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** $(t_{A_1} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, ...) \wedge t_{A_1} < T)$:
  - $t = t_{A_1}$
  - $N_A = N_A + 1$
  - $n_1 = n_1 + 1$
  - $n = n + 1$
  - $generar \;\; t_{A_L} \wedge \;\; t_{A_1} = t + t_{A_L}$
  - $if \;\; (n_1 == 1) \;\; then \;\; generar \;\; t_{A_S} \;\; \wedge \;\; t_{A_2} = t + t_{A_S}$
  - $A_1[N_A] = t$
- **Arribo a los seguidores** $(t_{A_2} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, ...) \wedge t_{A_2} < T)$:
  - $t = t_{A_2}$
  - $n_1 = n_1 - 1$
  - $if \;\; (n_1 \neq 0) \;\; then \;\; (generar \; t_{A_S} \;\; \wedge \;\; t_{A_2} = t + t_{A_S})$
  - $else \;\; t_{A_2} = \infty$
  - $if \;\; (|F_s| == 0) \;\; then \;\; (q = q + 1)$
  - $else$:
    - $serv = F_s.Dequeue()$
    - $client = N_A - n_1$
    - $generar \; t_{D_S} \;\; \wedge \;\; t_{D_{serv}} = t + t_{D_S}$
    - se inserta $client$ en $serv$
- **Partida** $(min(t_{D_1}, t_{D_2}, ...) == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, ...)) \wedge$ $(min(t_{D_1}, t_{D_2}, ...) \leq T$:
  - $t_{Dmin} = min(t_{D_1}, t_{D_2}, ...)$
  - $serv = ObtenerServidorPartida()$

- $client = OptenerClienteQueParte()$
- $t = t_{Dmin}$
- $N_D = N_D + 1$
- $n = n - 1$
- $if\ (q \neq 0)\quad then:$
  - $q = q - 1$
  - $client = N_A - q$
  - $generar\ t_{DS}\quad \wedge\quad t_{D_{serv,client}} = t + t_{DS}$
- $else\quad F_s.Add(serv)$
- $A_{d_{serv}}[client] = t_{Dmin}$

- **Arribo fuera de tiempo para el líder** $(t_{A_1} \neq \infty \wedge t_{A_1} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, ...) \wedge t_{A_1} > T)$:
  - $t_{A_1} = \infty$
- **Arribo fuera de tiempo para los seguidores** $(t_{A_2} \neq \infty \wedge t_{A_2} == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, ...) \wedge t_{A_1} > T)$:
  - $t_{A_2} = \infty$
- **Cierre** $(min(t_{D_1}, t_{D_2}, ...) == min(t_{A_1}, t_{A_2}, t_{D_1}, t_{D_2}, ...))\quad \wedge\quad ((min(t_{D_1}, t_{D_2}, ...) > T)\ \wedge\ ((min(t_{D_1}, t_{D_2}, ...) \neq \infty)\quad \wedge\quad n > 0$:

  El evento de cierre es análogo al evento de partida.

2da Entrega

Sobre la Simulación

Se dejó de utilizar el modelo de eventos discretos y se emplea ahora un modelo de agentes.

Ambiente

El ambiente tiene variables configurables (por ahora constantes) del tiempo de llegada de Request y Response, una lista de todos los agentes en la escena, el tiempo actual de la simulación, los responses (una vez haya acabado la simulación) a los request del cliente que se hacen en un inicio y una estructura de datos Heap (de mínimos) encargado de llevar acabo el funcionamiento de la línea de tiempo correctamente haciendo que transcurra este.

Agentes

Los agentes son los servidores y estos se dividen en 3 tipos:

- *workers* simples
- *workers* interactivos
- distribuidores de carga

Se modeló un comportamiento de agente con estados, en el cual también incluimos el concepto de percepciones, este concepto lo utilizamos para representar el comportamiento de los request y response que se realizan durante la simulación.

Para comenzar se realizó una clase **Agent** que es de la que van a heredar todos los servidores, entre estos se encuentra **Worker**, **InteractiveWorker**, **Distributor**. La clase **Agent** posee las principales propiedades de estos servidores, el ID de cada uno, un environment, un status y unas listas de funciones que son las que van a modelar de alguna manera el comportamiento de la función **next** que se le pasa un status, un perception y devuelve un nuevo status. Esta clase posee los funcionamientos de **HandleRequest**, **HandleResponse**, **HandleStatus** que son los encargados de llevar a cabo el funcionamiento del servidor cuando llega un request, un response o el servidor necesita cambiar su estado interno basado en algo como el tiempo, esto se logra recorriendo cada una de las listas de funciones respectivas donde cada una hace un pequeño cambio en el estado del agente o del ambiente. En el caso del **HandleRequest**, antes de realizar lo anterior se verifica si el server está disponible para poder procesar el request.

La clase **Distributor** posee una lista de workers y un protocolo representado como una función de selección, la cual es encargada de seleccionar el worker o server al que le vamos a enviar el request. Esta clase tiene funcionalidades como añadir los workers, enviar un request , que sería añadir el request a la lista de procesamiento de request en el

estado, con el protocolo de selección escogemos el server al cual le vamos a enviar el request y subscribimos el evento al estado. Otra de las funcionalidades es chequear el response, si ya el response está disponible pues se envía el request original si no, se selecciona otro worker y se envía el request.

La clase **InteractiveWorker** tiene un diccionario que posee las posibles necesidades del request y a cada una de estas se le asocia una lista de agentes que pueden dar response a este tipo de request, tiene otro diccionario que contiene todos los request que han llegado hasta el momento y a cada uno de estos se le asocia una lista de todos los request que fueron necesarios realizar para satisfacer sus necesidades y por último se tiene otro diccionario que contiene el id del request original contra el response que se irá conformando poco a poco. Aquí tenemos dos funcionalidades la primera es que si llegó un request revisamos si se encuentra en el diccionario de necesidades, si es asi enviamos un request a cada uno de los agentes que se encuentra en la lista de agentes asociada a la necesidad del request y la otra funcionalidad es a la hora en que llega un response, revisamos de cual request es y se verifica si ya tenemos todos los responses necesarios para realizar el response del request original, si es así realizamos el proceso y si no continuamos hasta obtenerlo completo.

La clase **Worker** posee varias funcionalidades como **GettingRequest** que revisa si el servidor puede atender el request, si no puede envía un response de servidor no disponible y si no subscribe el evento. Otra de las funcionalidades es el procesamiento el request que si el servidor esta disponible añade el request a la lista de procesamiento de request y en dependencia de la cantidad de request que tenga el servidor a procesar cambia el estado a no disponible. El método **SendResponse** conforma el response de acuerdo al request original y lo envía y **SetAvailableAfterSendResponse** luego de enviar el response de un request se elimina este de la lista de request en procesamiento

Status

La clase **Status**, representa el estado interno de un Servidor por ahora, tiene como objetivo guardar valores necesarios en la ejecución de la simulación y también guardar estadísticas. Este es modificado en las funciones de manejo de Request, Response u Observer. Tiene referencias a su agente correspondiente, y permite la suscripción de eventos a la línea temporal.

Eventos de la Línea Temporal

Estas 3 clases a continuación, representan cada vez que ocurre algo en la simulación, **Request** representa la llegada de un request, **Response** la llegada de un response y **Observer** el conocimiento de un cambio en el estado interno de un agente. Estos se agregan al Heap del ambiente a medida que van apareciendo y cuando es su turno de salir del ambiente, se "ejecutan" en su agente correspondiente variando el funcionamiento del mismo de acuerdo a sus valores internos.

La clase **Request** posee características como su ID que es único para cada request, la URL, el sender (quien lo envió) y el tiempo en que fue enviado. Son procesados por los agentes en el método que llama a HandleRequest para cambiar de estado y otro método para conformar el response y suscribirlo al ambiente.

La clase **Response** posee propiedades similares a la clase **Request** como su ID, el sender, el que lo recibe, el tiempo, el cuerpo de la respuesta y dos booleanos para conocer si el response está disponible o no. Son procesados por un agente al igual que en **Request** en vez de ser HandleRequest es HandleResponse.

La clase **Observer** tiene el agente el cual va (no necesariamente) a cambiar el estado, un objeto genérico (por ahora) el cual nos sirve para saber qué tipo de cambio en el estado interno del agente se va llevar a cabo y como los anteriores tiene una referencia al ambiente correspondiente. Es procesado por su agente correspondiente al salir del Heap del ambiente en el método HandleStatus.

Sobre el Lenguaje

El DSL se llama GoS y es un lenguaje de tipado dinámico desarrollado en C#.

Se implementó un *parser* LR(1).

Características

Se puede declarar variables:

```
let a = 5
```

Emplear expresiones matemáticas:

```
let a = 3.5
let b = (5+a)*5 + 1.52*100
```

Se puede declarar funciones:

```
fun sum(a, b) {
      return a + b
}
```

Statements en varias líneas:

```
let d = a \
      + b \
      - c
```

Crear servidores de tipo *worker* simple y distribuidores de carga:

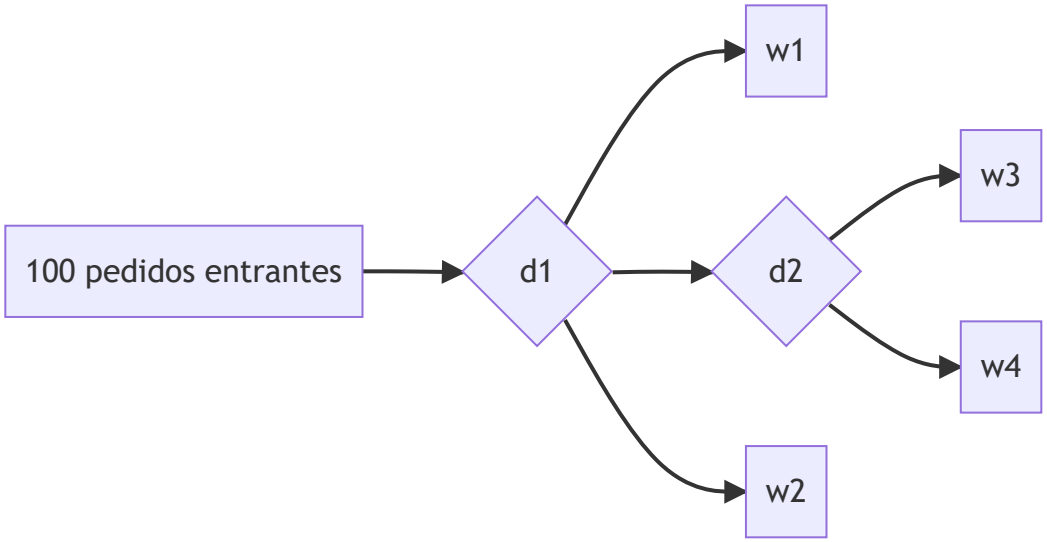
```
let w1 = simplew
let w2 = simplew
let d = distw
```

Conectarlos:

```
let w1 = simplew
let w2 = simplew
let d1 = distw
let w3 = simplew
let w4 = simplew
let d2 = distw

d1 → w1, d2, w2
d2 → w3, w4
```

Al primer distribuidor de carga declarado se le harán 100 pedidos, resultando el grafo de flujo de la siguiente manera:



Gramática de GoS

```
<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>
        | ID <right-conn>

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

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

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

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

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

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

```
        | "simplew"
        | "distw"

<right-conn> := "->" <id-list>

<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.

Nótese que bajo esta gramática no se soporta el llamado a funciones en la condición del `if`:

```
if f(arg) {
      # ...
}
```

Esto el equipo lo tiene en cuenta y será rectificado en entregas posteriores.

Tipos

El lenguaje tiene 3 tipos:

- `Number`: para todo tipo de números

- `Bool`: para valores de verdad (*true* o *false*)
- `Server`: para los servidores

Los operadores `+`, `-`, `*`, `/` solo están permitidos para el tipo `Number`.

En el caso del operador de conexión de servidores (`->`), solo está permitido para el tipo `Server`, y el operando de la izquierda debe ser un distribuidor de carga (lo que se espera es que pueda ser de cualquier tipo de servidor; será rectificado en una entrega posterior).

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.
- En el cuerpo de un *statement* de bloque o una función, los nombres de las variables ocultan los nombres de variables en ámbitos superiores.

Gramática de REGEX

El lenguaje de REGEX utilizado es el definido en la correspondiente conferencia de la asignatura.

Para el procesamiento de expresiones regulares se empleó la siguiente gramática:

```
<regex> := <union>

<union> := <union> "|" <concat>
         | <concat>
```

```
<concat> := <concat> <basic>
          | <basic>

<basic> := <atom> "*"
         | <atom> "+"
         | <atom> "?"
         | <atom>

<atom> := <group>
        | CHAR
        | <set>

<group> := "(" <regex> ")"

<set> := "[" <item-list> "]"

<item-list> := <item>
             | <item> <item-list>

<item> := <range>
        | CHAR

<range> := CHAR "-" CHAR
```

`CHAR` es un tipo de *token* definido especialmente para esta gramática. Representa a un caracter del lenguaje que no es *meta*. Por ejemplo, dada la REGEX `a*\+`, el *lexer* de REGEX devuelve los siguientes *tokens*

Tipo	Lexema
CHAR	a
*	*
CHAR	+

Nótese que como el `+` está precedido por un `\`, entonces el *token* producido es de tipo `CHAR`, en lugar de tipo `+`.

DSL para las Gramáticas

El equipo se enfrentó a la tarea de expresar estas gramáticas en C# mediante el empleo de un DSL embebido, desarrollado por el propio equipo 💪😎. Se cree que fue una buena oportunidad para aplicar las enseñanzas de las asignaturas Compilación y LP. Para lograrlo, se utilizó principalmente la sobrescritura de operadores.

La gramática de REGEX queda expresada entonces de esta manera:

```
Regex > Union,

Union > (Union | Concat),
Union > Concat,

Concat > (Concat, Basic),
Concat > Basic,

Basic > (Atom, times),
Basic > (Atom, plus),
Basic > (Atom, quest),
Basic > Atom,

Atom > Group,
Atom > @char,
Atom > Set,

Group > (lpar, Regex, rpar),

Set > (lbrak, ItemList, rbrak),

ItemList > Item,
ItemList > (Item, ItemList),

Item > Range,
Item > @char,

Range > @char - @char
```

Syntax Highlight **en el Code**

Es posible lograr *syntax highlight* de GoS en Visual Studio Code mediante la instalación de la extensión GoS, desarrollada por el equipo.

Ejecutando gos

Ahora el ejecutable del proyecto interpreta un archivo con código en GoS y ejecuta la simulación que se configura. La salida del programa es una serie de líneas con el tiempo y cuerpo de cada respuesta a los pedidos. Si el pedido no pudo ser procesado, el cuerpo es "Servidor no disponible", mientras que si fue procesado por el servidor w_1, entonces el cuerpo es "Cosas de servidor simple w1".

En la entrega se encuentra un archivo `distrb.gos` en la carpeta `Sources/`. Este se puede ejecutar de la siguiente manera:

```
$ gos Sources/distrb.gos
```