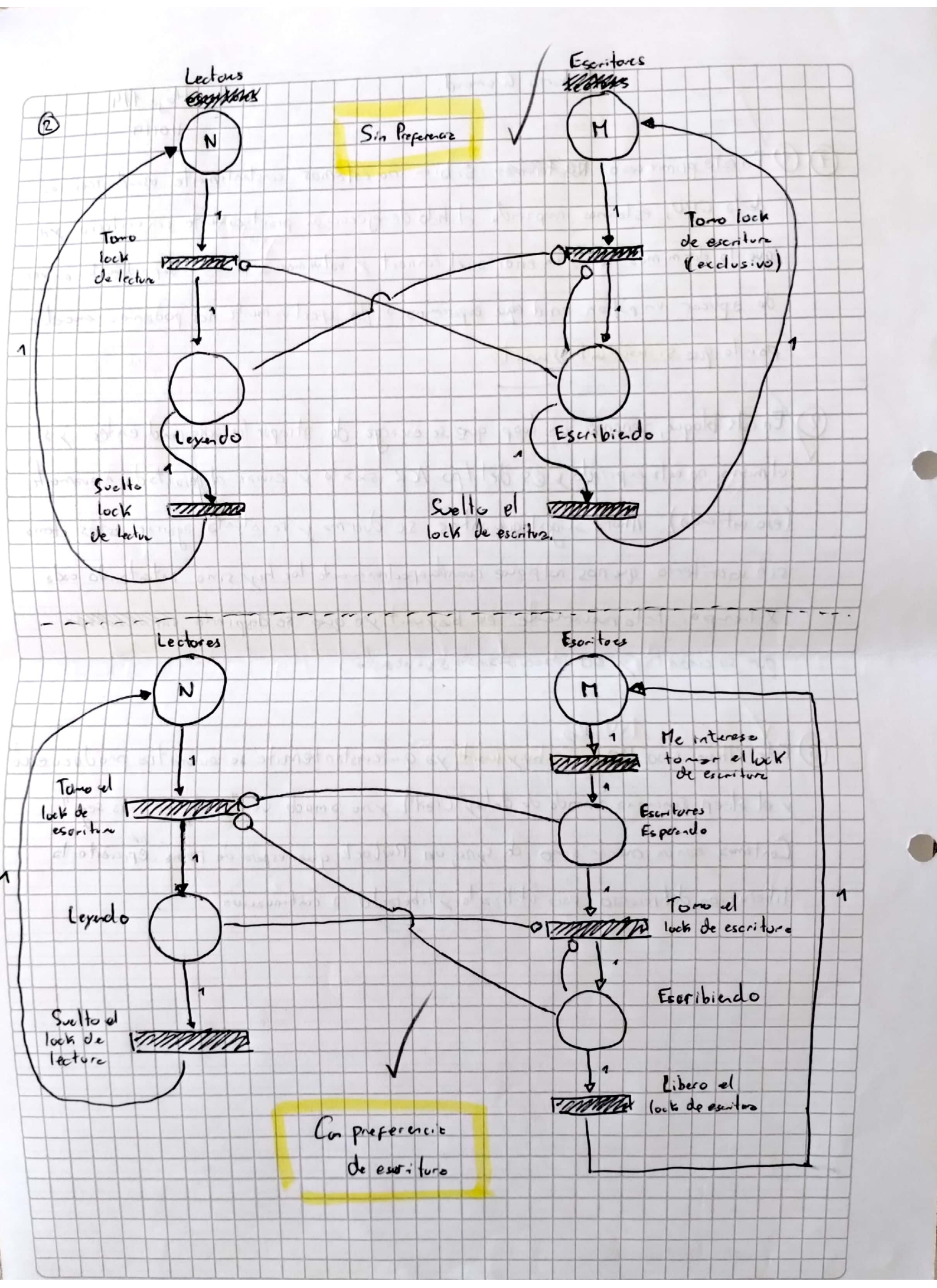
(1) (1) En este primer caso [illegible] si bien no estamos constantemente usando recursos de la CPU, estamos ocupando el hilo de ejecución prácticamente sin criterio ya que lo dormimos tras un error en el connect y volvemos a probar independientemente en vez de aplicar un patrón en el que esperemos a que efectivamente nos podamos conectar por lo que sí hay un busy wait.

(2) En este bloque, tenemos un loop que se encarga de atrapar los acks pendientes y si el mismo no está expirado y es del tipo ACK sacarlo y enviar el resultado nuevamente (eso entiendo). Ahora al igual que antes se duerme y reintenta agarrar estos items sin un criterio que nos notifique cuando efectivamente los hay sino intentando cada X tiempo. Esto nuevamente es busy wait ya que se despierta [illegible] por su cuenta y NO reaccionando a un evento

(3) Este último caso NO es ~~tiene~~ busy wait, ya que constantemente se encuentra produciendo y el sleep funciona a modo de delay (real) y no a modo de "pruebo en unos segs". Contamos con un mecanismo de sync, un RWlock que espera de forma eficiente la liberación del recurso para utilizarlo y liberarlo a continuación.



③ Mensajes:

[redacted]

- Llegoo (Addr < Cliente >) → ()
- Tomo Pedido ( ) → Addr < Mozo > → [redacted] Pedido
- Notifico Nuevo Pedido (Addr < Mozo >; Addr Pedido [redacted]) → ()
- Quiero Acceder Deposito ( ) → Addr < Cocinero > [redacted]
- Deposito Libre ( ) → ()
- Pedido Listo (Pedido, Comida) → ()
- Cocinar (Pedido)
- Pedir Cuenta (Addr < Cliente >) → [redacted] Cuenta
- Pago Cuenta (Addr < Cliente >, f64) → ().
  ↳ Dinero

Podría haber sido un ACTOR

Actores:

Cliente {

mozo_asignado : Option < Addr < Mozo > >

comida_deseada : Pedido

[redacted] recepción : Addr < Recepcion >

}

Recepción {

mozos : Queue < Addr < Mozo > >

}

Cocina {

cocineros : Queue < Addr < Cocinero > >

deposito : [redacted] Queue < Addr < Cocinero > >

}

Mozo {

pedidos : HashMap < Addr < Cliente >, Pedido >

cocina : Addr < Cocina >

}

Cocinero

pedidos : HashMap < Pedido, Addr < Mozo > >

[redacted]

ingredientes : HashMap < Ingrediente, Cantidad >

}

⑤ Un mozo podría hacer más de un pedido al mismo cocinero.

La idea sería la siguiente, los clientes llegan y notifican a la recepción pasando su address para ser atendidos. Luego la misma (de forma RR) le notifica a los mozos por el arribo de los clientes (Ambos casos con "Llego").

Los mozos le toman el pedido a los clientes y lo asocia al mismo con su dirección (un pedido por cliente), el cliente por su lado, responde el mensaje con el detalle del pedido y se queda a la espera de su comida.

Los mozos notifican el pedido a la cocina junto con su Address para que luego los pase a buscar. La cocina al igual que la recepción distribuye la carga de los pedidos mediante otra cola RR. Cuando los cocineros reciben los pedidos, ⑥ lo anotan junto a su mozo asignado y se ponen manos a la obra. Para ello si es que necesitan acceder al deposito solicitan el acceso al mismo y acceden de forma única gracias a que la cocina notifica a los cocineros para acceder y estos, devuelta para avisar que salieron. Una vez el pedido se encuentra listo, Se le notificará al mozo para que lo pase a buscar (Pedido Listo).

El mozo recibe el aviso, se lo entrega al cliente (forwardea el msg).

Por su parte los clientes reciben el pedido, comen y piden la cuenta al mozo. El mozo revisa el pedido, emite la cuenta y le responde al cliente a la espera de que este pague.

Por último el cliente efectivamente le paga a los mozos quienes muy amablemente le dan la despedida.

Diagrama en Hoja 4

Detalles:

- El cocinero toma los ingredientes cuando tiene acceso al deposito... en este caso asumo ingredientes infinitos
- Tanto la cocina como la recepción iteran de forma circular los Empleados, si el mailbox está lleno, se puede pasar al siguiente.
- Los empleados deberían agregarse luego de crear, faltaría un msg de ABM que los gestione.
- Los cocineros cuando tienen los ingredientes mandan a cocinar tantos pedidos como sea posible. (Auto Msg).

Es falso entonces... No todos se espera demon indep

4 a) Verdadero, cada proceso posee un Heap, Stack, Code, Data, Registros independientes

Viven en el espacio de memoria del proceso

cada hilo posee un Stack y una serie de registros ind.

cada task tan solo posee un contexto liviano independiente entre ellas que vive dentro de un hilo.

Entonces si, si bien comparten algunas cosas, cada uno tiene espacio de memoria independiente.

Es el EXECUTOR

b) Falso, es scheduler solo conoce Hilos y procesos, los runtimes (que corren dentro de un hilo de ejecución) orquestan que tareas se ejecutan a cada momento.

c) Falso, tan solo el Thread lo hace, las tareas asyncronas tienen un contexto en el que se guardan ciertos datos claves (Instruction pointer y wakers) pero el stack del mismo es compartido por todas las tareas en ejecución de UN thread.

d) En un ambiente de tan solo una CPU, para trabajo intensivo (sin esperas) usar un modelo de threads o de tareas (tardarán) esencialmente lo mismo. Segundo overheads de todo el tiempo NETO de procesamiento es Fijo y suponiendo $T_{ESPERA} = 0$ lo único que pesa es el overhead de cambiar de contexto, mantener las tareas y orquestar el orden (scheduler).

Es Falso entonces que la diferencia será sustancial y en esta situación iría por un estilo simple secuencial single thread.

⑤ a) Suponiendo que todos los archivos están en disco, usaría un modelo fork-join con una pool de threads para ir leyendo, procesando y escribiendo. Muy similar al TP, usaría Rayon para armar el sistema y usaría tantos threads como CPUs tenga mi procesador. Esto me permitiría minimizar los tiempos muertos y aprovechar todos los recursos de mi procesador.

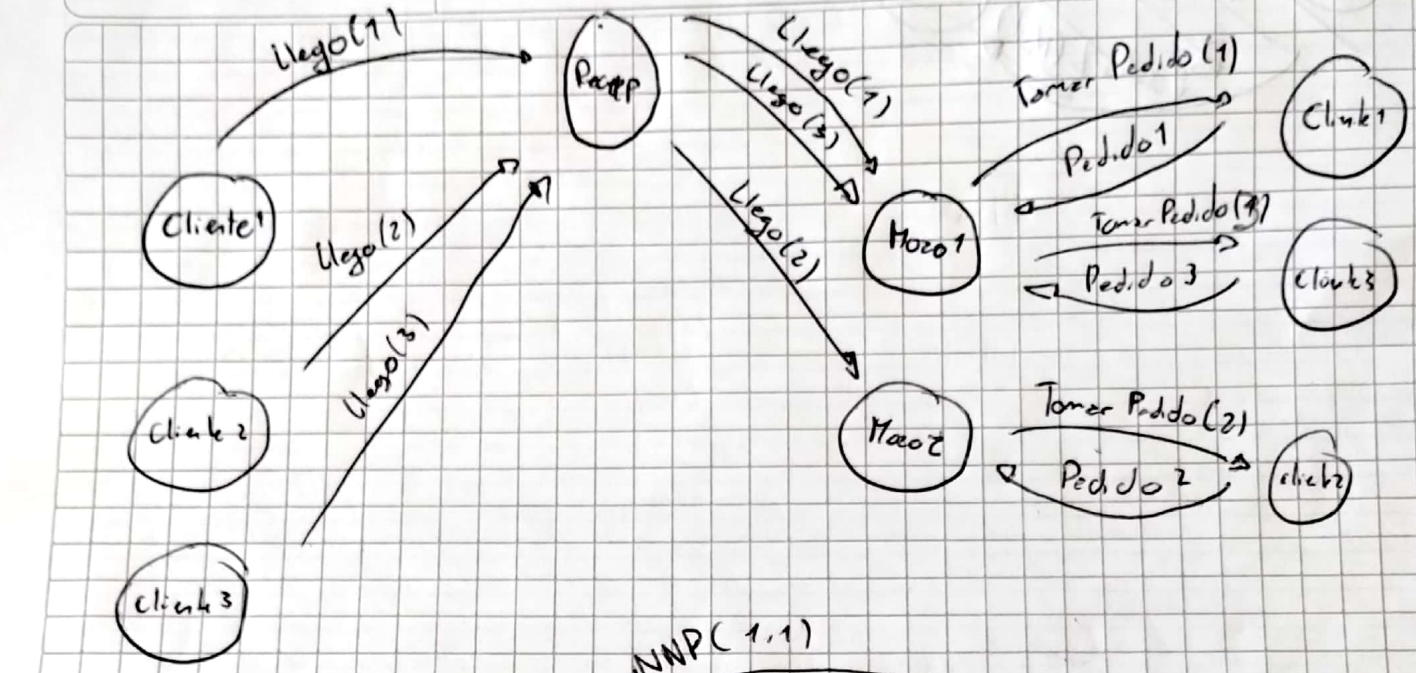
b) Suponiendo que tengo 1 servidor y N teléfonos conectados, podría crear tareas asincronas para cada teléfono que espere por la respuesta de con un timeout. Esto me permitiría no solo saber que clientes respondieron, sino que tendría sus rtas y la metadata de las mismas. Esto funcionaría como si de una barrera se tratase pero distribuida y con un timeout. → y para modificar/acceder a los resultados?

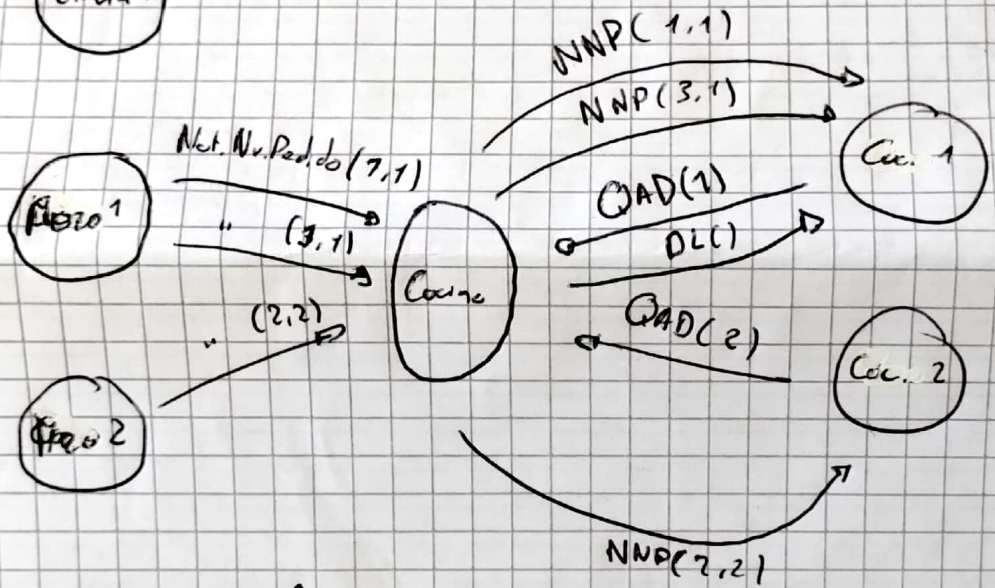
c) Podría usar una lógica de caché donde un thread [illegible] Suponiendo que [illegible], podría guardarme en HashMap<Rq, Resp> protegido por un RwLock. Este me permitiría compartirlo entre varios threads y de paso agregaría uno más que cada X tiempo limpie las Rq viejas o invalidas.
Cada uno de los threads de todas formas debería ser consiente que al acceder a la cache, podría haber datos erroneos, por lo que debería tras una operación de escritura (y suponiendo que se el servidor usa la BDD) limpiar las Rq afectada o todas según el caso.

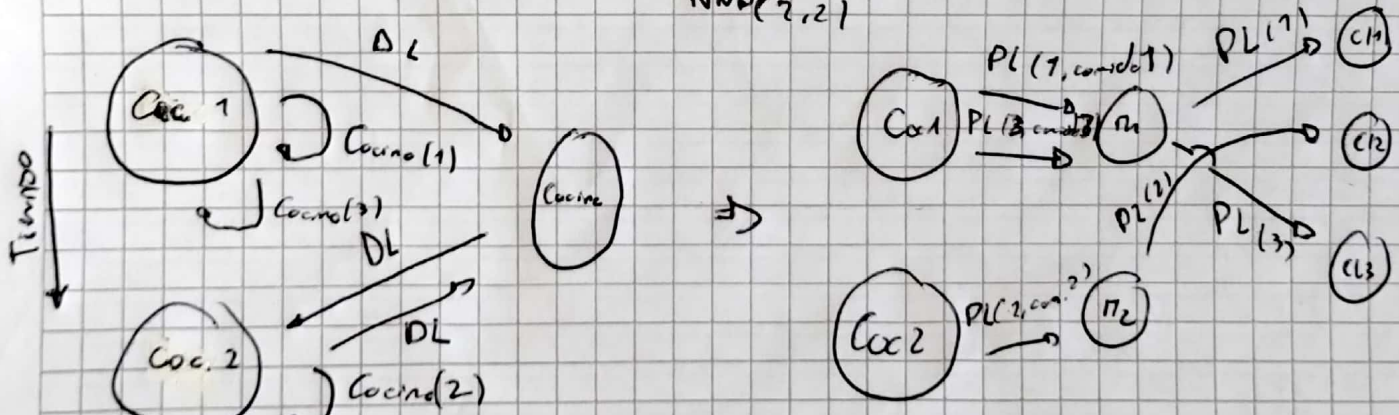
d) Si nos enfocamos en la parte del procesamiento podemos usar vectorización (suponiendo GPU) para ejecutar el modelo y minimizar los ciclos del reloj que tarde en procesar la request. A su vez deberá usar una estrategia multithreading (pool de threads) para manejar múltiples requests simultaneos. A priori no debería haber tiempo ocioso pues tan solo es procesamiento así que no necesitamos tareas asincronas ni nada por el estilo.
→ o asyncr tal vez sea más performante p/ atender requests

Máximo Grismaldi. Hoja 4/4

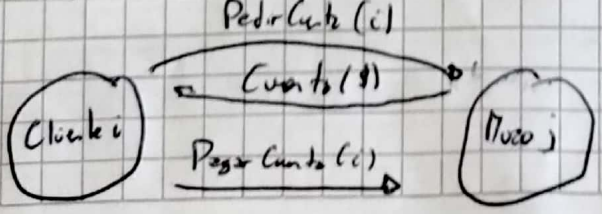
11C119







Pzos c/cliente



El mozo elimina el cliente de los pedidos.