Game of Servers

Propuesta

Somos

- Claudia Puentes Hernández (@ClauP99) 🦁,
- Omar Alejandro Hernández Ramírez (@OmarHernandez99) 🐯,
- Andy Ledesma García (@MakeMake23) 🐱 y
- Mauricio Mahmud Sánchez (@maux96) 🐱

y proponemos que el proyecto conjunto de Simulación, Compilación e IA sea sobre servidores y se llame ***Game of Servers***.

La idea va de simular un entorno con una cantidad determinada de servidores y un número potencialmente infinito de clientes. Los clientes emitirán pedidos a los servidores y estos responderán en consecuencia 😁 o no 😔, como sucede en la realidad.

El usuario de nuestro proyecto podrá programar cada uno de los servidores para que responda a los pedidos según crea conveniente. Esto se realizará en un lenguaje creado por nosotros para este dominio específico 🤓.

Un servidor también puede emitir pedidos a otro servidor 🤯, convirtiéndose el primero en un cliente del segundo. En este sentido, se pudieran aplicar algoritmos de IA 😳 para enrutar el pedido de forma óptima entre servidores.

En un sistema como este se pueden simular:

- ataques DoS y DDoS
- pérdidas de usuarios y capital en servicios online por demora en las respuestas
- distintas estrategias de ruteo y de distribución de carga
- el accionar de cada uno de servers, como agentes autónomos
- la viabilidad del sistema en conjunto en cuanto a la tolerancia a fallas, alta disponibilidad.

Incluyendo IA allá donde puede ser más útil 😁.

1ra Entrega

En la primera entrega del proyecto simulamos el procesamiento de pedidos en un sistema compuesto por un servidor (repartidor de carga) que selecciona cuál de los servidores restantes (*doers*) se encargará de procesar el pedido entrante. En la sección Modelo de un Líder y muchos Seguidores se explica en detalle cómo modelamos este sistema.

Esta simulación es ejecutada múltiples veces por un algoritmo genético, con el objetivo de determinar el número de *doers* necesarios para minimizar el tiempo de respuesta a los pedidos.

El algoritmo genético, a su vez, es ejecutado por una aplicación de consola llamada `gos` que recibe sus parámetros del archivo `appsettings.json`. En la sección Ejecutando gos se explica cómo se ejecuta el programa y el significado de cada parámetro.

Ejecutando `gos`

Para ejecutar nuestro programa, descargue el *release* para su sistema operativo y abra el archivo `gos` (Linux) o `gos.exe` (Windows) desde una terminal.

Los parámetros deben ser configurados en el archivo `appsettings.json`. Estos son

- `Followers`: cantidad de *doers*.
- `Lambda`: parámetro lambda de la distribución exponencial para determinar tiempos de ocurrencia de los eventos.
- `CloseTime`: tiempo de cierre del sistema (T). Cuando se arribe a este tiempo, no se recibirán más pedidos.
- `MonthlyMaintenanceCost`: costo mensual máximo de mantenimiento del sistema.
- `RunTimeMilliseconds`: tiempo en milisegundos de corrida de la metaheurística.
- `Poblation`: número de individuos del algoritmo genético.

Modelo de un Líder y muchos Seguidores

El sistema de la simulación fue modelado mediante dos capas conectadas en serie: la del repartidor de carga (líder) y la de los *doers* (seguidores). Estos últimos procesan los pedidos en paralelo.

A continuación se definen las variables y los eventos de la simulación.

Variables

- Variables de tiempo
  - t - tiempo general.
  - t_{A_1} - siguiente tiempo de arribo al líder.
  - t_{A_2} - siguiente tiempo de arribo a los seguidores.
  - t_{D_i} - siguiente tiempo de salida del i-ésimo seguidor.
- Variables contadoras
  - N_A - cantidad de arribos
  - N_D - cantidad de partidas
  - A_1 - Diccionario de tiempos de arribo al líder
  - A_{d_x} - Lista de diccionarios donde $A_{d_i}[j] = t_j$, siendo A_{d_i} el diccionario correspondiente al i-ésimo seguidor y t_j el tiempo de partida asociado al 'cliente' j-ésimo.
- Variables de estado
  - n_1 - número de clientes en el líder.
  - n - número de clientes en el sistema.
  - F_s - servidores libres.
  - q - cantidad de 'clientes' esperando en la cola de los seguidores.

Eventos

- **Arribo al líder** $(t_{A_1} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, \dots) \wedge t_{A_1} < T)$:
  - $t = t_{A_1}$
  - $N_A = N_A + 1$
  - $n_1 = n_1 + 1$
  - $n = n + 1$
  - $generar\ \ t_{A_L} \wedge \ \ t_{A_1} = t + t_{A_L}$
  - $if\ (n_1 == 1)\ \ then\ \ \ generar\ \ t_{A_S}\ \wedge\ t_{A_2} = t + t_{A_S}$
  - $A_1[N_A] = t$
- **Arribo a los seguidores** $(t_{A_2} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, \dots) \wedge t_{A_2} < T)$:
  - $t = t_{A_2}$
  - $n_1 = n_1 - 1$
  - $if\ (n_1 \neq 0)\ \ then\ \ \ (generar\ t_{A_S}\ \wedge\ t_{A_2} = t + t_{A_S})$
  - $else\ \ \ t_{A_2} = \infty$
  - $if\ (|F_s| == 0)\ \ then\ \ \ (q = q + 1)$
  - $else$:
    - $serv = F_s.\,Dequeue()$
    - $client = N_A - n_1$
    - $generar\ t_{D_S}\ \ \wedge\ \ t_{D_{serv}} = t + t_{D_S}$
    - se inserta $client$ en $serv$
- **Partida** $(min(t_{D_1}, t_{D_2}, \dots) == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, \dots)) \wedge (min(t_{D_1}, t_{D_2}, \dots) \leq T$:
  - $t_{Dmin} = min(t_{D_1}, t_{D_2}, \dots)$
  - $serv = ObtenerServidorPartida()$
  - $client = OptenerClienteQueParte()$
  - $t = t_{Dmin}$
  - $N_D = N_D + 1$
  - $n = n - 1$
  - $if\ (q \neq 0)\ \ then$:
    - $q = q - 1$
    - $client = N_A - q$
    - $generar\ t_{D_S}\ \ \wedge\ \ t_{D_{serv,client}} = t + t_{D_S}$
  - $else\ \ \ F_s.\,Add(serv)$
  - $A_{d_{serv}}[client] = t_{Dmin}$
- **Arribo fuera de tiempo para el líder**
  $(t_{A_1} \neq \infty \wedge t_{A_1} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, \dots) \wedge t_{A_1} > T)$:
  - $t_{A_1} = \infty$
- **Arribo fuera de tiempo para los seguidores**
  $(t_{A_2} \neq \infty \wedge t_{A_2} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, \dots) \wedge t_{A_1} > T)$:
  - $t_{A_2} = \infty$
- **Cierre** $(min(t_{D_1}, t_{D_2}, \dots) == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, \dots))\ \ \wedge\ \ ((min(t_{D_1}, t_{D_2}, \dots) > T)\ \wedge$
  $((min(t_{D_1}, t_{D_2}, \dots) \neq \infty)\ \ \wedge\ \ n > 0$:

  El evento de cierre es análogo al evento de partida.

2da Entrega

Gramática

```
<program> := <stat-list>

<stat-list> := <stat> ";"
             | <stat> ";" <stat-list>
               | <block-stat>
             | <block-stat> <stat-list>

<block-stat> := <if>
              | <def-func>

<stat> := <let-var>
        | <print-stat>
        | <return>
        | <func-call>

<let-var> := "let" ID "=" <expr>

<def-func> := "fun" ID "(" <arg-list> ")" "{" <stat-list> "}"

<print-stat> := "print" <expr>

<arg-list> := ID
            | ID "," <arg-list>

<cond> := <math> "<" <math>
        | <math> ">" <math>
        | <math> "==" <math>

<expr> := <cond>
        | <math>

<math> := <math> "+" <term>
        | <math> "-" <term>
        | <term>

<term> := <term> "*" <factor>
        | <term> "/" <factor>
        | <factor>

<factor> := <atom>
          | "(" <math> ")"

<atom> := NUMBER
        | ID
        | <func-call>

<func-call> := ID "(" <expr-list> ")"

<expr-list> := <expr>
             | <expr> "," <expr-list>

<if> := "if" <cond> "{" <stat-list> "}"

<return> := "return" <expr>
```

El `;` lo pone el *lexer*, no es necesario que el usuario lo haga. Este puede emplear `\` para definir *statements* de más de una línea.

Reglas Semánticas

Una variable solo puede ser definida una vez en todo el programa.

- Los nombres de variables y funciones no comparten el mismo ámbito (pueden existir una variable y una función llamadas igual).
- No se pueden redefinir las funciones predefinidas.
- Una función puede tener distintas definiciones siempre que tengan distinta cantidad de argumentos.
- Toda variable y función tiene que haber sido definida antes de ser usada en una expresión (salvo las funciones pre-definidas).
- Todos los argumentos definidos en una misma función tienen que ser diferentes entre sí, aunque pueden ser iguales a variables definidas globalmente o a argumentos definidos en otras funciones.
- En el cuerpo de una función, los nombres de los argumentos ocultan los nombres de variables iguales.