2020

Netcoreconf

¡Medir el consumo de recursos de programas .Net de forma fiable es posible!

José Manuel Redondo López

Profesor Contratado Doctor (Universidad de Oviedo)

@The_Rounded_Man

Sponsors

Microsoft
avanade
plain concepts
devs dna
everis
an NTT DATA Company
encamina
PIENSA EN COLORES
KABEL
SolidQ
intelequia
東京 TOKIOTA
DXC.technology
VISEO
DIGITAL MAKERS
dinahosting
ilitia
raona

El problema de medir rendimientos

- ¿Cuántas variables afectan al rendimiento de un software?
  - **El propio software**
    - Las tecnologías o APIs usadas para completar cada tarea
    - La “calidad” de la implementación
  - **Factores externos**
    - El rendimiento del sistema de almacenamiento
    - El rendimiento del sistema de memoria
    - El rendimiento de la CPU (cores, threads, frecuencia, IPC, instrucciones especializadas…)
    - Otros procesos con los que “convive” y que consumen recursos
- Todo ello hace que medir (de forma fiable) el rendimiento de un programa **sea una tarea no trivial**

Ejemplo práctico: rotura de hash

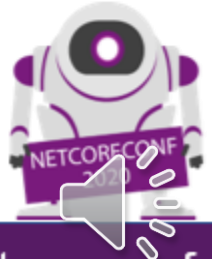
- Pongamos un ejemplo de todo ello con un ejemplo práctico “realista”: romper una hash de una clave
  - Las hash son secuencias alfanuméricas que se obtienen a partir de un texto
  - Existen varios algoritmos para generar hash, nosotros vamos a usar SHA256, incluido en la librería `Cryptography` de .Net Core
  - Dado un algoritmo de hashing, un mismo texto siempre genera la misma hash
    - **Pero es “imposible” obtener el texto original a partir de su hash**
  - Es parte de las técnicas para guardar passwords de forma segura en aplicaciones reales

Ejemplo práctico: rotura de hash

- Si esto es así, ¿cómo se obtienen passwords a partir de hash robadas habitualmente? ¡Por fuerza bruta!
  - Se obtiene un fichero de hashes de passwords que queremos averiguar
  - Se obtienen ficheros con passwords comunes (localizables fácilmente por internet / distribuidos habitualmente con herramientas típicas como Hydra, Jhon o Hashcat)
    - Uno famoso es llamado `rockyou.txt` (+14 millones de passwords)
  - Se elige la primera hash a "descifrar"
  - Se hashea cada password del fichero de claves comunes con el mismo algoritmo usado con las que queremos averiguar (y, si se usa, con sus bits de salt)
    - ¿Coincide? Ya sabemos la clave original y finalizamos
  - ¿No coincide ninguna? Mala suerte, habrá que probar con mas ficheros de claves comunes o generar aleatoriamente posibles claves (¡fuerza bruta pura!)

Ejemplo práctico: rotura de hash

- Obviamente, es un proceso muy costoso computacionalmente
  - Donde todas las variables que afectan al rendimiento que hemos mencionado entran en juego
  - Y que vamos a usar como ejemplo ilustrativo de lo que queremos explicar
- **El código de todos los ejemplos está disponible** en la dirección que se especificará al final de esta presentación
- Todos los tiempos que se muestren están tomados en estas condiciones
  - .Net Core 3.1
  - Visual Studio 2019 actualizado con una solución generada en modo Release
  - Ryzen 1700 con 64Gb de DDR4 2400Mhz y un Samsung SSD 850 EVO 500Gb



Implementación con arrays

- Vamos a ver un ejemplo de esto implementado con arrays
  - Lectura del fichero `rockyou.txt`
  - Prueba de una hash a descifrar con las claves leídas
- Obtenemos tiempos de sucesivas ejecuciones con los mismos datos
  - **3034ms, 3111ms, 3362ms…**

```
public static string[] ReadPasswordsAsArray(string fileName) {
    string line;
    var list = new List<string>();
    // Read the file and add it line by line.
    System.IO.StreamReader file =
        new System.IO.StreamReader(fileName);
    while ((line = file.ReadLine()) != null)
        list.Add(line);

    file.Close();
    return list.ToArray();
}
```

```
internal static string RevertSha256HashArray(string foundHash,
    string[] pwdArray) {
    foreach (var pwd :string in pwdArray) {
        if (foundHash.Equals(Sha256Hash(pwd)))
            return pwd;
    }
    return null;
}
```

Implementación con Linq

- Cambiamos ahora la implementación por el uso de `IEnumerable`, generadores y **Linq**
- Tiempos: **876ms**, **810ms**, **811ms**…
- ¡El resultado es mucho mejor que antes! (unas 3 veces más rápido)
  - No hemos cambiado ni el hardware ni los datos
  - El algoritmo, APIs, tecnología…tienen un impacto muy crítico en el rendimiento del programa

```
public static IEnumerable<string> ReadPasswordsAsIEnumerable
    (string fileName) {
    string line;
    var list = new List<string>();
    // Read the file and serve lines using a generator
    System.IO.StreamReader file =
        new System.IO.StreamReader(fileName);
    while ((line = file.ReadLine()) != null)
        yield return line;
    file.Close();
}
```

```
internal static string RevertSha256HashIEnumerable
    (string foundHash, IEnumerable<string> pwds) {
    return pwds.FirstOrDefault(
        predicate: pwd :string => foundHash.Equals(Sha256Hash(pwd)));
}
```

Implementación con PLinq

- Dado que el Ryzen 1700 tiene 8 cores y 16 threads, PLinq podría sacarle partido a sus capacidades para manejar muchos threads nativos, ¿no?
  - Para ello modificamos la función anterior de la forma indicada en la imagen de abajo
- ¡Pues **NO!**. Los tiempos son **10715ms, 11059ms, 11053ms**…
- PLinq particiona el `IEnumerable` para recorrerlo en paralelo con un nº de hilos óptimo de acuerdo a la máquina
  - Pero eso obliga a recorrerlo casi completamente
  - Cuando se encuentra la password en una de las particiones, las otras aún tienen que terminar
  - No se están parando todos los hilos cuando uno encuentra el resultado

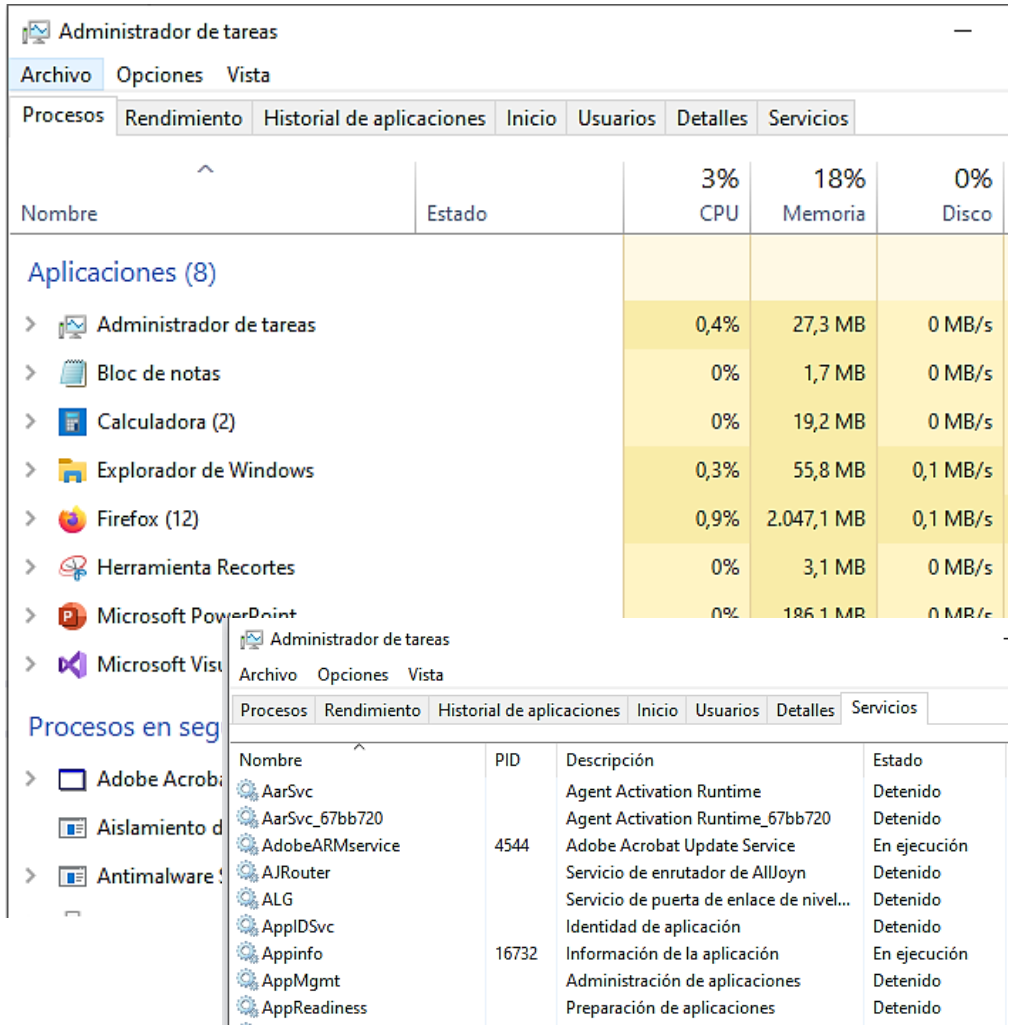
```
internal static string RevertSha256HashIEnumerablePLinq
    (string foundHash, IEnumerable<string> pwds) {
    return pwds.AsParallel().FirstOrDefault(
        predicate: pwd :string => foundHash.Equals(Sha256Hash(pwd)));
}
```

Primeras conclusiones

- Los algoritmos usados, APIs, código, técnicas, etc. son de importancia capital
  - Hacer pruebas previas de cómo vamos a implementar ciertas cosas es también muy importante, porque no siempre la mejor solución es la que parece
  - Podría parecer que PLinq era la mejor vía, ¡pero resultó ser mucho más lenta en este escenario concreto!
- Pero, **¿hemos medido bien los tiempos de ejecución?**
  - Obviamente, está claro qué técnica es mejor de las tres
  - Pero ¿no habéis notado que los tiempos devueltos son bastante inconsistentes?
  - Para hacernos una idea valen pero, **¿Qué pasa si tenemos que dar datos fiables, precisos y que puedan justificarse adecuadamente?**

Dar un dato de rendimiento fiable no es fácil

- Factores que afectan a este algoritmo
  - Velocidad del almacenamiento secundario
  - Velocidad de la RAM
  - Rendimiento de la CPU (cores/threads, frecuencia)
  - Y, sobre todo, **que siempre "convivimos" con procesos y servicios** ejecutándose en paralelo
    - Incluido el **recolector de basura**



Los SO son inherentemente multiproceso

- Nuestro programa puede estar todo lo optimizado posible pero *"he will never walk alone"* ☺
- Existirán multitud de procesos y servicios en ejecución en paralelo
  - Cada uno se lleva una pequeña porción de la CPU
- A veces, algunos de ellos necesitarán hacer una tarea pesada mientras nuestro programa está en ejecución
  - Esto nos va a afectar al rendimiento queramos o no, e introduce variabilidad en los tiempos que obtenemos
- Es necesario algo que mitigue este problema y nos permita dar un dato de tiempo fiable

¿Cómo medir entonces?

- Esta claro que tomar tiempos una sola vez es una mala idea si queremos dar un dato preciso
  - Es necesario entonces **medir varias veces**
- Pero… ¿cuántas? ¿2? ¿3? ¿10? ¿100?
  - El nº no puede ser arbitrario, ya que pueden ser muy pocas…o demasiadas
  - Tampoco es una buena forma de justificar tus resultados ☺
- Necesitamos un método estadísticamente riguroso que obtenga un dato preciso que podamos “defender” ante quien lo cuestione

¡Vamos a ello!

Midiendo “bien” ☺

Como ser estadísticamente riguroso cuando se miden rendimientos

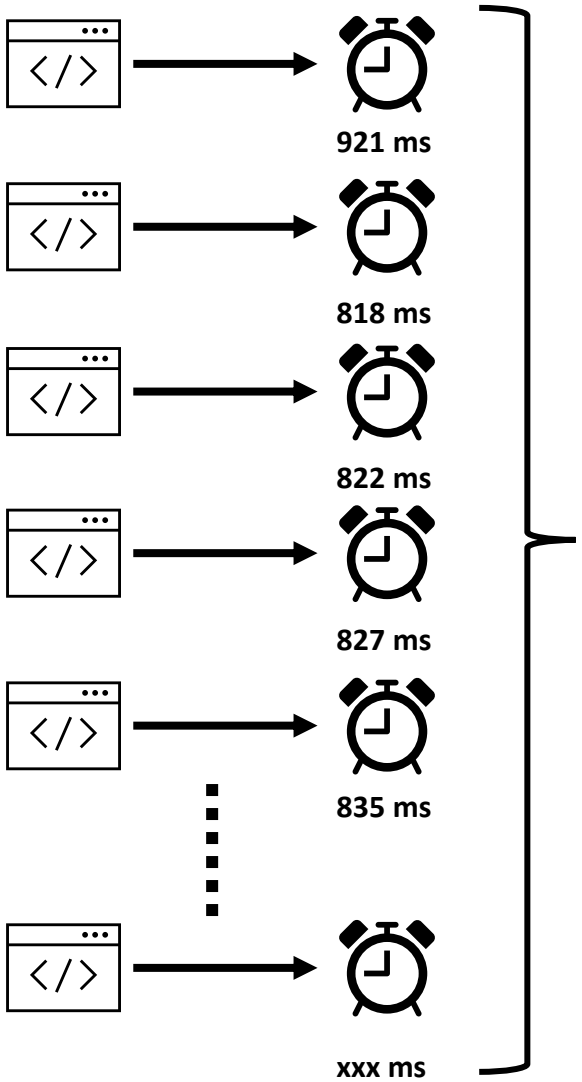
¡Este método no lo he inventado yo!

- El método que vamos a presentar está basado en un artículo de investigación muy citado
  - *Andy Georges, Dries Buytaert, Lieven Eeckhout. "Statistically rigorous Java performance evaluation". OOPSLA '07* (ver referencia completa al final)
- Plantea un método de medición de rendimientos **estadísticamente riguroso**, que obtiene datos precisos
  - Aporta además intervalos de confianza e información adicional para dar algo más justificable que un simple dato obtenido de cualquier forma
- Hemos usado este método con éxito en los artículos de nuestro grupo de investigación (*Computational Reflection* de la Universidad de Oviedo)
- **Yo sólo os voy a describir una implementación del mismo para .Net Core**

¿En qué se basa esta técnica?

- Como hemos dicho anteriormente, para hacerlo bien hay que medir varias veces hasta que los tiempos se puedan considerar “fiables”
- ¿Cómo se sabe si son fiables?: con el **intervalo de confianza**
  - Números entre los cuales se estima que estará cierto valor desconocido con un determinado nivel de confianza
- Usamos un intervalo de confianza del 95% calculado con la distribución T de Student
  - Esos números son un tiempo mínimo (`TMin`) y máximo (`TMax`) que estimamos que tarda el programa medido
  - Así decimos que tenemos un 95% de confianza en que el tiempo que tarda ese programa está dentro del intervalo que damos (`TMin, TMax`)
  - Como valor de tiempo final, usamos **el promedio de este intervalo**

Esquema: Funcionamiento básico

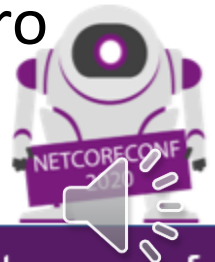


(TMin , TMax , Promedio)
(837.17 ms, 852.03 ms, 844.6 ms)

"El tiempo que tarda en ejecutarse este programa se encuentra entre 837.17ms y 852.03ms con una confianza del 95%. Como valor representativo para hacer cálculos, te doy su promedio 844.6 ms"

¿Y eso es todo?

- Me temo que no ☺
- Este método de medición tiene dos “perfiles” de uso, en función de lo que vayamos a medir
  - **Startup**: pensado para medir aplicaciones de ejecución corta, incluyendo su posible compilación JIT durante la ejecución
  - **Steady-state**: pensado para medir aplicaciones que se ejecutan durante mucho tiempo
    - El tiempo usado en la compilación JIT es despreciable frente a lo que tarda la aplicación en sí
    - Al ejecutarse durante mucho tiempo, el sistema puede implementar optimizaciones dinámicas al código en ejecución
- En ambos casos se sigue la filosofía de “medir hasta que quede fiable”, pero con variantes



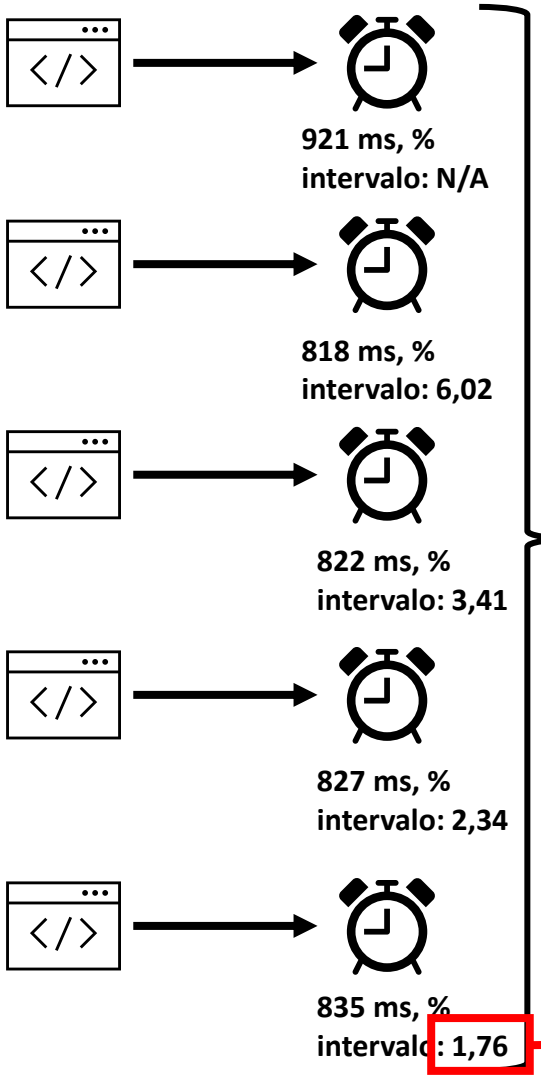
Startup

- “Técnicamente” se miden 30 iteraciones del programa (`pIterations`)
- Para cada iteración
  - Se toman tiempos y se añaden a una lista (`executionTimes`)
  - Se calcula el intervalo de confianza al 95% (`confidenceLevel`) de los tiempos tomados hasta el momento en todas las iteraciones anteriores y la actual
- Al acabar las iteraciones, se devuelve una tupla de información estadística útil como resultado
  - `(TMin, TMax, Promedio, Desviación típica, Tamaño % intervalo)`
  - Tamaño % del intervalo mide el tamaño del intervalo de tiempos obtenido, pero expresado como un % respecto a su promedio: `(TMax-TMin/Promedio)*100`)
  - Esto lo usamos también para **optimizar el proceso de medida**

¿Optimizar el proceso de medida?

- Con esto no nos referimos a medir mejor, **sino a medir más rápido** sin perder el “espíritu” del procedimiento de medida
- ¿De verdad es necesario medir 30 veces si vemos que un programa concreto devuelve tiempos muy estables tras N iteraciones?
  - En nuestro caso determinamos que no, por lo que usamos el dato de tamaño % del intervalo para “cortar” la medición antes de 30 iteraciones
  - En el ejemplo siguiente se termina cuando dicho porcentaje **cae por debajo del 2%** (5 iteraciones)
  - Es un parámetro configurable, y todo depende de cuánta “precisión” necesitemos…

Esquema: Startup



(TMin , TMax , Promedio, DesvEst , %)

(837.17 ms, 852.03 ms, 844.6 ms, 43.17 ms, 1.76 %)

"El tiempo que tarda en ejecutarse este programa midiéndolo en Startup se encuentra entre 837.17 ms y 852.03 ms con una confianza del 95%. Como valor representativo para hacer cálculos, te doy su promedio 844.6 ms, con una desviación típica de 43.17 ms y un tamaño del intervalo respecto a la media de 1.76%"

¡% intervalo < 2%! -> paramos la medición y no llegamos a 30 iteraciones

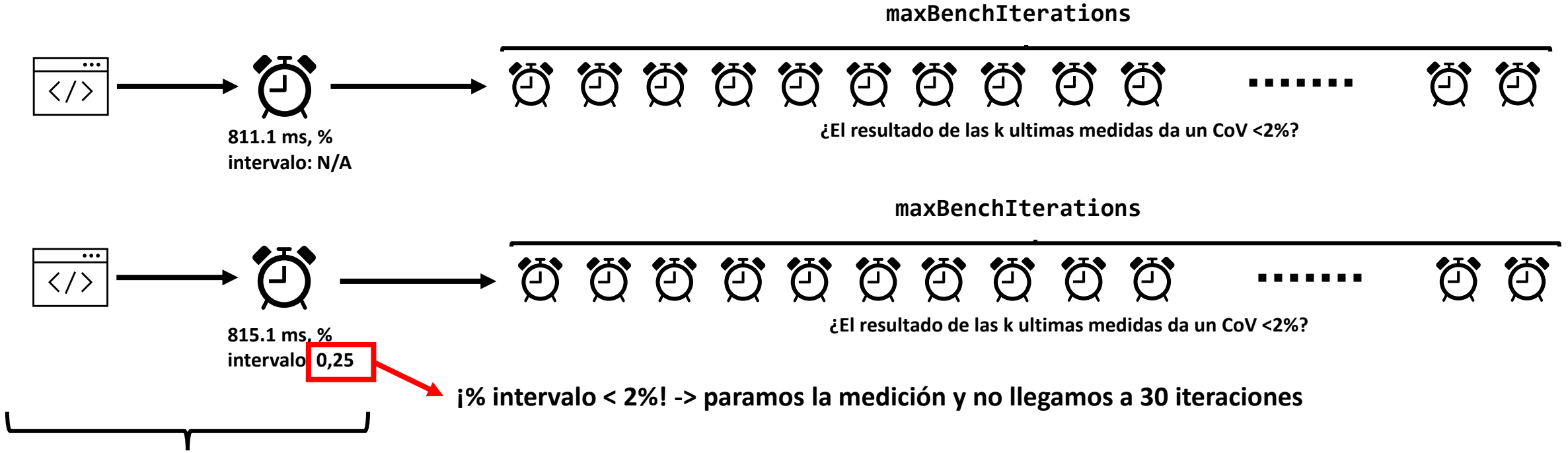
Steady-state

- Ejecutamos un máximo de 30 iteraciones (`pIterations`) de nuevo
- Para cada iteración
  - Ejecutamos el programa un máximo de 30 veces (`maxBenchIterations`) y un mínimo de 10 veces (`k`) (método `RunAsSteady`)
    - Cuántas se hagan realmente depende de si las k últimas medidas tomadas en esa iteración tienen un coeficiente de variación (CoV) inferior al 2%
      - El CoV se calcula como `(Desviación típica / media) * 100`
    - Si se da este caso se dice que el programa ha alcanzado su **steady-state**
  - Cada ejecución de `RunAsSteady` **devuelve el promedio de las últimas k mediciones**
  - Se calcula el intervalo de confianza al 95% (`confidenceLevel`) de esos tiempos promedio tomados hasta el momento en todas las iteraciones, como en Startup
- El resultado final se calcula (y optimiza) **también igual que en Startup**

Steady-State

- Debe tenerse en cuenta que el Steady-state mide un programa despreciando el tiempo empleado en cosas como la compilación JIT, optimizaciones dinámicas, etc. que no son puramente rendimiento del código del programa
- No es de extrañar por tanto que nos encontremos con datos de rendimiento que
  - Son algo mejores que los de Startup (optimizaciones)
  - Ofrecen menos desviación típica (al alcanzar el Steady-state)
- No obstante, todo depende del programa ejecutado y otras variables, por lo que no debe tomarse como una norma

Esquema: Steady-State



(TMin , TMax , Promedio, DesvEst, %)

(812.08 ms, 814.17 ms, 813.1 ms, 2.82 ms, 0.25%)

"El tiempo que tarda en ejecutarse este programa midiéndolo en Steady-state se encuentra entre 812.08 ms y 814.17 ms con una confianza del 95%. Como valor representativo para hacer cálculos, te doy su promedio 813.1 ms, con una desviación típica de 2.82 ms y un tamaño del intervalo respecto a la media de 0.25%"

Ventajas de este método

- **¡Estadísticamente riguroso!** Se dan intervalos, niveles de confianza, promedios, desviaciones típicas…
- Tiempos mucho más fiables que ejecuciones sueltas
- Devuelven medidas que pueden justificarse ante cualquier necesidad, dada su sólida base matemática
- Permite identificar ganancias de rendimiento pequeñas debido a optimizaciones de manera mucho más fiable
- Elimina el factor “a ojímetro”, y le da un aspecto más profesional a nuestro trabajo ☺

Desventajas de este método

- **Tarda bastante más en completar las mediciones**, al estar basado en ejecutar múltiples iteraciones del código a medir
  - ¡Especialmente en Steady-state!
  - Para paliar ese efecto en parte se han introducido las optimizaciones descritas
- Se debe tener mucho **cuidado con algoritmos con efectos laterales**
  - Si un algoritmo hace asignaciones destructivas, manipula BBDD, ficheros, etc., la ejecución de múltiples iteraciones podría destruir o corromper datos
  - Se debe tener especial cuidado y preparar los datos a medir cuidadosamente antes de lanzar las mediciones
  - Es decir, considerar si hay que hacer **un paso de "limpieza" previo a cada ejecución**

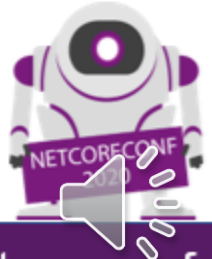
Resultados

- Aún con este método, los resultados aún más fiables se obtendrían cerrando todas las aplicaciones posibles y dejando el sistema "estabilizarse"
- Dicho de otro modo, no hacerlo inmediatamente tras el arranque, actualizando, pasando un escaneo…
  - ¡Mejor eliminar todas las interferencias posibles!

Algoritmo	Ejecuciones sueltas	Startup (Promedio, % intervalo)	Steady (Promedio, % intervalo)
Arrays	3034 ms, 3111 ms, 3362 ms…	2995 ms (0,3%)	3357.05 ms (0,34%)
IEnumerable + Linq	876 ms, 810 ms, 811 ms…	844.6 ms (1,76%)	813.1 ms (0,25%)
IEnumerable + PLinq	10715 ms, 11059 ms, 11053 ms…	9879 ms (1,6%)	9798.65 ms (1.2%)

¿Y con la memoria?

- Esta metodología también se puede usar para medir la memoria consumida por un proceso
- Usando la propiedad `PeakWorkingSet64` de `System.Diagnostics.Process`
  - Esta propiedad indica el tamaño máximo del **working set memory** empleado por el proceso desde que empezó
  - El **working set** de un proceso es el conjunto de páginas de memoria que un proceso tiene visible en la RAM física
    - Son las páginas disponibles y residentes en RAM para una aplicación (no generan fallos de página)
    - Incluyen tanto datos compartidos como privados del proceso
    - Los datos compartidos incluyen las todas las instrucciones a ejecutar por el proceso (también las de los módulos o librerías del sistema que use)



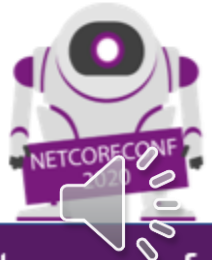
¿Y con la memoria?

- Por lo general estas mediciones son mucho más estables que las de rendimiento
- El valor de `PeakWorkingSet64` rara vez cambia durante la ejecución de un proceso, al ser un valor máximo
  - ¡Eso quiere decir que mejor medimos exactamente lo que queremos y no otras cosas antes!
- Por tanto, en este caso tiene sentido hacer una sola medición o (por asegurar) usar la metodología **Startup**
  - En la implementación de ejemplo se usa **Startup**, siguiendo el mismo algoritmo descrito

Algoritmo	Memoria (Kb)
Arrays	1.015.016
IEnumerable + Linq	22.852
IEnumerable + PLinq	27.432

¿Y con la memoria?

- Vemos que las implementaciones que usan Linq son **MUCHO** más eficientes en el uso de memoria que cargar un array
  - Un generador es aproximadamente 44 veces más eficiente en el uso de memoria que usar un array tradicional en este escenario
  - Y además devuelve mucho mejor rendimiento (siempre que usemos PLinq ☺)
- Queda pues demostrado que saber usar las APIs y tecnologías adecuadas puede marcar una diferencia ENORME
  - **¡Pero esta vez lo demostramos de manera estadísticamente rigurosa!** ☺



Implementación en .Net Core

Organización y significado del código que acompaña a esta presentación

API

- Todo el código presentado aquí está disponible en: https://github.com/jose-r-lopez/NetCoreConf2020
  - Los nombres de métodos y parámetros de los algoritmos de medida son exactamente los mismos que hemos descrito en las transparencias anteriores para facilitar su localización
  - Entre cada iteración de los algoritmos **se fuerza una recolección de basura completa** de forma síncrona para eliminar otro posible factor de variación
- La solución está dividida en dos proyectos
  - **`TestMeasurements`**: Los algoritmos medidos (implementación de descifrados de hash)
  - **`PerformanceMeasurementLibrary`**: La implementación de los métodos de medición Startup y Steady-state descritos, con todas sus funciones auxiliares

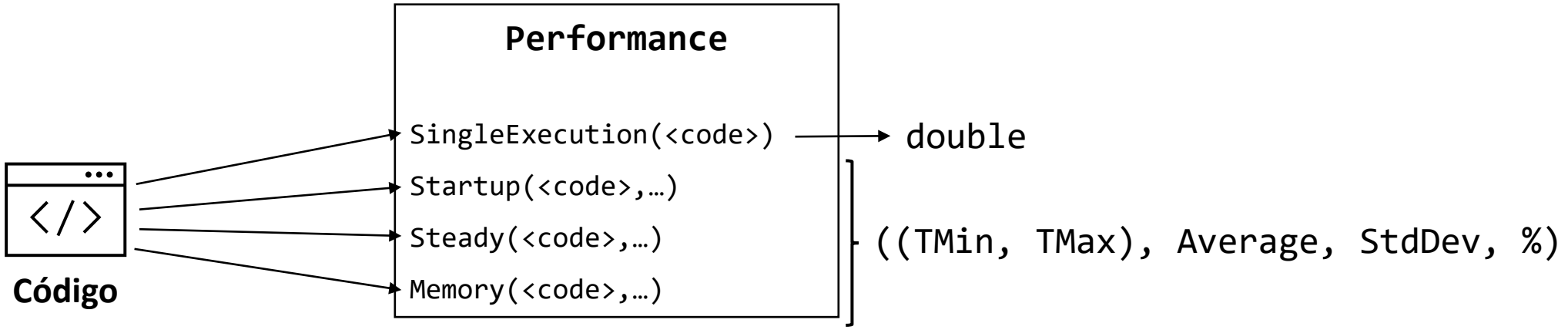
TestMeasurements

- Consta de los siguientes elementos
  - Carpeta `AuxiliaryFunctions`
    - Fichero `FileFunctions.cs`: Funciones para leer las passwords en arrays o `IEnumerable`
    - Fichero `HashFunctions.cs`: Funciones para crear / "descifrar" hash SHA256
  - Carpeta `data`: el fichero de passwords `rockyou.txt` de los ejemplos
  - Carpeta `Tests`
    - Fichero `ArrayTests.cs`: Implementación de los test con arrays (Startup, Steady, Memory, Single execution)
    - Fichero `IEnumerableTests.cs`: Implementación de los test con `IEnumerable` y Linq (Startup, Steady, Memory, Single execution)
    - Fichero `PLinqTests.cs`: Implementación de los test con `IEnumerable` y PLinq (Startup, Steady, Memory, Single execution)
  - `Program.cs`: El programa principal que lanza las medidas

PerformanceMeasurementLibrary

- El principal fichero de la librería es `Performance.cs`, que contiene el front-end de la misma (clase `Performance`)
- Dicha clase contiene un método para cada modo de medición visto: Single Execution, Startup, Steady, Memory)
- Todos esos métodos aceptan un **delegado tipo `Action`** como código a medir
  - Usaremos este delegado para pasar cualquier código que queramos
  - Si el código tiene parámetros, lo pondremos en forma de cláusula o lo invocaremos desde el propio delegado con parámetros predeterminados

PerformanceMeasurementLibrary




```
//In a real-life example, this should
//be an unknown hash you want to shake it off ;)
string foundHash = HashFunctions.Sha256Hash( pwd: "taylor swift");

var result :((intervalLow,intervalHigh),mean,…) = Performance.Startup( command: () =>
{
    var pwdArray :IEnumerable<string> =
        FileFunctions.ReadPasswordsAsIEnumerable( filePath: @"..\..\..\data\rockyou.txt");
    string pwd = HashFunctions.RevertSha256HashIEnumerable(foundHash, pwdArray);
});
```

PerformanceMeasurementLibrary

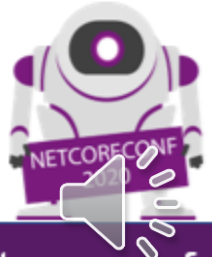
- Todos los métodos (salvo `SingleExecution`) aceptan los parámetros vistos en su descripción (con el mismo nombre mencionado)
  - `confidenceLevel`, `pIterations`, `k`, `CoV`…
  - **Cada uno tiene un valor por defecto**, establecido al que se mencionó en las descripciones anteriores
    - Esto permite simplemente llamar a librería con el código a medir y listo ☺
  - Un parámetro `verbose` permite mostrar mensajes adicionales por pantalla sobre la ejecución de cada función (para debug)
- Todos (salvo `SingleExecution`, que directamente devuelve un valor) devuelven una tupla de valores `double` como la descrita
  - `(TMin, TMax, Promedio, DesvEst, % tamaño del intervalo)`

PerformanceMeasurementLibrary

- La librería tiene además una serie de ficheros auxiliares
  - `SteadyFunctions.cs`: Funciones auxiliares para poder hacer la medición en Steady-State
    - La función que determina si las k mediciones tienen un CoV por debajo del especificado
    - La función `RunAsSteady` mencionada que ejecuta un programa N veces en cada iteración
  - `Statistics.cs`: Un fichero de funciones estadísticas simples auxiliares (media y desviación típica de un conjunto de valores)
  - `TStudent.cs`: Una implementación de la distribución T de Student basada en los algoritmos de ACM

Referencias / Enlaces

- **El artículo en el que está basada toda esta presentación:** *Andy Georges, Dries Buytaert, Lieven Eeckhout. "Statistically rigorous java performance evaluation". OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications. pp 57–76. DOI:* https://doi.org/10.1145/1297027.1297033
- **Nuestro grupo de investigación** Computational Reflection: http://www.reflection.uniovi.es/
- **Artículos más citados donde hemos usado esta librería** (su implementación Python)
  - *Francisco Ortín, Miguel A. Labrador, Jose M. Redondo. "A hybrid class- and prototype-based object model to support language-neutral structural intercession". Information and Software Technology 56 (2014) 199–219*
  - Jose Manuel Redondo, Francisco Ortin. *"A Comprehensive Evaluation of Common Python Implementations"*. IEEE Software 32 (4) (2014), 76-84



Sponsors

Microsoft
avanade
plain concepts
devs dna
everis
an NTT DATA Company
encamina
PIENSA EN COLORES
KABEL
SolidQ
intelequia
東京 TOKIOTA
DXC.technology
VISEO
DIGITAL MAKERS
dinahosting
ilitia
raona

Más información:

info@netcoreconf.com
@Netcoreconf

Visítanos en:
netcoreconf.com