1st Assignment – P2P Drive-Through using token ring

Subject: Computação Distríbuida

Professors:

- ***Diogo*** *Gomes*
- ***Mário*** *Antunes*

Students:

- ***Vasco*** *Ramos, nmec 88931*
- ***Diogo*** *Silva, nmec 89348*

Discovery Process

How it works:

1. Check whether we just started the discovery process
    1.1 if True: Return the message *{'method': 'NODE_DISCOVERY', 'args': "idDictionary": {self.name: self.id}, "rounds": 0}}*
2. Check if the name of the current entity is NOT in the *idDictionary* received
    2.1 if True: Add the current entity's name as a key and the id as a value to the *idDictionary*
3. If 2. is false, check if the current name-Id pair is NOT in the *idDictionary*
    3.1 if True: Add the current Id to the Name's Id's list
4. If 2. and 3. are false check if we're at the Token Generator entity
    4.1 if True: increment *rounds* by 1
    4.2 else: return none
    4.3 Register the nodes in the *idDictionary* to the local *ringIDs* variable
5. Pack the updated idDictionary and rounds to a message and return it

Notas:

-É usada uma variável *discoveryStarted*. O processo de descoberta e iniciado quando este for false e o ring estiver completo (*discoveryStarted* é mudado para true para não começarmos o processo mais do que uma vez)
-O algoritmo obriga a que mensagens de *NODE_DISCOVERY* passem 2 e só 2 vezes por cada entidade – uma para adicionar o seu id ao dicionário que vai ser passado, outra para registar no seu local – (A entidade que iniciar o processo sera a única que recebera *NODE_DISCOVERY* 3 vezes simbolizando que todas as entidades já tem o dicionário registado, sabemos que o processo deve terminar para uma dada entidade quando chegar uma mensagem com *rounds == 1*. O processo termina de vez quando o resultado do ***nodeDiscover()*** for uma mensagem vazia.

Chef – Order processing

How it works:

1. Check whether the Cook has any messages in it's *in_queue*

    1.1 if True: Interpret the message and go back to 1 after that's done

2. Check if currentOrder is None

    2.1 if True: Check if we have orders in the order queue

        2.1.1 Continue

        2.1.1 Get the top order from the order queue and set it as the current order

3. Check if the current order is ready to be delivered

    3.1 if True: Put DELIVER method TOKEN in the out_queue and set the current order as None

4. Check if we have nothing in the requestList

    4.1 if True: Add the equipments needed to complete the order to the requestList

Notes:

-To check whether an order is complete when we receive an order we add 3 variables: grill, fryer and drinks, and set each of them to 1. Then subtract them by the amount of times we need them (e.g if the order has 3 hamburgers, we set grill to 1-3 = -2). We know the order is complete when all 3 of these variables == 1

-We use the requestList to know whether or not we need an equipment (when we receive it in the case of Equipments as Tokens, or to request them in the case of Equipments in the Restaurant)

Equipments – as Tokens

How it works:

1. Initialize 3 token messages – ***GRILL_TOKEN ; FRIER_TOKEN ; DRINKS_TOKEN*** – (e.g *{'method': 'GRILL_TOKEN', 'args': {'available': 1}}*) and put them circulating
2. When receiving an equipment token check if available == 0
   2.1 if True: Keep it circulating
   2.2 else: Grab the equipment Token, change available to 0 and keep the now empty equipment token circulating

In Cook:

1. When receiving an equipment token, first check if the corresponding equipment is in the local *requestList* (e.g if we received a *GRILL_TOKEN* check if "grill" is in the requestList)
   1.1 if True: Use the equipment by "working" for a random amount of time – specified in the *conf.ini* file
2. Put the equipment token back in the *out_queue*

In Restaurant:

1. When receiving an equipment token, put it in the out_queue

Notes:

-We implemented a Blacklist – When receiving a new message the entity checks if the method received is in the blacklist, if it is, it ignores it and forwards it instantly
-All the Restaurant does is check if an equipment token passes by it, as a way to assure that the equipments tokens are still alive

Equipments – handled by the restaurant

How it works:

In Cook:

1. When handling a new order, the cook checks which equipments he'll need, sending a message with the method EQUIPMENT_REQUEST (e.g for a grill request:

{'method': 'TOKEN', 'args': {'method': 'GRILL_REQUEST', 'args': { 'id': self.discovery_table['RESTAURANT'], 'cookID': comm.getSelfID(), 'equipment': 'grill'}}})

2. When receiving an EQUIP_LEND message, check the type of equipment received (by looking at the equipment in the args).

    2.1 Work depending on the type of equipment received

    2.2 Increment the currentOrder[equipment] by 1 and remove an instance of the equipment from the requestList

    2.3 Put a return message in the out_queue {'method': 'TOKEN', 'args': {'method': 'EQUIP_RETURN', 'args': {

'id': self.discovery_table['RESTAURANT'], 'equipment': equipmentReceived}}}

In Restaurant:

1. When receiving a Request message (GRILL_REQUEST ; FRIER_REQUEST ; DRINKS_REQUEST) check if its available

    1.1 If the equipment is available (it's control variable is set to 1) send an EQUIP_LEND message

    1.2 Else: put the ID of the cook that requested the equipment in the corresponding request queue

2. When receiving a Return message (EQUIP_RETURN"), check the type of equipment returned

    2.1 Check if we have any cook waiting in the equipment's request queue

        2.1.1 If yes: Give the equipment to the next chef in the queue by sending an EQUIP_LEND message

        2.1.2 Else: Set the corresponding equipment as available

Notes:

-The Restaurant has 3 queues and 3 control variables, one for each equipment type. When the equipment is not available, the Chef that requested the equipment is added to the corresponding queue

Chefs – Simulation with 2 Chefs

In our simulations we first used one Chef, but then we added another one to allow for a faster simulation and to test whether or not our algorithms were robust enough to permit the inclusion of extra entities (as was requested by the professors).

When a new order comes in, the Clerk picks one of the existing chefs and gives it the order to process. The way we chose to implement how the chefs are assigned with the requests was via a simple ***Round Robin*** algorithm that chooses the chef by alternating between them.

In the end, each chef ends up with a similar amount of orders to process. This method was picked both due to it's simplistic nature, but also because it made sense to use it in this context: Since most requests have a similar processing time (i.e there's not going to be orders that take seconds to process and others that take several minutes), the Round Robin strategy allows for a performance boost without the hassle of having to implement a more sophisticated load balancing technique.

This modification lead to a much faster execution. In our tests with 20 clients and 2 Chefs the execution time was roughly half of the execution time with only 1 Chef.

In order to comply with the professor's guidelines we removed the second Chef due to consistency in evaluation, but our solution is robust enough to handle more than one Chef and it becomes more efficient in that way.