom pizza: Chorizo  
taille: M  
quantite: 1  
heure livraison souhaite: 18:40
```

Figure 2: exemple d'interface retournant les détails d'une commande lors de sa réception.

¹ Pour plus d'informations, voir le compte-rendu du TP5 traitant des échanges de paquets en utilisant la bibliothèque Python.

Premier pas avec une base de données

Pour pouvoir dialoguer avec une base de données en Python, nous utiliserons la bibliothèque *psycopg2*. Une première étape consiste à créer une session de connexion avec la base de données. Pour cela, nous utilisons la fonction *connect* qui nous renvoie une instance de connexion. On indique en paramètre de cette fonction le nom de la base de données, le nom d'utilisateur, le mot de passe, l'adresse IP de la machine où se trouve la base de données ainsi que le port de ladite machine où il faut envoyer les requêtes.

Ensuite on crée une variable de type *cursor* (ici « cur ») permettant de parcourir la base de données d'où pointe la session de connexion.

En appliquant la fonction *execute* à la variable de type *cursor*, nous sommes capables d'envoyer des requêtes SQL à la base de données. Les requêtes sont à passer en paramètre de cette fonction.

La fonction *execute* prend seulement une chaîne de caractère en paramètre dans lequel se trouve toute la requête SQL. Ce type de fonctionnement nous obligera à faire attention à quelques petites subtilités. Comme lorsque l'on veut envoyer un guillemet, il faut toujours mettre un *backslash* devant afin que le guillemet ne soit pas considéré comme une fin de chaîne de caractère mais bien comme étant une partie intégrante de la requête SQL.

Pour pouvoir récupérer les résultats retournés par la requête, nous utilisons la fonction *fetchall* sur la variable de type *cursor*. Les différents t-uplets retournés par la requête sont représentés par des tuples. Ces tuples se trouvent eux-mêmes dans un tuple, qui est stocké dans la variable de type *cursor* (ici « cur »).

A l'aide d'une boucle *for* ainsi que la fonction *fetchall*, nous sommes en mesure de parcourir tous les tuples retournés par la requête et d'afficher chaque élément se trouvant dedans. La fonction *fetchall* nous permet de parcourir tous les uplets retournés par l'exécution de la requête, mais dans le cas où nous savons que seul un seul uplet est retourné par la requête, nous pouvons utiliser la fonction *fetchone*.

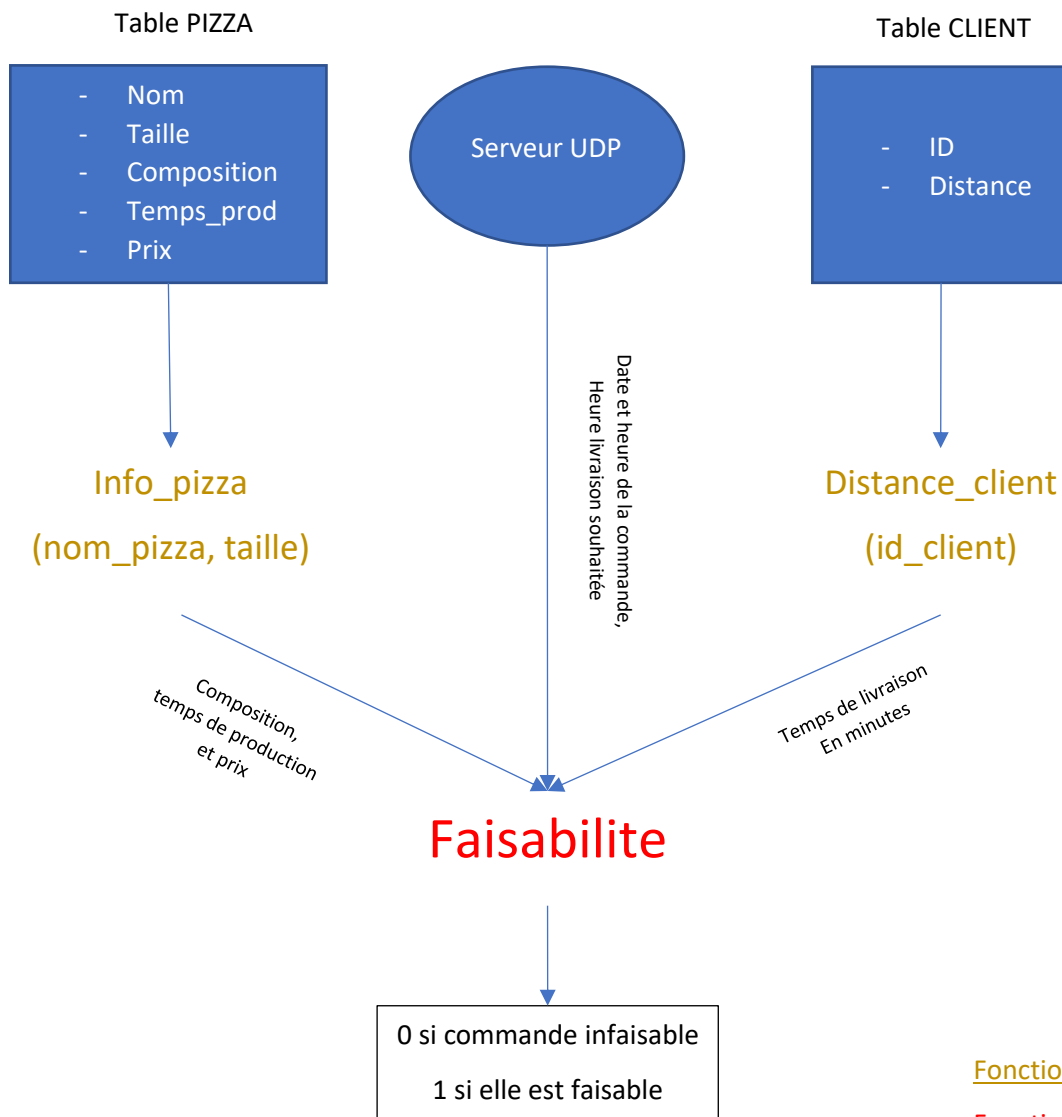
Dialoguer avec une base de données

Dans l'objectif de réaliser une fonction permettant de vérifier la faisabilité d'une commande, nous devons comparer la date à laquelle une commande a été passée avec la date à laquelle la livraison est souhaitée. Avec cela, nous devons savoir si la pizza pourra être préparée avant la date de livraison souhaitée. Tout cela nous permettra de créer une fonction « faisabilité » nous permettant de savoir si une commande est faisable ou non.

Dans un premier temps, nous devons récupérer la distance (en minutes) à laquelle se trouve un client en fonction de son identifiant. Cela nous permettra de connaître le temps de livraison d'une commande particulière. Ces informations se trouvent dans une base de données dont on connaît l'adresse IP sur le réseau. Dans cette base de données se trouve une table contenant la distance du client en fonction de son identifiant. Il nous faut donc récupérer cette distance pour un identifiant donné. Nous connaissons aussi le nom de la pizza commandée par le client ; il nous faudra donc récupérer le temps de préparation de la pizza en question dans la table « Pizza ».

Pour pouvoir créer une fonction « faisabilité » permettant de savoir si une commande est réalisable, il faut donc que cette fonction utilise deux autres fonctions :

- `Info_pizza` : prend en paramètre le nom de la pizza à fabriquer ainsi que sa taille. Cette fonction récupère, en utilisant des requêtes SQL, toutes les informations concernant cette pizza en sortie, la fonction renvoie un tuple contenant toutes ces informations. Nous sommes donc en mesure de récupérer le temps de production de la pizza en connaissant sa position dans ce tuple.
- `Distance_client` : prend en paramètre l'identifiant du client. De la même manière que la fonction « `Info_pizza` » cette fonction récupère la distance du client (identifié par son ID) et renvoie cette valeur en format *int*.



Fonctions secondaires

Fonctions principales

Figure 3: schéma représentant la situation générale permettant de savoir si une commande est réalisable ou non.

Maintenant que la fonction « faisabilité » connaît la date et l’heure à laquelle la commande a été passée, le temps de livraison chez le client et l’heure à laquelle le client souhaite être livré, il faut qu’elle soit capable d’indiquer si la commande peut être livrée à temps chez le client. Nous prenons pour acquis le fait qu’un client ne peut pas commander de pizza tard le soir et vouloir qu’elle soit livrée le lendemain. Ainsi, la fonction « faisabilité » doit pouvoir connaître le temps disponible ainsi que le temps nécessaire à la fabrication des pizzas. Le temps disponible correspond au temps, en minutes qu’il y a entre l’heure de la commande et le temps de livraison souhaité. Le temps nécessaire correspond au temps de fabrication du type de la pizza commandé, multiplié par le nombre de pizzas commandé, auquel on ajoute le temps de livraison.

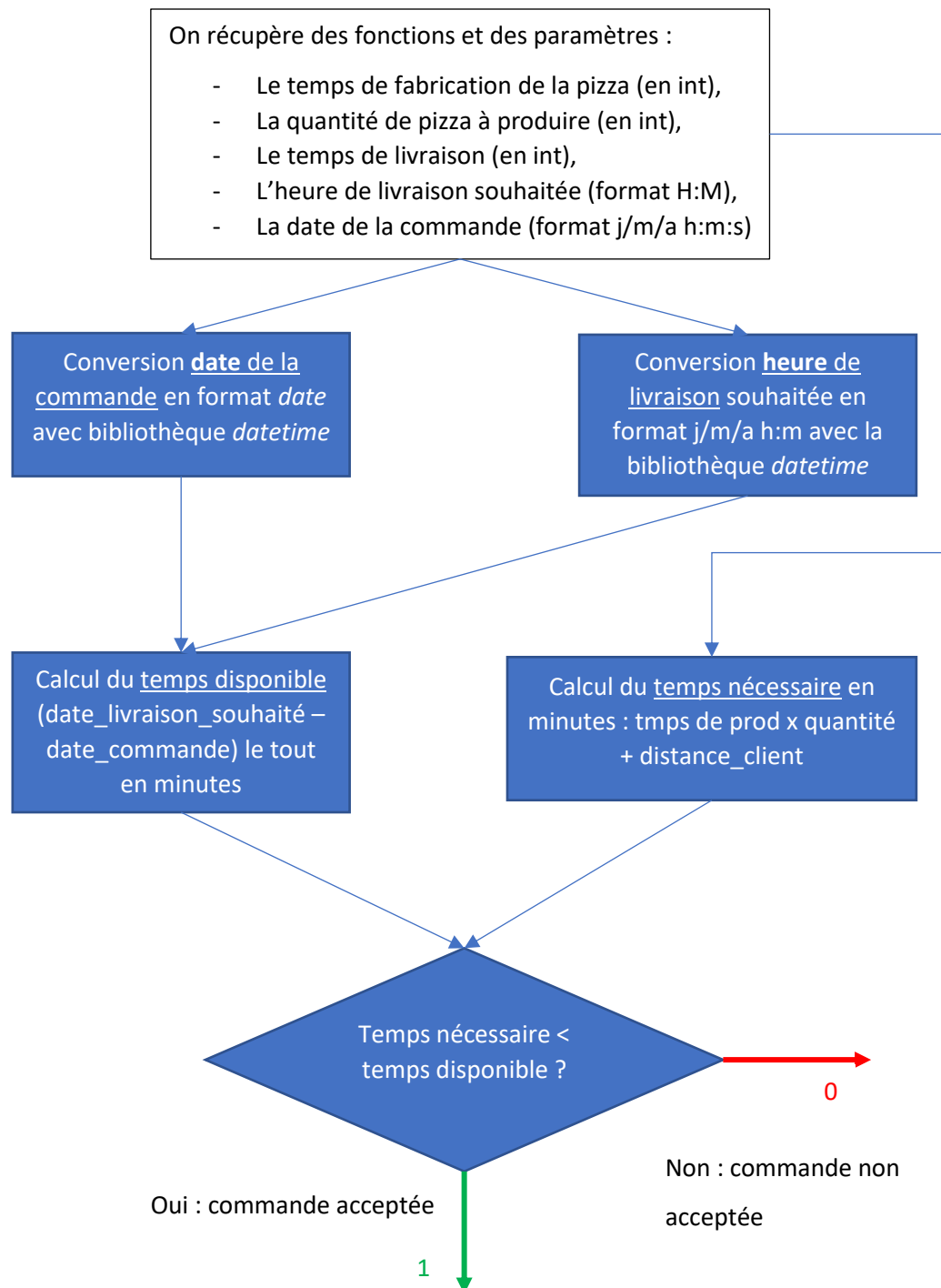


Figure 4: schéma représentant le fonctionnement de la fonction "faisabilité" permettant de savoir si une commande est faisable ou non.

Concaténation des programmes

Nous disposons d'un programme permettant de récupérer les commandes renvoyées par le serveur UDP et d'un programme permettant de savoir si une commande est réalisable. La prochaine étape consiste donc à concaténer ces deux programmes afin de savoir, pour chaque commande reçue, si la commande est réalisable dans les temps. Pour cela, il nous faut simplement exécuter la fonction de faisabilité chaque fois qu'une nouvelle commande est reçue en UDP. Il nous faut donc récupérer les différentes informations reçues (nom de la pizza, heure de la commande, quantité de pizzas à produire, etc...) et utiliser ces informations dans la fonction « faisabilité » pour savoir si la commande est acceptée ou non.

La manière dont nous récupérons les différentes informations est la même qu'expliquée dans la partie « Récupération des commandes envoyées par le serveur UDP ». Nous sommes donc en mesure d'assigner les différentes informations dans des variables en sachant à quel indice du tableau renvoyé par la fonction *split* se trouve une information en particulier.

Ayant toutes ces informations à disposition, la fonction « faisabilité », comme indiqué dans la partie « Dialoguer avec une base de données », est capable d'indiquer si la commande sera prête avant la l'heure où le client veut être livré.

Nous pouvons faire une « interface » primitive sur le terminal permettant d'afficher les différentes commandes passées avec leurs informations ainsi que pour savoir si la commande a été acceptée ou non. Pour cela, il nous suffit d'utiliser la fonction *print* pour afficher les différents éléments du tableau retournée après séparation des informations contenues dans les trames UDP. Enfin, on affiche une phrase en conséquence si la commande est acceptée par la fonction « faisabilité ». Un exemple d'interface lors de la réception d'une commande peut être retrouvé en Figure 6.

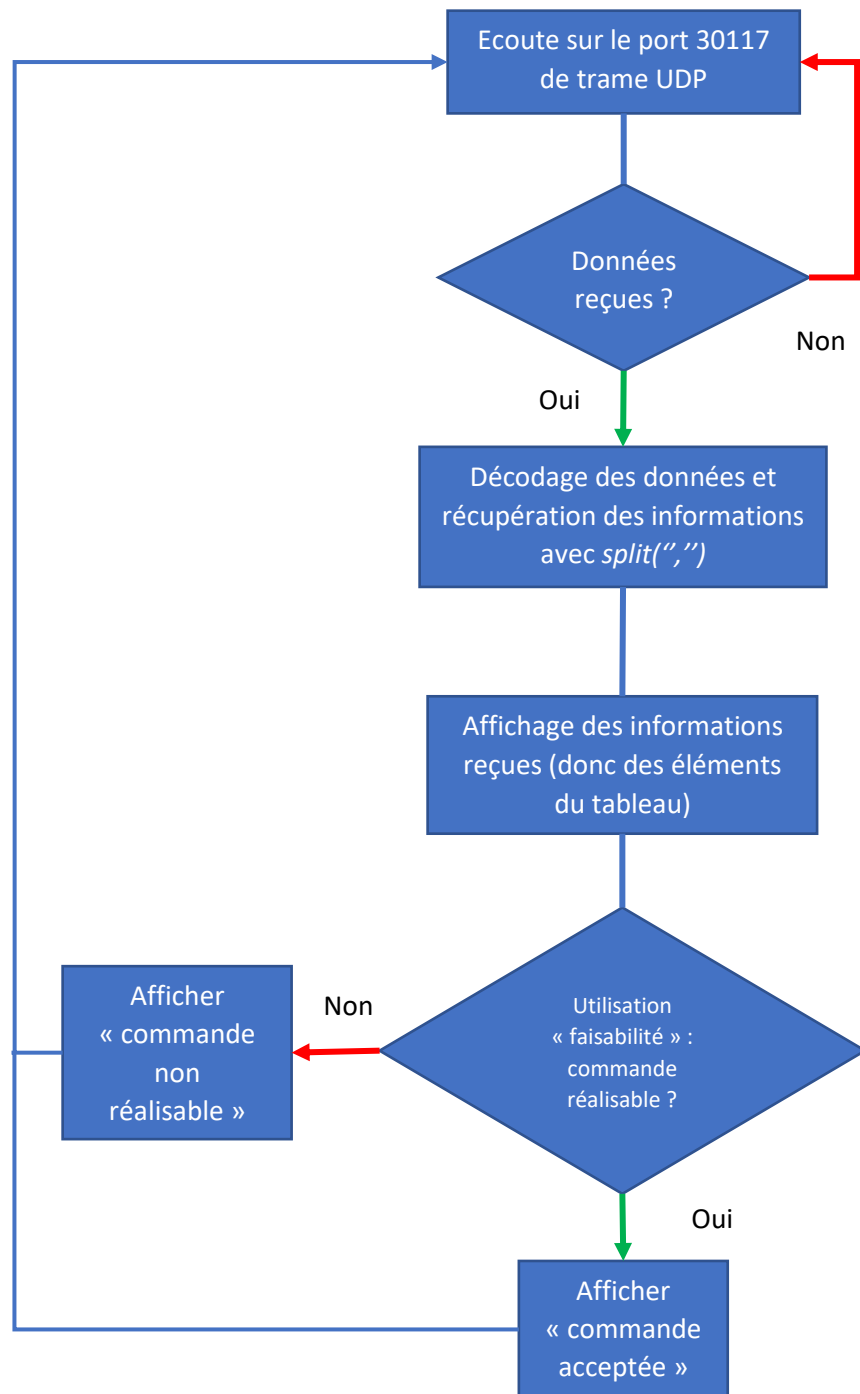


Figure 5: schéma représentant le fonctionnement du programme récupérant les commandes et indiquant si elles sont faisables.

```
Nouvelle commande!  
date et heure: 14/12/2022 17:53:43  
id client: 530010  
nom pizza: Chorizo  
taille: M  
quantite: 1  
heure livraison souhaite: 18:40  
  
Commande acceptee.
```

Figure 6: exemple d'interface retournée lors de la réception d'une commande.

```
Nouvelle commande!  
date et heure: 14/12/2022 17:53:35  
id client: 530064  
nom pizza: Veggie  
taille: G  
quantite: 4  
heure livraison souhaite: 18:23  
  
Commande refusee, le temps disponible pour la preparation et la livraison est insuffisant.
```

Figure 7: exemple d'interface retournée lors de la réception d'une commande refusée.

Capacités de production et choix du poste de production

Désormais nous considérons que l'usine qui produit ces pizzas possède six postes de production. Ces postes de productions possèdent une capacité de production correspondant au nombre de pizzas qu'ils peuvent fabriquer en même temps. Ils possèdent aussi des restrictions comme la taille des pizzas qu'ils peuvent produire (« M » ou « G » ou "-" pour les deux) ainsi que le nom des pizzas ne pouvant pas être produite sur ce poste ("-" si pas de restrictions particulière). Ces postes possèdent aussi un champ renseignant si le poste est opérationnel (*True*) ou non (*False*). Notre but consiste donc à répartir les commandes de pizzas parmi ces postes de production en prenant en compte les différentes restrictions.

Les informations concernant les postes de productions se trouvent dans la même base de données où se trouvent les informations concernant les pizzas et les clients. La table qui nous intéresse cette fois-ci est la table *Production*.

Cette table se présente sous la forme suivante (voir Tableau 1) :

Champs :	Poste	Capacité	Disponibilité	Taille	Restriction
Type :	INT	TEXT	BOOL	TEXT	TEXT
Exemple de valeur :	5	15	TRUE	G	Chevre, 4_Fromages
Exemple de valeur :	6	18	FALSE	-	-

Tableau 1: présentation de la table "Production".

Pour chaque commande reçue, il nous faudra donc parcourir tous les postes de production et vérifier que celui-ci est disponible, que sa capacité maximale n'est pas atteinte, que la taille de la pizza correspond à la taille de restriction du poste (ou qu'il n'y a pas de restriction) et que le nom de la pizza ne fait pas partie des pizzas « interdites » par le poste.

Avant tout cela, il nous faut tout d'abord créer la structure de nos postes dans notre programme. Pour cela, nous avons décidé de créer une liste comportant chaque poste, cette liste est stockée dans la variable *Postes*. Chaque poste est lui-même une liste comportant les pizzas en cours de préparation. Enfin, chaque pizza est représentée par un tuple comportant le nom de la pizza, sa taille ainsi que l'heure de fin de préparation. Nous avons donc un tableau à trois dimensions si l'on compte les tuples représentant les pizzas comme une dimension. Si l'on veut obtenir une information sur une commande de pizza, comme sa taille, nous devons entrer :

`Postes[indice du poste][indice de commande][indice où se trouve la taille]`

Pour l'instant, aucune pizza ne se trouve dans les postes de production. Donc au début de l'exécution de notre programme, nous disposons d'un tableau à deux dimensions entièrement vides.

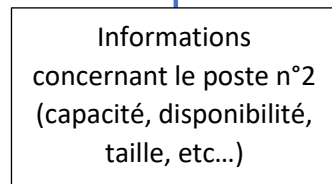
Pour éviter de faire des requêtes SQL récupérant les informations concernant les postes de nombreuses fois, nous décidons de créer une fonction « Info_Poste » prenant en paramètre le numéro du poste dont il faut récupérer les informations. Cette fonction retourne un tableau contenant les valeurs des champs *Capacités*, *Disponibilite*, *Taille* et *Restriction* du poste dont on a précisé le numéro. Pour cela, nous devons exécuter la requête SQL suivante :

```
« SELECT "Capacite", "Disponibilite", "Taille", "Restriction" FROM "Production" WHERE "Poste" = » + str(numero_poste)
```

Comme le numéro de poste est considéré comme une clé primaire de la table « Production », nous pouvons afficher le seul uplet que la base de données nous renvoie avec la fonction *fetchone*.

Ainsi, nous assignons à une variable *infoPostes* une liste dont les éléments sont les listes contenant toutes les informations relatives aux postes de productions.

```
infoPostes = [Info_Poste(1), Info_Poste(2), Info_Poste(3), Info_Poste(4), Info_Poste(5),  
Info_Poste(6)]
```



Cette liste *infoPostes* n'est créée et actualisée qu'au début de l'exécution du code (donc une seule fois), or la disponibilité des postes est susceptible de changer au cours de l'exécution du code, pendant que celui-ci récupère les commandes et les assigne aux différents postes. Nous avons donc créé une autre fonction *Dispo_Poste* prenant en paramètre un numéro de poste et retournant un booléen correspondant à la disponibilité du poste² en question. Pour résumer :

- *Postes* est une liste comportant les six postes qui sont eux-mêmes représenté par une liste comportant les informations sur les pizzas,
- *infoPostes* est une liste comportant six éléments, pour les six postes. Ces éléments sont des listes contenant les informations relatives aux postes 1 à 6,
- *Dispo_Poste* est une fonction permettant de renvoyer un booléen correspondant à la disponibilité du poste dont le numéro est entré en paramètre de la fonction.

Ayant tout cela, nous sommes déjà en mesure de distribuer les commandes de pizzas aux différents postes de production. En effet, lorsque nous recevons une commande faisable, il nous suffit de parcourir les postes jusqu'à en trouver une pouvant fabriquer la pizza de la commande. Il faudra donc ajouter à la liste représentant le poste un tuple contenant le nom de la pizza, sa taille ainsi que l'heure de fin de préparation. Nous ne pouvons ajouter la commande au poste seulement si les conditions énoncées [précédemment](#) (page 13) sont respectées.

Après réception d'une commande et avant de l'assigner à un poste de production, nous utilisons la fonction *Dispo_Poste* pour mettre à jour la disponibilité de chaque poste en changeant la valeur déjà enregistrée dans le tableau *infoPostes* par celle que retourne la fonction et cela pour chaque poste.

² Pour rappel, la disponibilité d'un poste est représentée par un booléen (voir Tableau 1).

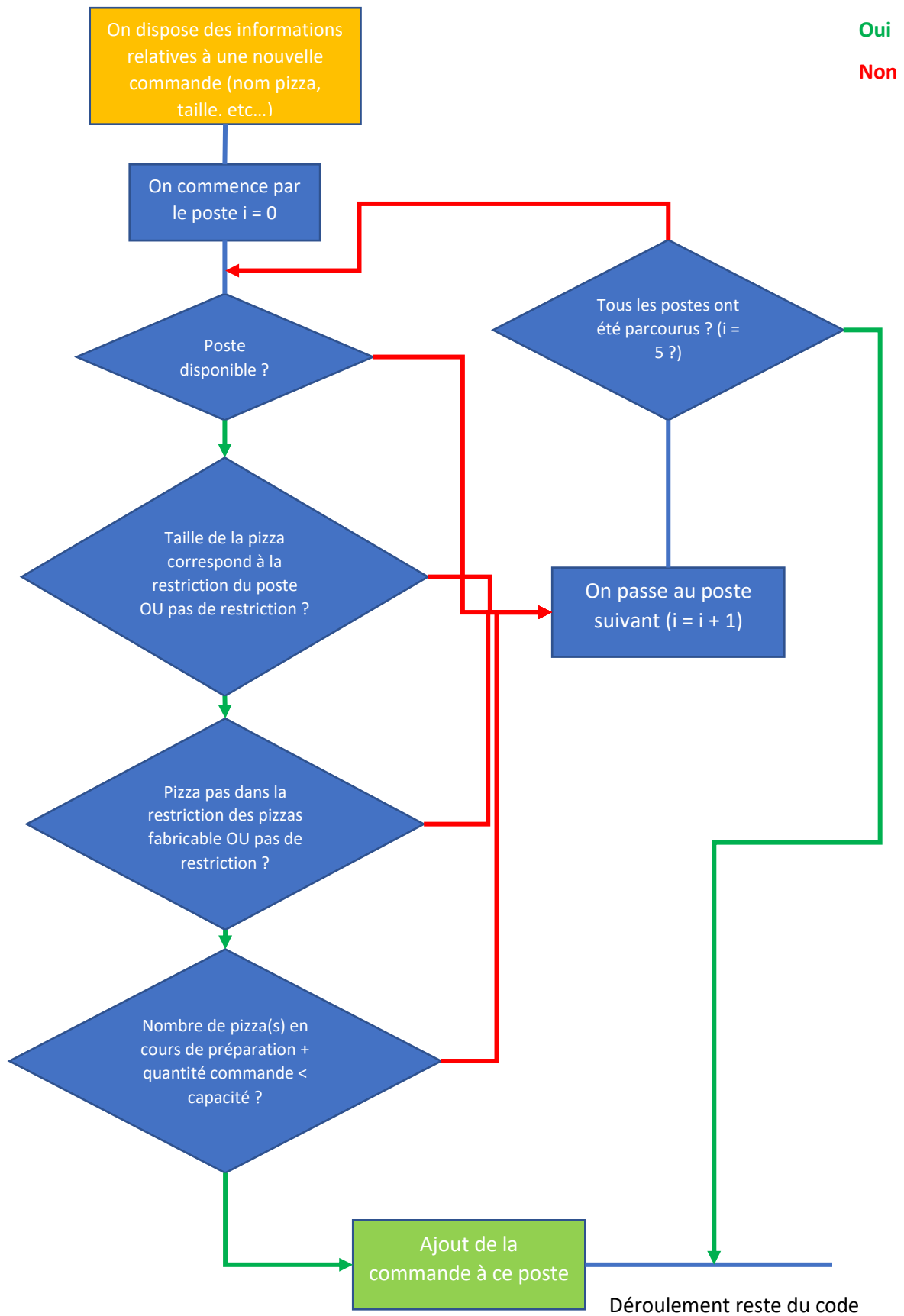


Figure 8: schéma représentant le fonctionnement de l'algorithme permettant d'assigner une commande à un poste.

Nous utilisons cette méthode de distribution des commandes à travers les postes pour chaque commande reçue, mais il ne faut pas oublier de prendre aussi en compte la quantité de pizza dans cette commande. C'est pourquoi nous réitérons l'opération aussi en fonction de la quantité, pour chaque commande.

Maintenant que nous savons assigner des commandes à différents postes de production, nous devons désormais être capable de supprimer les pizzas des postes quand celles-ci sont considérées comme prêtes. Lors de l'ajout d'une commande à un poste de production, nous avons mis dans le tuple correspondant à la commande l'heure de fin de production de celle-ci dans le format heure:minute. Il nous faut donc connaître l'heure actuelle, parcourir les postes de production et supprimer les pizzas dont l'heure de fin de production correspond à l'heure actuelle.

Pour cela, il existe des fonctions en Python permettant de récupérer l'heure du système de l'ordinateur. Cependant, l'heure de notre machine virtuelle n'étant pas correcte et n'osant pas modifier les paramètres du système en conséquence, nous avons dû trouver une alternative.

Lors de la réception d'une commande, nous recevons comme information la date et l'heure (précision à la seconde) à laquelle la commande a été passée. Nous considérerons donc cette heure comme celle actuelle au moment où nous recevons la commande.

Nous avons créé une fonction permettant de ne récupérer seulement l'heure et la minute où la commande a été passée. Ensuite nous parcourons tous les postes de production et nous comparons, pour chaque commande en cours de préparation l'heure considérée comme actuelle avec celle de fin de préparation. Si les deux informations coïncident, nous supprimons la commande en question. Le parcours des postes de production se fait lorsque nous devons assigner une commande à un poste de production.

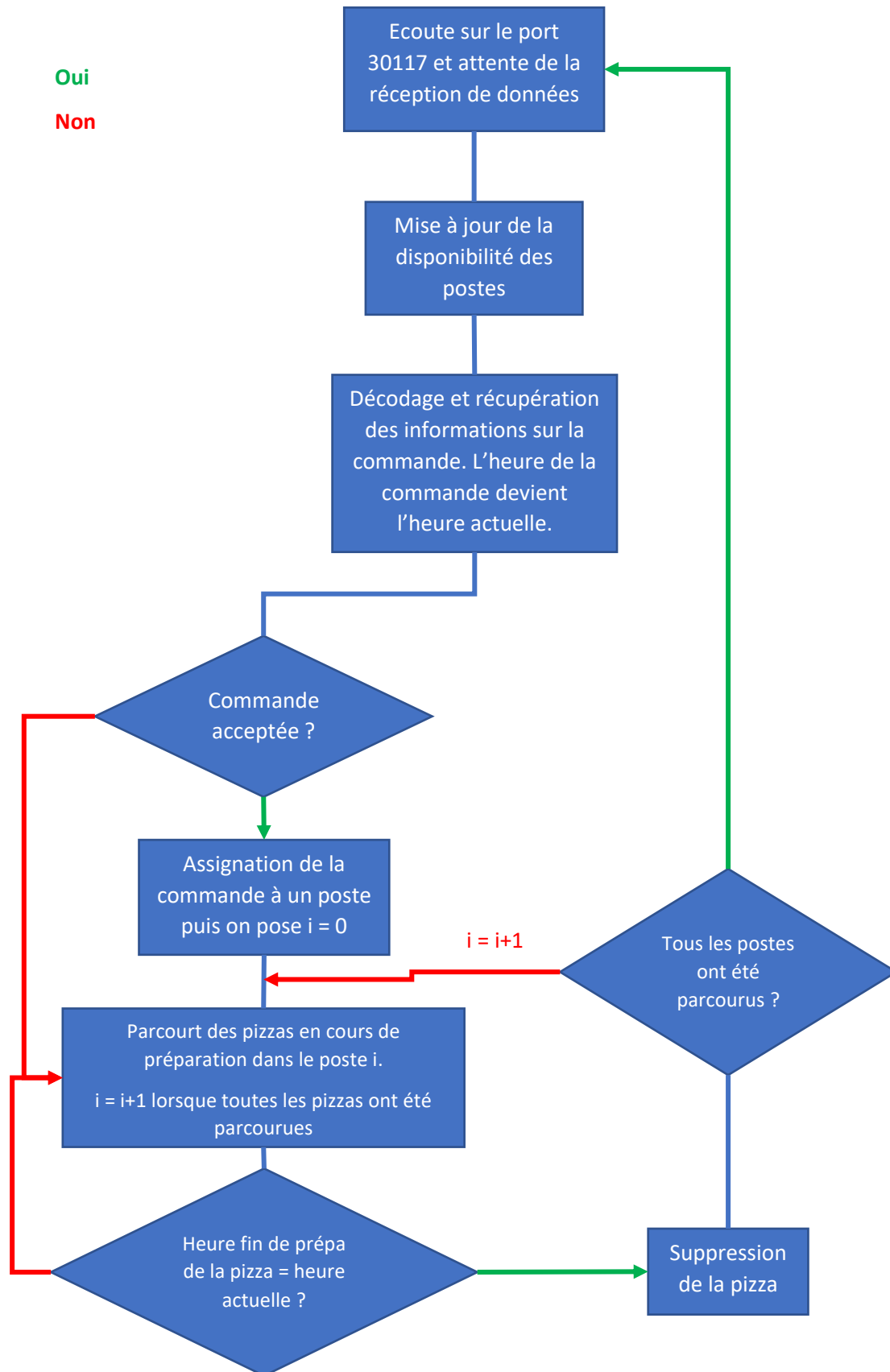


Figure 9: schéma représentant le fonctionnement de la gestion des commandes de pizzas.

Statistiques et IHM (*Interface Humain-Machine*)

Nous voulons désormais faire apparaître des statistiques concernant les différentes commandes dans une interface web simple. Pour cela, nous allons développer un autre code Python fonctionnant en parallèle du code principale gérant les commandes de pizzas. Il nous faut donc faire en sorte que les deux programmes puissent communiquer entre eux. En effet, le programme permettant de gérer l'interface web doit pouvoir recevoir les données que l'on veut afficher depuis le programme de gestion des commandes, dit « principal ». Nous utiliserons le protocole TCP pour envoyer ces données. Le programme de gestion de l'interface web devra donc utiliser un port pour recevoir ces données (disons le port 12345) et un autre port pour la connexion au navigateur web (disons le port 10017).

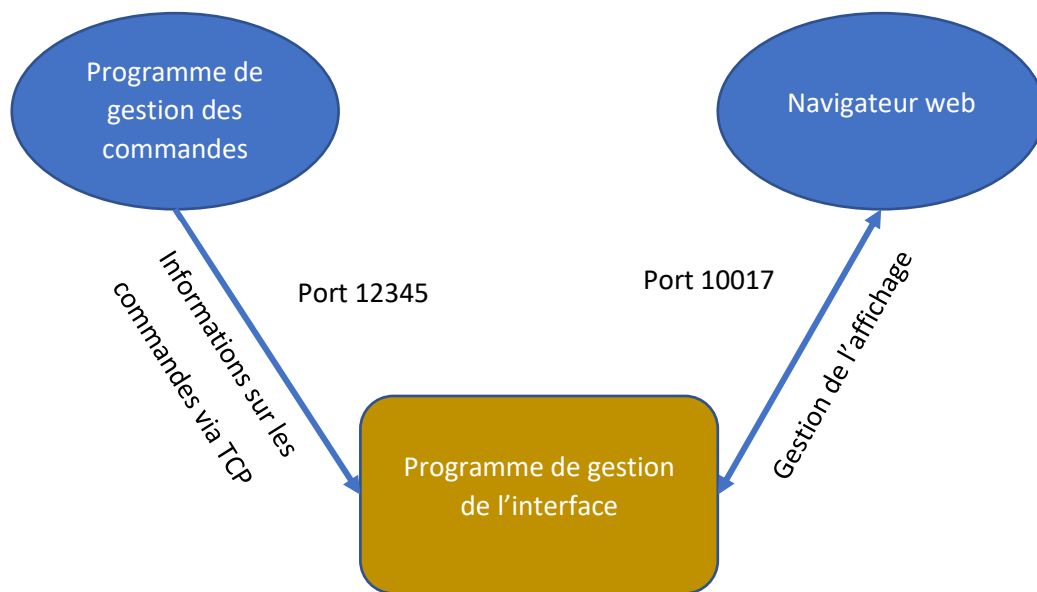


Figure 10: schéma représentant les communications du programme de gestion de l'affichage web.

Le transfert des données depuis le programme « principal » jusqu'au programme de gestion de l'interface web est simple. Nous utilisons la bibliothèque *socket* de manière habituelle pour se connecter au port 12345 depuis le programme de gestion des commandes. Pour pouvoir envoyer toutes les données qui nous intéressent en un seul envoi, nous envoyons un *string* comportant la valeur des variables (contenant les données qui nous intéressent) et ces valeurs sont séparées par un « - ». Du côté du programme de gestion de l'interface web, nous sommes en mesure de récupérer ce *string* et de séparer les différentes valeurs à l'aide de la fonction *split* prenant en paramètre le séparateur « - ».

Nous avons remarqué une subtilité quant au transfert d'informations entre le programme de gestion des commandes et le programme de gestion de l'interface web. En effet, lorsque que le programme de gestion de l'interface web est en « pause » le temps qu'une nouvelle connexion de la part du navigateur web se fasse (5 secondes), il se peut que de nouvelles informations concernant les dernières commandes soient envoyées pendant cette pause. Nous avons remarqué que l'expéditeur accumule les informations à envoyer le temps que le receveur établisse une nouvelle connexion et reçoive ces informations. Cette accumulation étant inévitable, car en 5 secondes plusieurs commandes arrivent généralement, pour être sûr de récupérer les dernières informations on parcourt le tableau renvoyé

après utilisation de la fonction *split* depuis la fin. Par exemple, pour pouvoir récupérer le dernier élément du tableau, nous utiliserons la syntaxe *tableau[len(tableau)-1]*³.

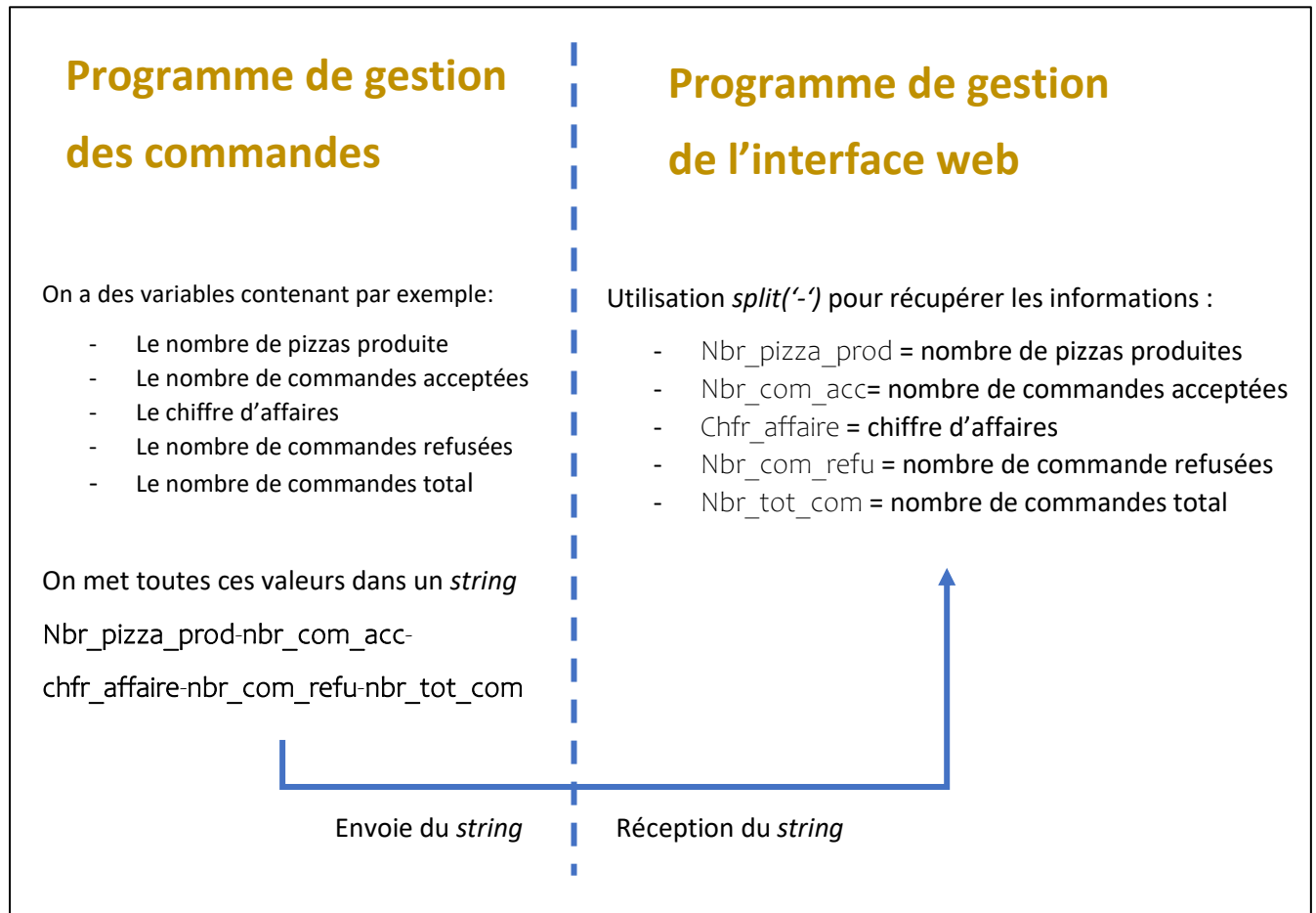


Figure 11: schéma représentant le fonctionnement de la communication entre le programme de gestion des commandes et celui de gestion de l'interface web.

Maintenant que le programme de gestion de l'interface web possède les informations à afficher sur le navigateur, il nous faut maintenant les afficher. Une manière simple de le faire est d'envoyer le code HTML directement via la bibliothèque *socket* au navigateur web. Ce navigateur web correspond à une connexion au programme Python au port 10017 (voir Figure 10).

Il nous suffit d'écrire le code HTML dans un *string* assigné à une variable. Dans le corps, nous indiquons entre les balises « `<p></p>` »⁴ de simples phrases ainsi que les valeurs des variables contenant les informations que l'on veut afficher.

Dans l'en-tête du code HTML (entre les balises `<head></head>`), nous voulons indiquer au navigateur web que les caractères sont encodés en *utf-8* pour éviter les problèmes d'affichage des caractères comportant des accents. Nous ajoutons donc la ligne `<meta charset='utf-8'>`. Nous voulons ensuite que les informations affichées sur la page web soient actualisées. Au lieu de créer une nouvelle instance de connexion chaque fois que des informations doivent être actualisées, nous indiquerons à la page web d'actualiser la connexion toutes les 5 secondes. Pour cela il nous faut ajouter la ligne `<meta http-equiv='refresh' content='5'>`.

³ Nous aurions aussi pu utiliser la syntaxe *tableau[-1]*.

⁴ Ces balises permettent de préciser qu'il s'agit d'un simple paragraphe. Les balises h1, h2, h3, etc, h7 permettent d'indiquer qu'il s'agit d'un titre, qu'il faut afficher en grand, par ordre décroissant de grosseur.

Pour que le code HTML soit envoyé au navigateur client, il nous suffit d'envoyer le *string* contenant ce code à l'instance de connexion correspondant. Après envoi du code HTML, nous pouvons fermer la connexion et en attendre une nouvelle de la part du navigateur. Comme le navigateur rafraichira la connexion toutes les 5 secondes, cette nouvelle connexion se fera de manière automatique. Cette manipulation nous permet d'envoyer du nouveau contenu pour le code HTML, permettant ainsi d'actualiser les informations à afficher sur l'interface web.

Code optionnel

Comme indiqué précédemment, nous avons décidé d'afficher aussi le nombre de commandes total, le nombre de pizzas produites ainsi que le chiffre d'affaires. La manière de récupérer ces informations est assez simple. Pour afficher le nombre de commandes total, il nous suffit d'additionner le nombre de commandes refusées avec le nombre de commandes acceptées. Le nombre de pizzas produites correspond au nombre de pizzas que nous avons supprimées des postes de production lorsque celles-ci sont prêtes. Il nous suffit donc d'incrémenter une variable chaque fois qu'une pizza est supprimée d'un poste de production. Enfin, le chiffre d'affaires correspond au prix de la pizza commandée multipliée par la quantité. Cette variable s'incrémente à chaque commande acceptée.

Conclusion

Ce projet nous a permis de mettre en œuvre un système complet de gestion des commandes de pizzas, en terminant par la création d'une interface web simple et primitive permettant de suivre l'évolution des commandes. Il nous a permis d'avoir une idée sur à quoi ressemble le développement d'un système de gestion quelconque au sein d'une entreprise et nous permettra de mettre en pratique les compétences acquises lors d'un véritable projet d'entreprise.

Ce projet nous à permis d'utiliser des techniques variées telles que la programmation en Python, la communication client/serveur en TCP et UDP avec la bibliothèque *socket* en Python, l'utilisation de requêtes SQL pour récupérer des informations dans une base de données ainsi que le développement d'un code HTML. Nous pouvons imaginer utiliser ces techniques et compétences dans de nombreux autres domaines d'application, comme la gestion d'une usine de fabrication d'objets, le but de ce projet étant principalement d'acquérir des compétences pour pouvoir les utiliser dans d'autres domaines.

Une des compétences les plus importantes que ce projet nous à permis de développer est le travail d'équipe ainsi que l'organisation et la répartition du travail au sein du groupe. En effet, nous avons trouvé difficile de créer une « feuille de route » nous permettant de répartir le travail au sein des membres du groupe. Nous avons donc commencé ce projet par dessiner sur une feuille un découpage du projet nous permettant de savoir ce que nous devons faire précisément. Ensuite, nous avons réparti le travail au fur et à mesure des séances, l'un avançant ce compte-rendu et l'autre avançant le programme.

Le but de ce compte-rendu est de ne parcourir que les grandes lignes du fonctionnement de nos codes. Au fur et à mesure des séances, nous avons créé plusieurs « sous-fonctions » permettant d'alléger les fonctions décrites dans ce compte-rendu.