SERVICIO STREAMING

getTweets(string mensaje) lee 25 tweets y los almacena en mensaje con el formato "$0tweet$1tweet$...$24tweet"

channel of string operation

Process streaming

```
        string pedido operation => (pedido)
        while(pedido != "FIN")
                if (pedido = "GET_TWEETS")
                        string mensaje
                        getTweets(mensaje)
                        tweets <= (mensaje)
                end if
                operation => (pedido)
        end while
end process
```

El servicio de *streaming* procesa el fichero de tweets eliminando los caracteres extraños y aquellos tweets que no contengas los campos necesarios y los envía al proceso *master* hasta que reciba un mensaje de fin de parte de este.

MASTER-WORKER

channel of (string) [1..N_WORKERS]worker_gestor

channel of (string) master_gestor, master_streaming

Process worker i := (1..N_WORKERS)

string recibido, datosTags, datosQoS

while(!finalizado)

worker_gestor(i) <- “READ_TAREAS, i”

worker_gestor(i) -> recibido

if(recibido = “FIN”)

finalizado := true

else

datosTags := procesar_tags()

datosQoS := procesar_QoS()

worker_gestor(i) <- “PUBLISH_TAGS, datosTags”

worker_gestor(i) <- “READ_QOS, datosQoS”

end if

end while

end process

Process master

string bloqueTweets, tweets

for i := 0..N_VECES

master_streaming <- “GET_TWEETS”

master_streaming -> bloqueTweets

tweets := procesar_tweets()

master_gestor <- tweets

end for

master_streaming <- “FIN”

master_gestor <- “FIN”

end process

Este modelo conta de dos tipos de proceso: el proceso *master* que se encarga de enviar las tareas y un conjunto de procesos *workers* que recibirán las tareas y se encargarán de llevarlas a cabo.

El proceso *master* se encarga de pedir los tweets a procesar y los recibe directamente del servicio de *streaming*, luego los envía al gestor de colas que los almacenará en la cola de tareas hasta que los *workers* empiecen a procesarlos. Como resultado de este procesamiento de mensajes, los *workers* enviarán la información obtenida sobre la calidad de servicio y de los tags al gestor de colas y este los almacenará en sus colas de QoS y de Tags respectivamente.

Además, será el proceso *master* el que se encargará de cerrar la comunicación. Se enviará un mensaje de fin al servicio de streaming y otro al gestor de colas, después este se encargará de informas a los *workers* y a los analizadores del cierre de la comunicación.

Los procesos *workers* estarán continuamente pidiendo nuevos tweets a procesar al gestor de colas y enviarán la información obtenida hasta que reciban el mensaje de fin del gestor de colas. Como resultado del procesamiento enviarán al gestor de colas información sobre el rendimiento de los propios procesos *worker* e información sobre los tweets que servirá a los procesos analizadores.

GESTOR DE COLAS

El gestor de colas proporcionará las herramientas necesarias para almacenar y gestionar todos los datos del resto de procesos.

Implementación: Primero, es necesario almacenar los datos que el resto de procesos envían. Para ello, se utiliza la cola genérica BoundedQueue ya dada.

Para evitar el solapamiento de entrada y salida de los datos en las colas, se usa un monitor cuyo esquema en alto nivel es el siguiente:

```
Monitor ControldeCola
        integer numElementos := 0
        boolean fin := false
        condition estaEscribiendo
        condition estaBorrando

        operation escribirCola( BoundedQueue cola, string elemento )
                while (numElementos >= MAX_ELEMENTOS)
                        waitC( estaEscribiendo )
                end while
                cola.encolar(elemento)
                numElementos := numElementos + 1
                signalC (estaBorrando)
        end operation

        operation leerCola(BoundedQueue cola, string elemento)
                boolean leido = false
                while (numElementos <= 0 && !fin)
                        WaitC(estaBorrando)
                end while
                if (numElementos > 0)
                        numElementos - -
                        cola.desencolar(elemento)
                        signalC(estaEscribiendo)
```

```
                leido= true;
            end if
            return leido
        end operation

        operation finalizar()
            fin := true
            signalC_all( estaBorrando )
        end operation
end monitor
```

El monitor está diseñado de tal forma que para cada cola que se necesite, se invoque un monitor.

El monitor cuenta con dos variables condición y dos operaciones. Si se quiere escribir en la cola primero se comprueba que la cola no esté llena y si no lo está se añade el elemento deseado al final de la cola. Para leer un elemento de la cola se comprueba primero que la cola no es vacía y si no lo es se guarda el primer elemento y después se desencola. Si fuera vacía, se devuelve falso en un booleano de control.

El programa principal crea dos sockets: uno para el master y los workers y otro para los analizadores. A continuación lanza N + 1 procesos master/worker (siendo N el número máximo de procesos que se pueden conectar) y dos procesos analizadores. A continuación se detalla el funcionamiento de ambos tipos de proceso:

```
channel of string socTareas, socAnalizadores
ControldeCola controlTareas, controlTags, controlQoS
BoundedQueue colaTareas, colaTags, colaQoS

Process masterWorker
    string mensaje
    bool out := false
    while (!out)
        socTareas => mensaje
        if (datos = “FIN”)
            out := true
        else if (mensaje = “PUBLISH_TAREAS,datos”)
```

```
                controlTareas.escribirCola(colaTareas, datos)
        else if (mensaje = “READ_TAREAS”)
                if (controlTareas.leerCola(colaTareas, datos) )
                        socTareas <= datos else mensaje := “FIN” socTareas <= mensaje
                end if
        else if (mensaje = “PUBLISH_QoS”)
                controlQoS.escribirCola(colaQoS, datos)
        else if (mensaje == “PUBLISH_TAGS”)
                controlTags.escribirCola(colaTags, datos)
        end if
    end while
end process
```

El proceso masterWorker es el encargado de gestionar la comunicación con los procesos master/worker dependiendo de lo que escuche por su canal. Por ejemplo si un proceso worker quiere leer tareas pendientes, se desencolara de la cola de tareas los datos correspondientes y se le enviaran estos. Si no quedaran tareas pendientes se le envia un mensaje de fin y se finaliza la comunicación.

```
Process analizadores
    string mensaje
    boolean out := false
    while(!out)
        socAnalizadores => mensaje
        if (mensaje = “READ_QoS”)
            if (controlQoS.leerCola(colaQoS, mensaje))
                socAnalizadores <= mensaje
            else
                mensaje := “FIN”
                socAnalizadores <= mensaje
                out := true
        else if (mensaje =“READ_TAGS”)
            if (controlTags.leerCola(colaTags, mensaje)
                socAnalizadores <= mensaje
```

```
                    else
                        mensaje := “FIN”
                        out := true
                    end if
                end if
            end while
        end process
```

El proceso analizadores se encarga de recibir peticiones de lectura por parte de los analizadores y enviar los datos correspondientes. Por ejemplo: si un analizador quiere leer un dato de la cola de QoS, enviará a través del canal socAnalizadores el mensaje “READ_QoS” y el gestor de colas desencolará el primer elemento que haya en la cola de QoS y se lo enviará al analizador.

ANALIZADORES

1. Analizador de tags

El analizador de Tags proporcionará información de los tweets desde diferentes perspectivas.

- Visión global:
  . Top 5 Clientes más utilizados para acceder a twitter
  · Top 5 Clientes más utilizados para escribir tweets
  · Top 5 Clientes más utilizados al escribir tweets con tags
  · Top 5 Clientes más utilizados al escribir tweets con menciones
  · Top 5 Clientes más utilizados para hacer retweet

- Visión desde los tags:
  · Top 10 Hashtags más utilizados durante la vida del sistema
  · Top 10 Hashtags más utilizados en retweets
  · Top 10 Hashtags más utilizados en tweets con menciones
  · Representación gráfica con evolución temporal

- Visión desde los usuarios (por cada tag del top 10):
  · Top 10 usuarios con más actividad con el tag
  · Top 10 usuarios con más tweets escritos con el tag
  · Top 10 usuarios con más retweets con el tag
  · Top 10 usuarios con más menciones en tweets con el tag
  · Representación gráfica con evolución temporal

Implementación:

Para llevarlo a cabo, habrá un proceso principal que se comunicará con el gestor de colas para recibir la información de los tags.

Lo irá almacenando en un buffer de tamaño BUFFSIZE. También hay un proceso tagsAnalyzer y un proceso globalAnalyzer que leerán del buffer la información extraída en concurrencia y la procesarán.

tagsAnalyzer guardará los resultados en una lista enlazada en la que cada nodo contendrá el Hashtag que es, el número de apariciones totales, en retwteets y en tweets con menciones, y para el total de apariciones, los retweets y los tweets con mención, un string con el siguiente esquema:
“fechaMasAntigua;AutorPrimerTweet/fecha4;ATweet/…”
“fechaMasAntigua;AutorPrimerRetweet/…” “fechaMasAntigua;AutorPrimerMenciont/…”

globalAnalyzer guardará también los resultados en una lista enlazada en la que cada nodo tendrá el cliente que es, número de usos totales del cliente con el que se accede a twitter, con el que se escriben twteets, con el que se escriben tweets con tags, con el que se escriben tweets con menciones y con el que se hace retweet. Además habrá un string asociado a cada número, que guardará el registro del momento en que se utilizó el cliente para las distintas acciones, por ejemplo: “fechaMasAntigua/fecha4/…”

Una vez hayan terminado de leer ambos procesos el contenido del buffer, el proceso principal reescribirá el buffer con nuevos contenidos.
Se continúa con el mismo funcionamiento hasta que no hay más datos que procesar, determinado por una marca de fin que mandará el gestor de colas.
Una vez se finalice, se lanzarán 10 procesos userAnalyzer que producirán los resultados de cada tag del top 10 obtenidos anteriormente para los 10 usuarios más activos con el Hashtag.

Cada userAnalyzer contendrá una lista enlazada con una estructura similar a las 2 listas anteriores: un identificador, y pares formados por un entero que representa un número de acciones, y un string que registra cada accion leida en el tiempo.

Finalmente, el proceso Cliente, leerá de cada lista generada por cada proceso descrito la información necesaria para mostrar por pantalla las conclusiones que mostramos al principio.
Un acercamiento en pseudocódigo de las listas anteriores sería lo siguiente:

```
TAD lista enlazada resultadosTags {
        struct nodo {
                string Hashtag;
                interger apHashtag;
                string quienAH;
                interger apHashtagRT;
                string quienAHRT;
                interger apHashtagM;
                string quienAPM;
        }
        nodo* init;
        interger totalHashtags;
};
TAD lista enlazada resultadosGlobales {
        struct nodo {
                string Client;
                interger usoClient;
                string cuandoUC;
                interger usoTweetC;
                string cuandoUTC;
                interger usoTTagC;
                string cuandoUTTC;
```

***Nota: TODOS los procesos involucrados en el analizador, tendrán como memoria compartida los siguientes datos:**

- el buffer que se gestiona su uso con el monitor contado a continuación y su entero size que indica hasta dónde hay datos en el buffer.

- los punteros a las diversas listas enlazadas contadas anteriormente.

```
            interger usoTMencionC;
            string cuandoUTMC;
            interger usoRTC;
            string cuandoURTC;
        }
        nodo* init;
        interger totalClientes;
};
```

Para la gestión de la sincronización del buffer con tagsAnalyzer, globalAnalyzer y el principal, hemos pensado en usar un monitor gestorBuffer que tendrá como recursos compartidos:

- Un contador “listo” inicializado a 0 para identificar cuando han completado de leer los 2 procesos

- Una variable booleana estaEscribiendo inicializada a falso para avisar que el principal está escribiendo en el buffer.

***Nota: en el constructor del monitor, al inicializarlo, se rellena el buffer con lo leído por el principal.**

```
Monitor gestorBuffer
        condition esperandoLectores
        condition esperandoEscritor

        operation quieroLeerBuffer()
            if ( estaEscribiendo = true )
                waitC(esperandoLectores)
            end if
        end operation
operation dejoLeerBuffer()
        listo:= listo + 1
        if (listo = 2)
            signalC(esperandoEscritor);
        end if
end operation
```

```
operation quieroEscribirBuffer()
        if ( listo != 2 )
                waitC(esperandoEscritor)
        end if
        estaEscribiendo:= true
end operation

operation dejoEscribirBuffer()
        listo:= 0
        estaEscribiendo:= false
        signalCAll(esperandoLectores)
end operation
```

Los userAnalyzer no necesitan ninguna gestión de sincronización ya que no son dependientes entre sí y únicamente leen los datos compartidos. Es un trabajo en paralelo para agilizar la tarea.

Un esquema de alto nivel de los diferentes procesos que intervienen:

```
channel of string c

Process main
        string mensaje
        c => mensaje
        while (mensaje != "FIN")
                quieroEscribirBuffer()
                //escribir(c,mensaje,buffer,size)
                dejoEscribirBuffer()
        end while
end process

Process globalAnalyzer
        // TAD lista enlazada resultadosGlobales
```

Nota: el main lanza todos los procesos involucrados y los lanza en el siguiente orden:

start

lanzo procesos globalAnalyzer y tagsAnalyzer

(concurrencia)

fin procesos anteriores

lanzo procesos userAnalyzer (i:1..10)

(trabajo en paralelo)

fin procesos

lanzo proceso cliente

end

```
        boolean end:= false
        while (end = false)
                quieroLeerBuffer()
                //leer(end,resultadosGlobales,buffer,size)
                dejoLeerBuffer()
        end while
end process

Process tagsAnalyzer
        // TAD lista enlazada resultadosTags
        boolean
        end:= false
        while (end = false)
                quieroLeerBuffer()
                //leer(end,resultadosTags,buffer,size)
                dejoLeerBuffer()
        end while
end process


Process userAnalyzer (i:1..10)
        // TAD lista enlazada resultadoUser i
        // producirResultados(resultadoUser i, resultadosTags)
end process

Process Cliente
        // mostrarResultadosGlobales(resultadosGlobales)
        // mostrarResultadosTags(resultadosTags)
        // mostrarGraficas(resultadosTags)
        for (i:1..10)
                // mostrarResultadosUser(resultadoUser,i)
```

```
                // mostrarGraficas(resultadosUser,i)
        end for
end process
```

2. Analizador de rendimiento

El analizador de rendimiento pretende mostrar los siguientes resultados:

- tiempo medio que le cuesta resolver a cada worker 200 bloques de tareas
- gráfico temporal (para cada worker) tiempo de ejecución cada 200 bloques
- worker más lento y el más rápido (rango en que oscila el tiempo de ejecución medio)
- bloque más lento y más rápido

Implementación:

Se creará un proceso que se comunicará con el gestor de colas e irá guardando los resultados de cada worker en terminar un bloque en una variable que tras llegar a haber acumulado 200, se calculará la media y se entregará la información de cada worker en un nodo de la lista enlazada que utilizaremos (el cual contiene el identificador del worker, un string con la estructura: "primerBloqueDoscientos/bloque56Doscientos/…") y un entero donde se guarda la media. Además se irá actualizando conforme se completen bloques de 200 almacenado en las variables.
Además al calcular la media en las variables, se tendrá una variable que guardará el bloque más rápido y el más lento que se tendrá que comparar a cada 200 bloques completos.

Al finalizar el sistema habrá un proceso cliente que creará un informe que se mostrará por pantalla.

Un esquema de alto nivel de los procesos:

channel of string c

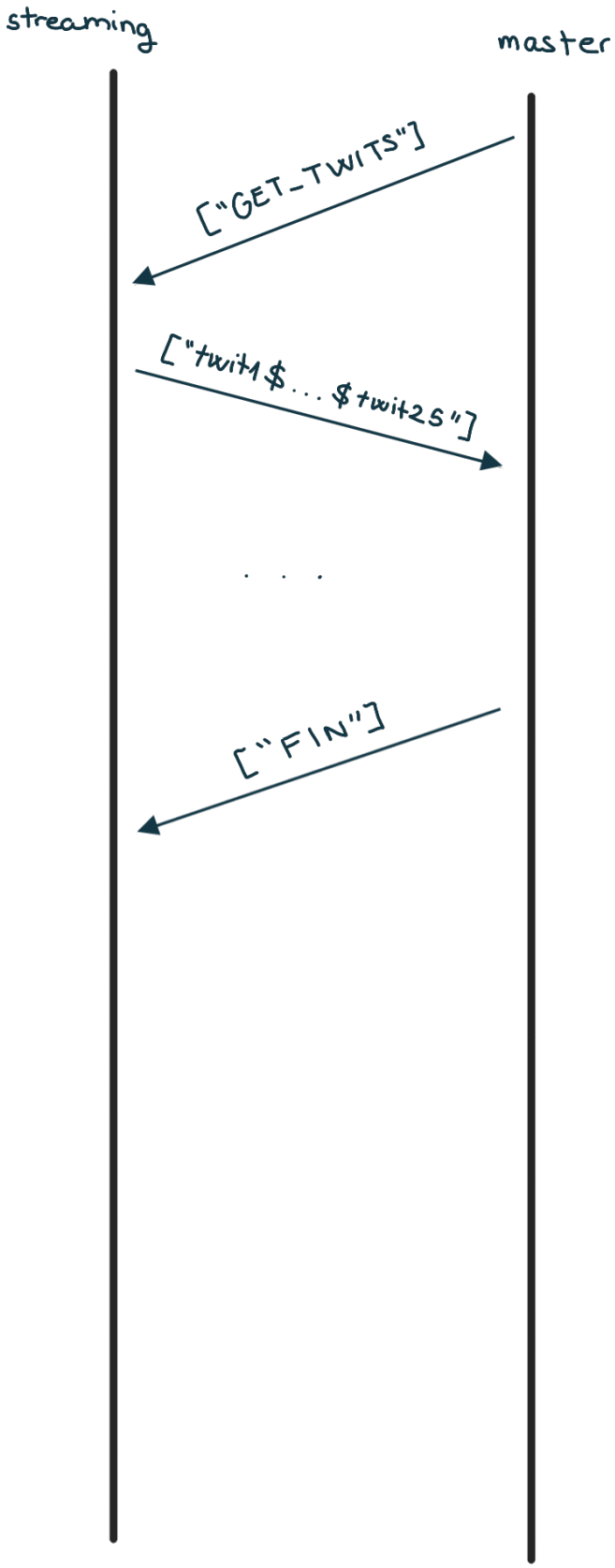
Process AnalizadorRendimiento

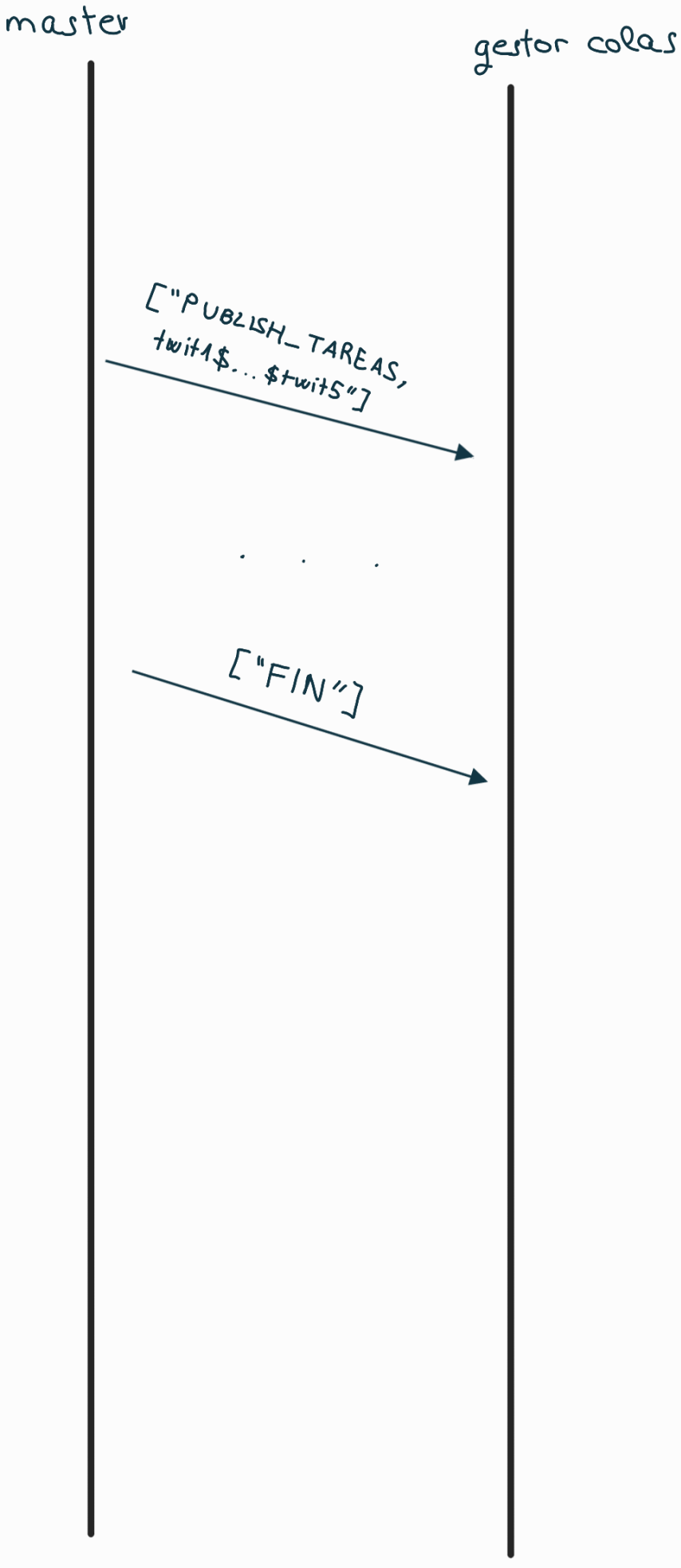
```
string mensaje
struct auxWorker {
        interger acumworker
        interger numBloque
}
struct array auxWorker workers[5]
c => mensaje
while (mensaje != "FIN")
        tratarMensaje(mensaje,workers,timeWorkers,bestTime,worstTime)
        // para averiguar donde guardar mensaje y si se
        // llega alguno del vector a numBloque = 200 se divide su acumWorker/200
        // y se manda a la lista enlazada
```

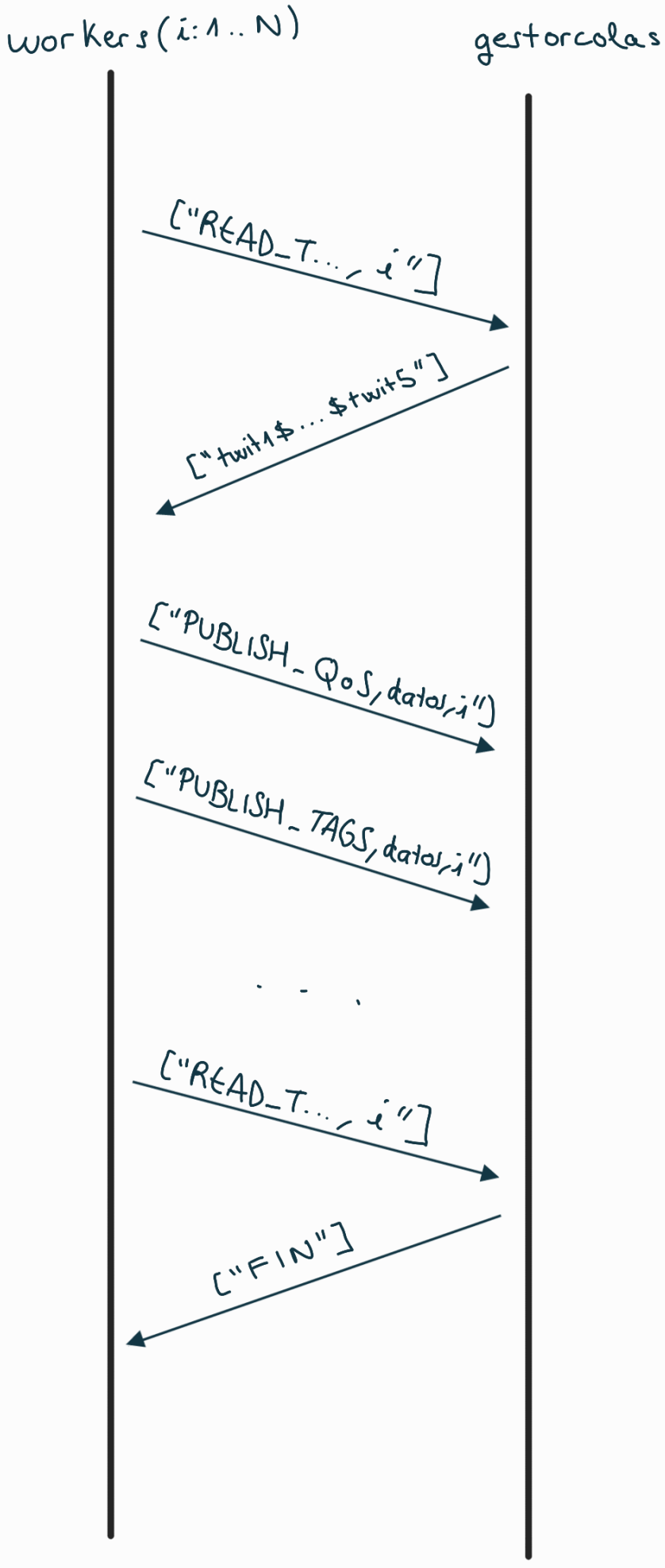
***Nota: el cliente y el analizador de Rendimiento tendrán en común la lista con los worker llamada timeWorkers y las 2 variables bestTime y worstTime.**

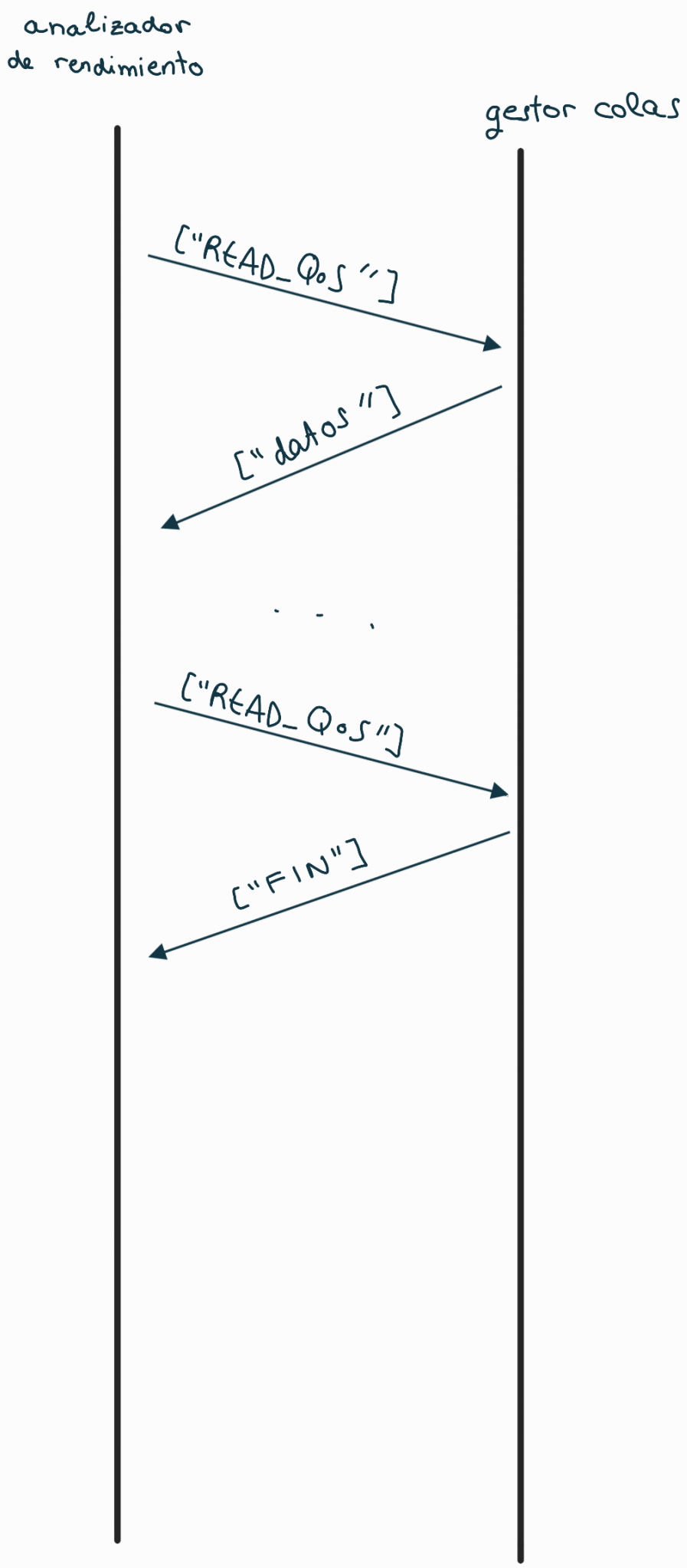
```
        end while
end process


Process Cliente
        // mostrarRendimientoMedio(timeWorkers)
        // mostrarGraficas(timeWorkers)
        // mostrarRangoTimeWorkers(timeWorkers)
        // mostrarMasLentoYRapido(bestTime,worstTime
end process
```

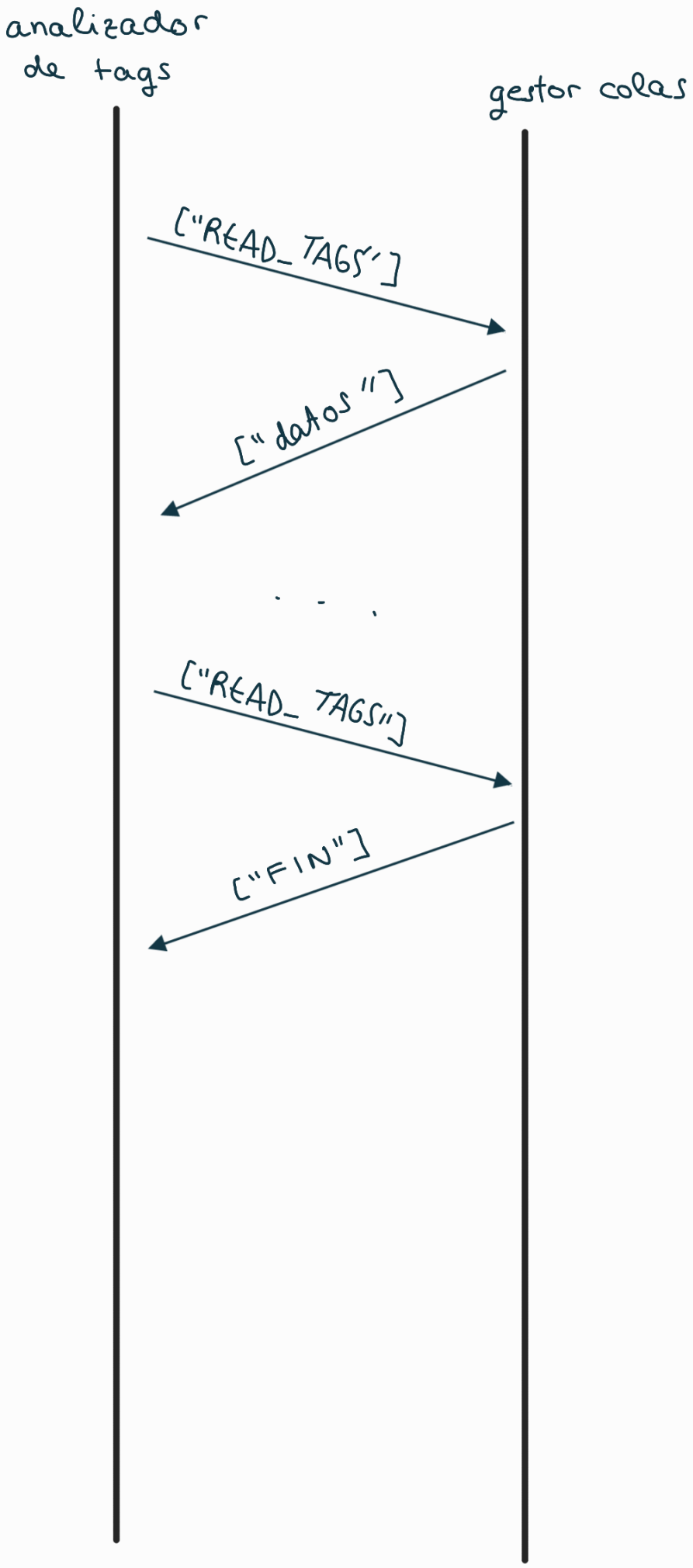
DISEÑO DE COMUNICACIONES




master
gestor colas
["PUBLISH_TAREAS,
twit1$...$twit5"]
. . .
["FIN"]


workers(i:1..N)
gestorcolas
["READ_T..., i"]
["twit1$...$twit5"]
["PUBLISH_QoS, data1, i"]
["PUBLISH_TAGS, data1, i"]
["READ_T..., i"]
["FIN"]


analizador
de rendimiento
gestor colas
["READ_QoS"]
["datos"]
["READ_QoS"]
["FIN"]


analizador de tags
gestor colas
["READ_TAGS"]
["datos"]
. . .
["READ_TAGS"]
["FIN"]