UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE UN FRAMEWORK PARA LA PROGRAMACIÓN DE COMPONENTES AUTO-ADAPTABLES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS NICOLÁS IBÁÑEZ POZO

PROFESOR GUÍA:
JAVIER BUSTOS JIMÉNEZ

MIEMBROS DE LA COMISIÓN:
MARIA CECILIA RIVARA ZUÑIGA
RODRIGO ARENAS ANDRADE

SANTIAGO DE CHILE
2015

Resumen

La *Service Oriented Arquitecture* (SOA) ha sido introducida para fomentar una interacción dinámica y de bajo acoplamiento entre servicios ofrecidos por diferentes proveedores, permitiendo el desarrollo de sistemas distribuidos altamente escalables. Para abordar la complejidad de este tipo de aplicaciones se ha propuesto la *Service Componente Arquitecture* (SCA), un conjunto de especificaciones tecnológicamente agnósticas que combina la programación basada en componentes con la orientación a servicios. Sin embargo, la SCA no considera modificaciones en la aplicación durante el tiempo de ejecución y, por lo tanto, las tareas de monitoreo y administración deben ser manejadas por la plataforma que implementa la SCA.

Ante esta problemática se propuso un *framework* de monitoreo y reconfiguración inspirado en la *computación autonómica*, iniciativa que promueve sistemas capaces de administrarse a si mismos dados algunos objetivos de alto nivel. De esta manera, se hizo posible diseñar aplicaciones SCA basadas en componentes auto-adaptables; componentes cuyo comportamiento puede ser programado para adaptarse a diferentes requisitos de administración.

Actualmente existe una implementación de referencia de este *framework*, sin embargo, esta implementación esta inconclusa y carece de una API que facilite su utilización en la practica. Por lo tanto, en este trabajo de memoria se retoma dicha iniciativa para completar la **implementación de un *framework* para la programación del comportamiento auto-adaptable de componentes**. Adicionalmente, en esta implementación se propone una **API simple para la definición y modificación del comportamiento auto-adaptable de componentes**, API que permitirá modificar el comportamiento auto-adaptable en tiempo de ejecución.

Para esto, se realiza un análisis de la propuesta original de este *framework* y se definen formalmente los objetivos de alto nivel que determinarán el comportamiento auto-adaptable del componente. Luego, se terminan de implementar los elementos que hacen posible las reconfiguraciones autonómicas y se integran con las herramientas de reconfiguración propias de la plataforma SCA sobre la cual se basa esta implementación.

Finalmente, se muestra la efectividad y capacidades de este *framework* a través la API propuesta mediante la experimentación con una aplicación SCA real. Para esto, se implementa un crackeador de contraseñas distribuido y se muestra como utilizar esta API para proveer un comportamiento auto-adaptable en dos sentidos; en la capacidad de reconfigurarse autonómicamente para distribuir su trabajo en las proporciones óptimas cada vez que cambien las condiciones del ambiente distribuido, y en la capacidad de modificar su arquitectura autonómicamente para cumplir con la calidad de servicio esperada.

Esta memoria fue parcialmente financiada por el Equipo Asociado SCADA, entre Universidad de Chile e INRIA Sophia-Antipolis, Francia.

Tabla de contenido

Introducción

Los sistemas de software han demostrado ser un artefacto tremendamente exitoso. Su uso se ha extendido desde un ámbito estrictamente profesional hacia una gran parte de nuestras actividades cotidianas. Por otro lado, las innovaciones tecnológicas de estos últimos años ha llevado a estos sistemas a evolucionar desde una estructura estable, monolítica y centralizada hacia una dinámica, distribuida y descentralizada. Esta evolución plantea nuevos desafíos a la forma en como se diseñan y mantienen estos sistemas de software.

Para abordar la complejidad introducida por la computación distribuida se ha acogido el paradigma de diseño *orientación a servicio*. Un servicio de software se define como un mecanismo que permite acceder a una o más capacidades, donde el acceso es provisto utilizando una interfaz de forma consistente con las restricciones y políticas especificadas por la descripción del servicio [1]. Por lo tanto, en este paradigma los servicios son tratados como elementos de primera clase; la estrategia consiste separar las funcionalidades que realizan una tarea específica de modo que puedan ser encapsuladas detrás den un único servicio y reutilizadas como parte de otras soluciones. Ahora, para organizar el diseño y construcción de aplicaciones orientadas a servicios se ha definido el *Service Oriented Arquitecture* (SOA) como una forma lógica de proveer servicios tanto a la aplicación de uso final como otros servicios distribuidos a través de la red [2]. Los principales objetivos del SOA consisten en soportar la interoperabilidad de servicios proveniente de diferentes proveedores y facilitar modificaciones que permitan al sistema evolucionar.

Por otro lado, el paradigma de la programación orientada a componentes aborda el problema de la separación de conceptos. En este paradigma los elementos de primera clase corresponden a componentes de software; cada componente encapsula su lógica interna, exponiendo sus capacidades a través interfaces definidas, y las aplicaciones son construidas mediante la composición de estos componentes. De la convergencia entre la programación orientada a componentes y la orientación a servicios nace el *Service Component Architecture* (SCA), donde se los servicios son implementados como componentes y la aplicación final se construye como la composición de un conjunto de servicios.

El enfoque orientado a componentes del SCA simplifica la adopción de servicios procedentes de un conjunto heterogéneo de proveedores y fomenta la reutilización, sin embargo, carece de soporte para una evolución dinámica. Asuntos relacionados con la reconfiguración y otros aspectos no funcionales no son cubiertos en la especificación de SCA y deben ser manejados en su implementación.

La falta de soporte para tareas reconfiguración no es un problema menor, la mayoría de estos sistemas necesitan evolucionar constantemente para mantener contentos a sus usuarios.

La evolución involucra actividades como agregar nuevas funcionalidad, renovar interfaces, aumentar la capacidad de carga o adaptarse a nuevas plataformas. Con el tiempo, estos cambios se van acumulando, aumentando el tamaño del sistema y su complejidad estructural, y en consecuencia haciendo las tareas de mantención más costosas. Mientras más grande sea el sistema, más difícil es su mantención; a medida que el tamaño del sistema crece, el numero de entidades que deben ser ensambladas aumenta exponencialmente, aumentando también la probabilidad de incrementar la complejidad estructural del sistema [3].

Las actividades de mantención consisten en realizar reconfiguraciones y/o pequeñas modificaciones para reparar *bugs*, actualizar las políticas de calidad del servicio, o para adaptarse a cambios en las condiciones del ambiente de ejecución. Comúnmente estas tareas son realizadas por un operador humano. Es común no considerar a este operador como una potencial fuente de fallas, sin embargo, en la practica la complejidad de estas tareas pueden superar las capacidades humanas del operador. Por ejemplo, una encuesta realizada el 2001 [4] sobre tres sitios web de tamaño mediano, a lo largo de 6 meses, muestra que el 51 % de las fallas fueron producidas por errores humanos.

Estas actividades de mantención se clasifican en cuatro categorías; *Correctiva* para reparar fallas y errores, *Preventiva* para reparar vulnerabilidades, *Adaptativa* para hacer frente a los cambios en el ambiente de ejecución y *Perfectiva* para manejar los cambios en la calidad de servicio deseada. Estudios muestran solo un 25 % de los esfuerzos en mantención son destinados a tareas correctivas y preventivas [5], mientras que el restante 75 % son dedicados a tareas ligadas a la adaptación del sistema; un 25 % a tareas adaptativas y un 50 % a perfectivas.

Como solución a este problema Cristian Ruz[1] propuso *framework* de monitoreo y administración de servicios para una aplicación SCA [6]. Este *framework* esta diseñado para ser adjuntado a los servicios y es capaz de hacerse cargo de la administración de la composición completa de una aplicación.

La solución propuesta está basada en los planteamientos de la *computación autonómica*, una iniciativa que promueve la idea de tener sistemas capaces de administrarse a si mismos dado algunos objetivos de alto nivel. Para lograr este comportamiento la computación autonómica propone la implementación de un *bucle de control autonómico*, donde se incluyen cuatro actividades básicas: observación, análisis, toma de decisiones y reacción. El *framework* propuesto adopta este bucle de control autonómico de tal forma que, definido los objetivos administrativos de alto nivel para un servicio; se observa el estado del servicio, se analizan las observaciones, se determinan las reconfiguraciones necesarias para cumplir con los objetivos administrativos y, finalmente, se aplican estas reconfiguraciones. De esta forma, se hizo posible diseñar aplicaciones SCA utilizando componentes capaces de administrarse por si mismos cuando sea requerido, es decir, *componentes auto-adaptables*.

Actualmente, este *framework* cuenta con una implementación de referencia[2] que esta inconclusa; no todas sus funcionalidades están soportadas, y no cuenta con una API bien definida que facilite su utilización en la practica.

[1] Cristian Ruz, *Docteur en Informatique*. Investigador del DCC, Pontificia Universidad Católica de Chile.

[2] Implementación escrita por C. Ruz como una extensión de la implementación de GCM/ProActive. Repositorio ubicado en *http://git.gitorious.ow2.org/ow2-proactive/programming.git*, bajo la rama *SHORT_ Component_ Monitoring*.

Es por esto que en la siguientes paginas se define una una API simple y de rápido acceso que permita aprovechar al máximo las capacidades de este *framework* para la programción de comportamiento auto-adaptable de componentes ACS. Con esta API se pretende reducir al mínimo posible la interacción de un operador humano durante las tareas de mantención de este tipo de aplicaciones, flexibilizando y simplificando al mismo tiempo la operabilidad de este *framework* durante las instancias estrictamente necesarias.

Para lograr esto, se procederá de la siguiente manera:

- *Definición de los objetivos de alto nivel*, estos objetivos son quienes determinan el comportamiento auto-adaptable del componente. Se espera poder resumir completamente las capacidades del *framework* a través de estos objetivos, limitando la interacción los componentes únicamente la inserción o remoción de estos objetivos.
- *Implementación del framework*, considerando como punto de partida la implementación inconclusa existente y basándose en los objetivos de alto nivel definidos anteriormente. Durante este proceso se realizarán pequeñas modificaciones al diseño original y se reimplementarán algunos elementos con el fin de mejorar su usabilidad y/o desempeño.
- *Integración con la implementación de la plataforma SCA*, se extenderán las herramienta de reconfiguración propias de la la plataforma SCA donde se construyó esta implementación de referencia, facilitando la manipulación de los objetivos de alto nivel.

Finalmente, se mostrará la efectividad y capacidades de este *framework* a través la API propuesta mediante la experimentación con una aplicación SCA real. Se implementará crackeador de contraseñas distribuido y se mostrará como utilizar esta API para proveer un comportamiento auto-adaptable en dos sentidos; en la capacidad de reconfigurarse autonómicamente para distribuir su trabajo en las proporciones óptimas cada vez que cambien las condiciones del ambiente distribuido, y en la capacidad de modificar su arquitectura autonómicamente para cumplir con la calidad de servicio esperada.

Capítulo 1

Contexto

En este capitulo se presentará el contexto en el que se desarrolló este trabajo mediante una breve introducción a cada una de las tecnologías involucradas.

1.1. GCM

El *Grid Component Model* (GCM) [7] es un modelo de componentes pensado para diseñar, implementar y ejecutar aplicaciones distribuidas basadas en componentes. Su nombre proviene de la *computación grid*, una forma de computación distribuida donde se admiten recursos de hardware de múltiples arquitecturas ubicadas a grandes distancias geográficas. Este modelo fue especificado por el Instituto en Modelos de Programación del proyecto CoreGRID Europa y estandarizado por el Instituto de Estandares de Telecomuniaciones (ETSI) [8].

1.1.1. El Modelo GCM

La especificación de GCM esta basada en el modelo de componentes de uso general Fractal [9], de quien hereda la estructura básica de los componentes, la capacidad para reconfigurar sus enlaces dinámicamente, y su modelo jerárquico de abstracción. GCM extiende Fractal para soportar un despliegue distribuido de componentes, mejorar el soporte de comunicaciones colectivas y para establecer una separación estricta entre los aspectos funcionales y no funcionales del componente.

En este modelo, un componente representa un servicio encapsulado capaz de soportar una o más interfaces. Según su jerarquía, un componente puede ser de tipo *primitivo*, componente básico; o de tipo *compuestos*, definidos a partir de otros subcomponentes. De este modo, todo un subsistema puede ser modularizado y expuesto como un único componente compuesto.

Estos componente se comunican entre sí a través de sus interfaces. Estas interfaces pueden ser de tipo *servidor*, encargada de recibir solicitudes entrantes; o de tipo *cliente*, utilizada para enviar solicitudes a otros componentes. Al mismo tiempo, estas interfaces son etiquetadas como *funcional* (F) o *no funcional* (NF). Las interfaces funcionales corresponden a aquellas utilizadas para cumplir con el objetivo principal propuesto para el componente, el resto

de las interfaces son declaradas no funcionales y normalmente están relacionadas con la configuración y administración del componente.

Una aplicación GCM se contruye mediante la composición de estos componentes, enlanzando sus interfaces entre sí; desde una interfaz de tipo *cliente* hacia una *servidor*, restringiendo este enlace a Fs con Fs y NFs con NFs como se puede observar en el ejemplo de la Figura 1.1:

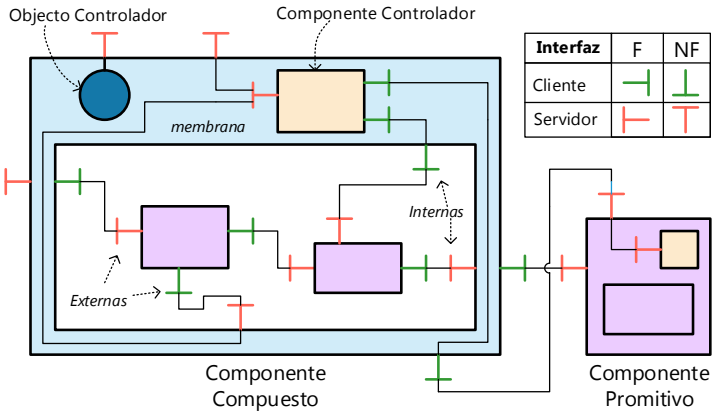


Figura 1.1: El modelo de componentes GCM.
F y NF son abreviaciones de *funcional* y *no funcional*, respectivamente.

La membrana

Todo componente posee un área especial, denominada *membrana*, donde se ubican los elementos relacionados con la configuración y administración del componente (aspectos no funcionales). Por defecto, todo componente posee un su membrana una conjunto de *objetos controladores* que proveen las operaciones básicas necesarias para la manipulación del componente, operaciones que son expuestas a través de un interfaces de tipo servidor no funcionales. Un ejemplo de estos controladores básicos se lista a continuación:

- *Attribute Controller*: Expone atributos internos configurables del componente por medio de *getters* y *setters*.
- *Binding Controller*: Interfaz que permite enlazar y desenlazar una interfaz tipo cliente del componente a una interfaz servidora externa.
- *Content Controller*: Intefaz que permite listar, agregar, o remover sub-componentes al componente, solo en el caso de tratarse de un componente compuesto.

Esta membrana se encuentra abierta para añadir nuevos controladores personalizados, esta vez en forma de *componentes controladores*. Estos componentes controladores pueden hacer uso de las interfaces no funcionales del componente para enlazarse con los componentes controladores de otros componentes.

1.1.2. GCM/ProActive

GCM/ProActive es la implementación de referencia de GCM, desarrollada sobre el middleware ProActive [10], y corresponde a la plataforma SCA utilizada en este trabajo de memoria. EN GCM/ProActive los componentes son construidos utilizando *objetos activos*, un patrón de diseño consiste implementar un objeto utilizando un único *thread de control* propio para desacoplar la ejecución de un método de su invocación [11]. Este thread de control hace de intermediario entre el invocador y el objeto activo, recibiendo y administrando las invocaciones, con el fin de asegurar una acceso síncrono al objeto activo mientras se provee de una comunicación asíncrona al invocador mediante del envío de objetos Futuros como respuesta.

En GCM/ProActive un componentes es implementado a partir de un objeto común que implementa las funcionalidades deseadas. Este objeto, normalmente denominado objeto pasivo, es utilizado como raíz para construir el objeto activo encargado de administrar el acceso al componente. Cada vez que solicitamos el acceso a alguna de las interfaces del componente lo que en realidad obtendremos en un objeto auto-generado que funcionará a modo de *proxy* para la comunicación con el objeto activo del componente. De esta forma, las solicitudes enviadas por esta interfaz serán redirigidas transparentemente al objeto pasivo, o a alguno de los controladores de la membrana, de forma asincrona. El objeto pasivo por su parte contará con la seguridad de que recursos serán accedidos sincronamente. Un ejemplo de grafico de esta implementación se muestra en la Figura 1.2:

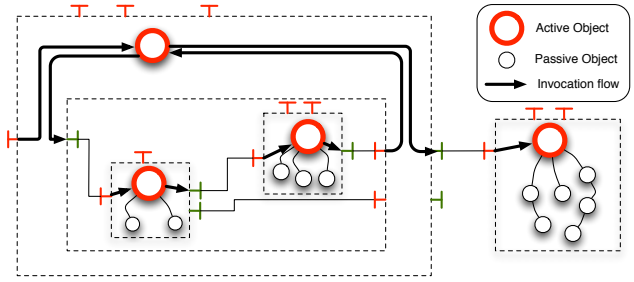


Figura 1.2: Implementación de componentes GCM en GCM/ProActive.

Los componentes de GCM/ProActive pueden ser desplegados de forma distribuida y trasparente para la estructura de la aplicación. La estructura lógica, composición de componentes, y la física, despliegue en maquinas reales, se configuran por separado son independientes el uno del otro. Un ejemplo gráfico de esto se puede apreciar en la Figura 1.3:

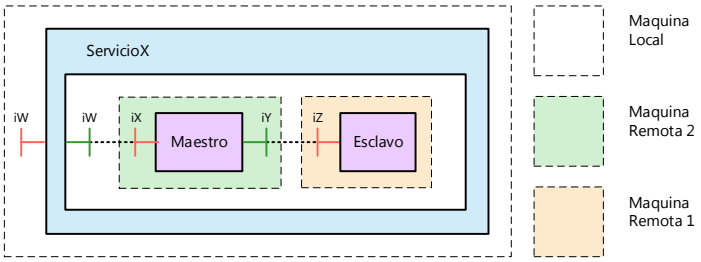


Figura 1.3: Ejemplo de un despliegue distribuido.

1.1.3. GCMScript

GCMScript es un lenguaje interpretado de dominio especifico diseñado para realizar reconfiguración sobre GCM. GCMScript extiende a FScript [12], herramienta homóloga diseñada para Fractal, para añadir soporte a características únicas de GCM como, por ejemplo, su sistema de despliegue.

En GCM, los componentes traen por defecto un conjunto de controladores que permiten administrar al componente y realizar cambios en la arquitectura de la aplicación, incluso durante el tiempo de ejecución. Los controladores permiten realizar las reconfiguraciones necesarias durante un proceso de adaptación, sin embargo, esto requiere un buen dominio de las interfaces y de su funcionamiento. Para utilizar los controladores, además, es necesario hacerlo a través de Java, lo que dificulta el proceso de adaptación si consideramos que las necesidades de reconfiguración se originan normalmente post-despliegue de la aplicación.

GCMScript resuelve estos problemas abstrayendo los conceptos de GCM en un modelo simple de manipular. El modelo esta basado en *nodos* y *ejes* donde los nodos representan elementos de la arquitectura, como componentes o interfaces, mientras que los ejes representan relaciones entre estos estos elementos, como posesión o conexión entre interfaces. Gracias a este modelo GCMScript permite navegar a través de la arquitectura del sistema y realizar reconfiguraciones sobre ella de forma interactiva y en tiempo de ejecución. Los comandos de GCMScript son traducidos en llamados de bajo nivel a los controladores y componentes de GCM, reduciendo los riesgos de una mala manipulación. Una versión simple de este modelo se puede ver en la Figura 1.4.

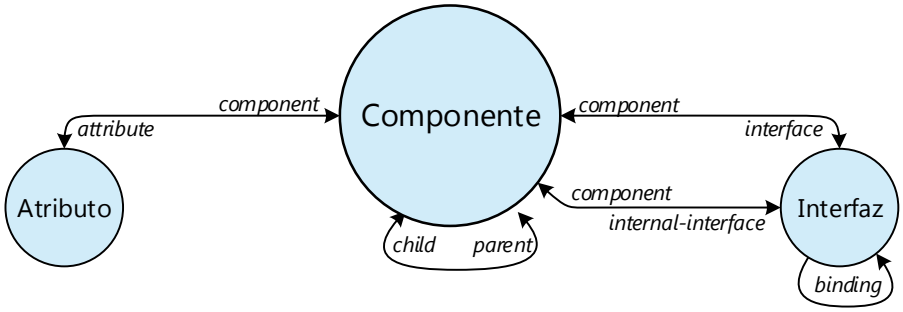


Figura 1.4: El modelo de GCMScript, sus principales nodos y ejes.

Recorriendo la arquitectura del sistema

Dado un nodo de referencia en la arquitectura, como por ejemplo el nodo de un componente, podemos utilizar los ejes para definir un *camino* hacia cualquier otro nodo de la arquitectura. El nombre de los ejes son siempre constantes y corresponden a los mostrados la Figura 1.4. Los nombres de los nodos corresponde al nombre con el que se identificó al elemento que representan. A modo de ejemplo se listarán algunas muestras código GCMScript tomando como referencia la arquitectura del sistema de la Figura 1.3. Se asumirá que el componente compuesto `ServicioX` se encuentra referenciado en la variable `$serviciox` (en GCMScript, el nombre de las variables es precedido por el signo $):

```
-- Comando basico para avanzar de un nodo a otro es:
--   nodo_origen/nombre_eje::nombre_nodo_destino
-- Ej: Referencia a componente Maestro
$serviciox/child::Maestro
-- Ej: Referencia a componente Maestro (version extendida)
$serviciox/internal-interface::iW/binding::iX/component::Maestro
```

Listado 1.1: Recorriendo la arquitectura del sistema con GCMscript

Propiedades de los nodos

Normalmente los elementos en GCM poseen propiedades que los describen o que representan un estado interno del elemento. Por ejemplo, un componente posee como propiedades su nombre y el estado de su ciclo de vida, entre otras. Estas propiedades pueden ser fácilmente consultadas, y modificada si la propiedad lo permite, a través de los nodos. Por ejemplo, continuando con el caso de la Figura 1.3:

```
-- Consultar una propiedad:
--    "nombre_de_la_propiedad(nodo_a_consultar)"
-- Ej: nombre del componente
name($serviciox);

-- Modificar el valor de una propiedad:
--    "set-nombre_de_la_propiedad(nodo, nuevo_valor)"
-- Ej: detener un componente (cambiar su estado del ciclo de vida)
set-state($serviciox, "STOPPED");
```

Listado 1.2: Propiedades de los nodos en GCMScript

Funciones y Acciones

Finalmente, es deseable poder realizar algunas modificaciones a la arquitectura del sistema además de solo recorrerla. Esto es posible gracias un conjunto de *funciones* y *acciones* básicas provistas por GCMScript. Una función consiste en un conjunto de instrucciones que permiten obtener información relevante sin realizar ningún tipo de cambio en el sistema. Una acción, por el contrario, consiste en un conjunto de instrucciones causan modificaciones en el sistema. Por lo tanto, una función solo puede ser definida utilizando otras funciones, mientras que una acción puede utilizar ambas funciones y acciones. Hacer esta diferencia permite al interprete de GCMScript tomar las medidas preventivas necesarias para evitar llevar al sistema a un estado de inconsistencia.

```
-- Esta funcion verifica que el ciclo de vida del componente este detenido
-- Ejemplo de uso: is-stopped($foo);
function is-stopped(comp) {
    return (state($comp) == "STOPPED");
}
```

```
-- Esta accion detiene el ciclo de vida de un conjunto de componentes
-- Ejemplo de uso: stop-all($foo/child::*);
action stop-all(components) {
    for comp : $components {
        set-state($comp, "STOPPED");
    }
}
```

Listado 1.3: Ejemplo de una función y una acción en GCMScript

1.2. Computación Autonómica

La computación autonómica [13] promueve la elaboración de software auto-administrado. Es decir, su objetivo es permitir que sistemas de software sean capaces de administrarse a sí mismos, minimizando así la necesidad de una interacción humana. Para hacer esto posible el sistema debe recibir un conjunto de objetivos o metas que servirán como guías para el proceso autonómico; el sistema se encargará de interpretar estos objetivos y de auto-administrarse para asegurarse de que estos objetivos se cumplan.

Los sistemas de software que poseen estas propiedades autonómicas se basan en el concepto de *elemento autonómico*. Un elemento autonómico es cualquier unidad de software que posee la capacidad de administrarse a si mismo. Estos elementos autonómicos comúnmente siguen un modelo simple y ampliamente aceptado, el cual fué propuesto por IBM [14]. El modelo, visible en la Figura 1.5, define dos tipos de entidades distintas; el *recurso administrado* y el *administrador autonómico*. El recurso administrado representa los artefactos que serán autonómicamente administrados dentro del elemento autonómico. El administrador autonómico es la entidad encargada de administrar el recurso administrado en tiempo de ejecución. Este administrador autonómico conoce los objetivos del sistema y es de su responsabilidad realizar las modificaciones que sean necesarias para asegurar que estos objetivos se cumplan.

El recurso administrado provee dos tipos de interfaces o puntos de control a través de los cuales interactúa el administrador autonómico; los *sensores* y los *efectores*. Los sensores exponen información sobre el recurso administrado que pueda ser de utilidad para el administrador autonómico, mientras que los efectores permiten realizar ajustes sobre el recurso administrado. Por lo tanto, el nivel de invasividad que tendrá el administrador autonómico se verá limitado por la implementación de estos puntos de control por parte del recurso administrado.

Administrador Autonómico

Para el administrador autonómico existe también una arquitectura de referencia, comúnmente llamada *el bucle MAPE-K* [15], visible en la Figura 1.6. Esta consiste una serie de actividades que se deben realizar dentro de un bucle infinito, estas son; *monitoreo*, *analysis*, *planificación* y *ejecución*. El nombre MAPE-K proviene de las siglas de estas cuatro actividades. La K corresponde a un componente adicional llamado *conocimiento* (*knowledge* en ingles) que

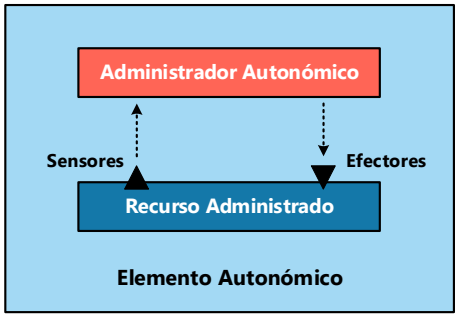


Figura 1.5: Elemento Autonómico

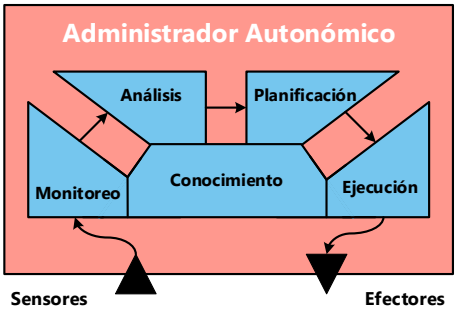


Figura 1.6: El bucle MAPE-K

representa una combinación entre los datos del sistema y los objetivos especificados por el administrador, conocimiento que se comparte en todas las actividades del bucle MAPE-K.

El bucle comienza con la fase de monitoreo, aquí se recolectan datos sobre recurso administrado a través de los sensores para generar una imagen de la situación actual. Después sigue la fase de análisis, aquí se estudia la imagen de la situación actual generada en la fase anterior y se buscan anomalías o errores que se desean ser reparados. Luego viene la fase de planificación donde, en base a los antecedentes anteriores, se determina el conjunto de acciones que deben ejecutarse para llevar al recurso administrado de su estado actual al nuevo estado deseado. Finalmente, en la fase de ejecución es donde se gatillan las acciones escogidas en la fase de planificación a través de los efectores.

Capítulo 2

Estado del Arte

En este capitulo se presentará el *framework* de monitoreo y reconfiguración propuesto para el SCA. Se expondrá su diseño teórico y se mostrará el estado de la implementación al momento de comenzar con esta memoria.

2.1. Propuesta teórica

Para mejorar la adaptabilidad de aplicaciones orientadas a servicios se propuesto un *framework* para proveer tareas de monitoreo y administración. La solución propuesta se basa en las ideas de la computación autonómica para tomar la forma de un bucle de control autonómico adjuntable a los servicios. La propuesta original basa el diseño de este *framework* tomando como referencia el modelo de componentes GCM; en particular, sobre su implementación en GCM/ProActive.

La solución propuesta consiste en la implementación del bucle de control autonómico MAPE-K utilizando un componente controlador distinto para cada fase del bucle. Estos componentes controladores (desde ahora controladores MAPE) son añadidos a la membrana del componente que se desea administrar, como se muestra en la Figura 2.1:

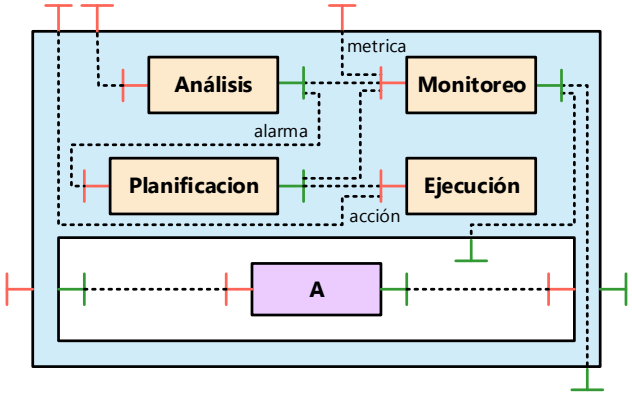


Figura 2.1: Implementación final del administrador autonómico en GCM

Los controladores MAPE están conectados entre sí. Esto permite mantener el conocimiento distribuido entre ellos, por ejemplo los datos recolectados durante la fase de monitoreo pueden ser consultados directamente al componente de monitoreo. De manera adicional, se introduce también un concepto de conocimiento compartido, donde se permite que el controlador de monitoreo pueda consultar a los controladores de monitoreo de otros componentes durante la fase de recolección de datos.

Por lo tanto, los controladores MAPE funcionan de la siguiente forma. El *controlador de monitoreo* recolectara datos sobre la ejecución del componente administrado, solicitando colaboración a controladores de monitoreo remotos de ser necesario. Luego, el *controlador de análisis* solicitará estos datos para estudiarlos y decidir si es que nos encontramos en un estado potencialmente indeseado o no. El *controlador de planificación* consultará el dictamen del controlador de análisis y estudiará los datos del controlador de monitoreo para definir un conjunto de reconfiguraciones que deben ser aplicadas al componente administrado para llevarlo al estado óptimo deseado. Finalmente, estas reconfiguraciones serán efectuadas por el *controlador de ejecución* quien interactuará directamente con la implementación de GCM.

Se debe tener presente que el modelo GCM se limita a lidiar con la estructura y la comunicación de los componentes que encapsulan el código de negocio de una aplicación, y no con el contenido de estos. De la misma forma, esta propuesta se limita a proveer reconfiguraciones autonómicas a nivel de componentes, sin hacer intrusión en el contenido de la aplicación.

A continuación se describirá en detalle cada uno de los elementos involucrados en este modelo:

2.1.1. Sensores y Efectores

La arquitectura de referencia del bucle de control autonómico MAPE-K sugiere que el recurso administrado debe proveer sensores que permitan recolectar información y efectores que permitan realizar las reconfiguraciones necesarias. En este caso los sensores y efectores son provistos por las interfaces de bajo nivel de la implementación de GCM.

A modo de sensor se utiliza un sistema un sistema de eventos internos. GCM/ProActive ofrece un sistema de eventos que anuncia cada cambio en el estado de las solicitudes envíadas entre componentes. Cada vez que se comienza a atender una solicitud, se actualice el valor de un futuro o cualquier otra actividad relacionada ocurra, se genera un evento descriptivo con información relevante adjuntada a él. Esta información esta focalizada en la solicitud que generó el evento e incluye datos como; el identificador único de esta solicitud, el componente de origen, el componente de destino, el nombre de la interfaz utilizada, el nombre del método utilizado y el tiempo de generación del evento, entre otros. El controlador de monitoreo podrá recolectar los datos que necesite atrapando estos eventos de GCM/ProActive.

A modo de efectores se utilizan las interfaces de reconfiguración estándar provistas por la implementación de los componentes en GCM/ProActive. El controlador de ejecución podrá llamar a estas interfaces a través de GCMScript.

2.1.2. Monitoreo

El controlador de monitoreo tiene por objetivo principal recolectar la información requerida desde el componente administrado y exponerla como un conjunto **métricas** para que pueda ser consultada. Este controlador expone una interfaz para permitir que otros componentes consulten el valor de sus métricas, adicionalmente, puede poseer también cero o más referencias a otros controladores de monitoreo externos para consultar el valor de sus métricas. Para poseer una referencia a otro controlador de monitoreo se debe cumplir la siguiente condición: si un componente A pose una referencia funcional al componente B, entonces el controlador de monitoreo de A puede poseer una referencia al controlador de monitoreo de B. De esta forma se hace posible definir métricas en base al valor de otras métricas, como se muestra en la Figura 2.2:

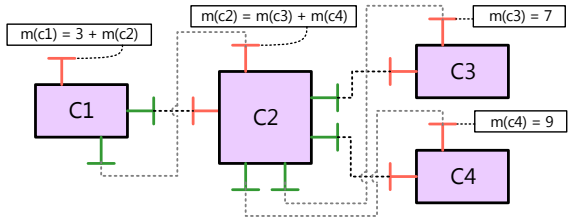


Figura 2.2: Referencias a controladores de monitoreo externos

Internamente, el controlador de monitoreo debe recolectar información sobre el estado de la ejecución de la aplicacion. Esta información es representada y almacenada internamente en forma de **registros**. Estos registros son leídos por las métricas para realizar sus mediciones, cada vez que el valor de estos registros sea actualizado el controlador se encargará de recalcular el valor de sus métricas.

El diseño de este controlador contempla el uso un componente compuesto que modulariza a otros cuatro subcomponentes, como se muestra en la Figura 2.3:

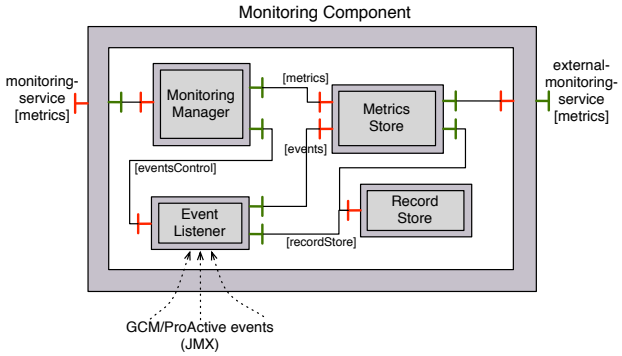


Figura 2.3: Controlador de monitoreo [16]

- **EventListener:** Este componente captura los eventos de GCM/ProActive y traduce la información contenida a una representación interna denominada *Registros*. Posee una referencia al componente *RecordStore*, para almacenar los registros generados, y una referencia al *MetricStore*, para anunciar que se capturó un nuevo evento.

- **MetricsStore:** Este componente almacena las métricas y se encarga de recalcular sus valores cada vez que recibe una notificación desde el *EventListener*. El calculo del valor de las métricas se basa en la información contenido en los *registros*, por lo tanto se cuenta con una referencia a *RecordStore*. Adicionalmente, este componente es quien posee las referencias a controladores de monitore externos.
- **RecordStore:** Este componente almacena los *registros* generados, se encarga de que los registros sean fácil de actualizar y de consultar.
- **MonitoringManager:** Este componente esta encargado de la comunicación con otros componentes y de administrar el proceso de monitoreo.

2.1.3. Análisis

El controlador de análisis tiene por objetivo verificar que un conjunto de reglas o condiciones predefinidas se cumplan. Estos requisitos se denominan *Service Level Objectives* (SLO) y juntos conforman un contrato ente el servicio brindado y el cliente denominado *Service Level Agreement* (SLA). Por ejemplo, podríamos definir un SLO para asegurar una buena experiencia de usuario requiriendo que las solicitudes entrantes deban ser servidas en un tiempo menor o igual a T_0. En otras palabras, el objetivo de este controlador es verificar que el SLA se respete.

Los SLOs son definidos en base a métricas, por lo tanto el controlador de análisis puede solicitar el valor de las métricas a través una referencia al controlador de monitoreo. En caso de que un SLO sea violado, se genera una **alarma** que será enviada al controlador de planificación a modo de notificación.

El diseño de este controlador contempla el uso un componente compuesto que modulariza a otros cuatro subcomponentes, como se muestra en la Figura 2.4:

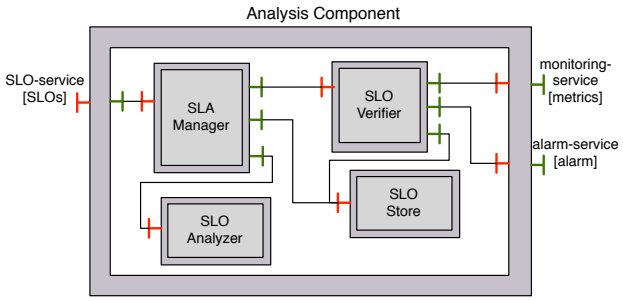


Figura 2.4: Controlador de análisis [16]

- **SLOStore:** Este componente almacena los SLOs cargados en el controlador. Permite también habilitar o deshabilitar un SLO.
- **SLOAnalyzer:** Este componente recibe los SLOs cargados traduce a una representación interna para que puedan ser almacenados en el *SLOStore*. Esta pensado para

ser reemplazado facilmente en caso de requerir un cambio en el modelo utilizado para representar los SLOs.

- **SLOVerifier:** Verifica que los SLOs almacenados en el *SLOStore* se respeten. Este componente tiene una referencia al controlador de monitoreo para consultar el valor de las métricas durante el proceso de verificación.
- **SLAManager:** Este componente esta encargado de recibir los SLOs cargados y administrar el proceso de análisis.

2.1.4. Planificación

El controlador de planificación esta encargado de reaccionar ante un incumplimiento de un SLO. Su deber consiste en generar una secuencia de acciones que cambiarán el estado del sistema para volver a cumplir con el SLO trasgredido.

Como se muestra en Figura 2.5, este controlador esta diseñado como un componente compuesto. El componente principal es el *StrategyManager*, donde se reciben las alarmas y se gatilla la ejecución de las distintas estrategias. Cada estrategia se implementa a través de un componente denominado *Planner*, varios *Planners* son soportados para permitir la ejecución de varias estrategias en paralelo. Cada *Planner* tiene acceso a una referencia del controlador de monitoreo, para recolectar los antecedentes necesarios, y una referencia al controlador de ejecución, donde se envían finalmente las acciónes a tomar.

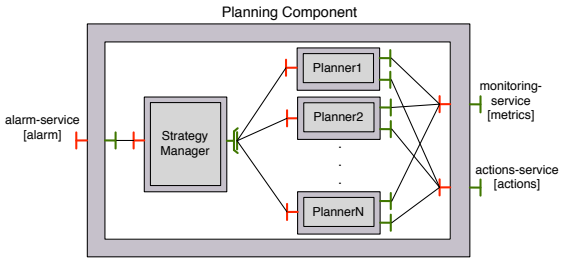


Figura 2.5: Controlador de planificación [16]

2.1.5. Ejecución

El controlador de ejecución esta encargado de aplicar sobre el componente administrado la secuencia de acciones que han sido determinadas por el controlador de planificación. Estas acciones son aplicadas finalmente utilizando GCMScript (efector). Como se mencionó anteriormente, esta acciones se aplicarán sobre el sistema utilizando GCMScript, debiendo traducir las acciones a comandos GCMScript de ser necesario.

El diseño de este controlador contempla el uso un componente compuesto que modulariza a otros tres subcomponentes, como se muestra en la Figura 2.6:

- **Translation:** Este componente se encarga de traducir la representación actual de una acción al lenguaje GCMScript.

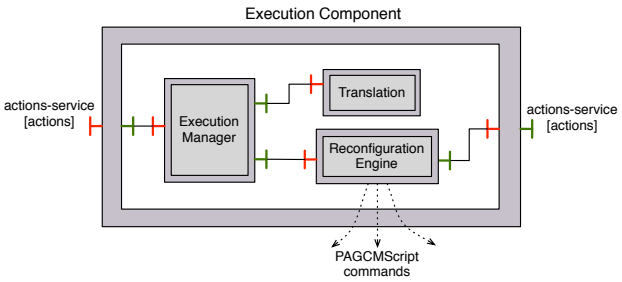


Figura 2.6: Controlador de ejecución [16]

- **ReconfigurationEngine:** Este componente encapsula un interprete de comandos GCMScript. Su deber es ejecutar las acciones previamente traducidas por *Translation* y ejecutarlas.
- **ExecutionManager:** Este componente se encarga de recibir las acciónes y administrar el proceso de ejecución.

2.2. Estado de la implementación

En esta sección se presentará el estado de los controladores MAPE en la implementación de referencia al momento de comenzar con este trabajo de memoria. El controlador de planificación, sin embargo, es deliberadamente excluido debido a que no contaba con ningún tipo de implementación.

2.2.1. Controlador de monitoreo

El controlador de monitoreo existente sigue fielmente el diseño original e implementa los cuatro subcomponentes definidos.

El proceso de monitoreo comienza cuando el `EventListener` captura los eventos de GCM/ProActive y la genera los registros. Los datos recolectados a través de estos eventos describen el estado de una solicitud recibida o enviada por el componente administrado. Cada registro representa una solicitud en particular y se clasifican en dos tipos:

- `INCOMING_RECORD` : Información sobre una solicitud entrantes.
- `OUTGOING_RECORD` : Información sobre una solicitud salientes.

A medida que la ejecución avanza nuevos eventos relacionados con la misma solicitud van llegando y el registro se va actualizado. Por ejemplo, un registro tipo `INCOMING_RECORD` se creará cada vez que se reciba una solicitud, y se actualizará cuando la solicitud comience a ser servida y cuando se termine de servir. Cada vez que un cambio en los registros ocurre se emite un evento especial denominado `RecordEvent`, de los cuales hay cuatro tipo:

- `NEW_INCOMING` : Nuevo registro de tipo `INCOMING_RECORD`.
- `INCOMING` : Registro de tipo `INCOMING_RECORD` actualizado.
- `NEW_OUTGOING` : Nuevo registro de tipo `OUTGOING_RECORD`.
- `OUTGOING` : Registro de tipo `OUTGOING_RECORD` actualizado.

Los *RecordEvent* son generados en el *EventListener* y enviados al *MetricStore*. El *MetricStore* recalculará las métricas necesarias basandose en el *RecordEvent* recibido.

Las métricas por su parte son implementadas como un objeto Java. Toda métrica debe debe extender la clase abstracta `Metric` e implementar los siguientes métodos:

```
abstract Object calculate(); // Calcula el valor de la métrica
abstract Object getValue();  // Obtiene el último valor calculado
```

Listado 2.1: Principales métodos de una métrica

Las métricas poseen métodos para suscribirse a eventos *RecordEvent*, de esta forma, cada vez que el *MetricStore* recibe un evento *RecordEvent* recalculará todas aquellas métricas suscritas a dicho evento, y ninguna más. Las métricas poseen también un método para habilitar o deshabilitar las suscripciones, donde una métrica con suscripciones deshabilitada ingnorará toda suscripción y no se recalculará.

En la búsqueda de un acercamiento hacia una solución tecnológicamente agnóstica, se provee junto con esta implementación una *librería de métricas predefinidas*. Esta librería pretende poner a disposición un conjunto de métricas razonablemente utiles y potencialmente requeridas en la practica. De esta manera, una interacción directa con la implementación de las métricas será necesaria únicamente en situaciónes de alta complejidad. Un ejemplo de estas métricas predefinidas se lista a continuación:

- Métrica *avgInc*: Tiempo promedio empleado en solicitudes entrantes.
- Métrica *avgOut*: Tiempo promedio empleado en solicitudes salientes.
- Métrica *maxInc*: Mayor tiempo empleado en una solicitud entrante.

Monitoreo remoto

En esta implementación incluyen también las referencias no funcionales a controladores de monitoreo remotos, tal como se vio en la Figura 2.2. El diseño original de estas referencias es genérico y no considerá el caso particular de los componentes compuestos de GCM. Sin embargo, esta versión se apega a la regla de que un controlador de monitoreo de un componente C_1 solo puede tener una referencia a otro controlador de monitoreo de un componente C_2 si y solo si C_1 posee una referencia a C_2. Un ejemplo de caso se muestra en la Figura 2.7, donde el controlador de monitoreo del componente compuesto `B` solo posee una referencia al subcomponente S_1, con quien tienen comunicación directa, y no con S_2. Por la misma razón, el controlador de S_2 posee una referencia a B y no a C.

A pesar de ser controlador de monitoreo completamente funcional, actualmente no se utilizan las referencias no funcionales en el calculo de métricas. Esta implementación provee algunos métodos que utilizan estas referencias para hacer un seguimiento de las solicitudes a lo largo de los componentes del sistema. Sin embargo, una métrica como tal no puede hacer uso de estas referencias, viéndose limitada a calcular su valor utilizando únicamente los registros del *RecordStore*

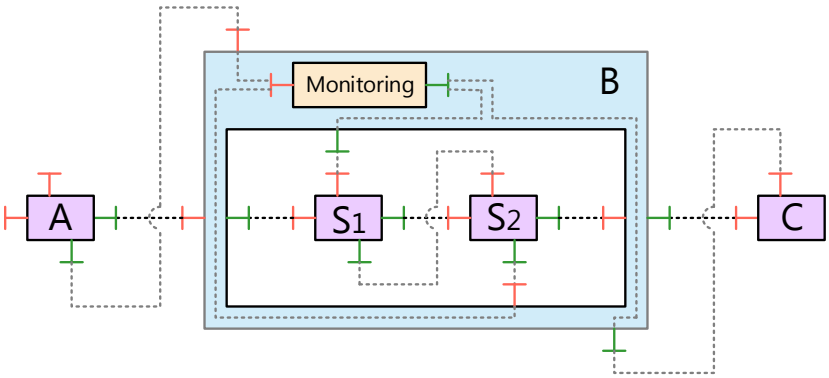


Figura 2.7: Referencias a controladores de monitoreo remotos dentro de un componente compuesto

2.2.2. Controlador de Análisis

El controlador de análisis existente se encuentra en un estado muy temprano de desarrollo y no logra proveer las funcionalidades esperadas de él. Sin embargo, del código disponible se puede inferir representación que se le intentó dar a los SLOs.

En esta implementacion los SLOs son representados por un arreglo de `Strings` con tres elementos: el nombre de la métrica, el nombre de la condición y el valor umbral para la métrica. El nombre de la métrica corresponde al nombre utilizado para registrar la métrica que se desea verificar en el controlador de monitoreo, mientras que el nombre de la condición se utiliza, probablemente, para identificar una condición dentro de un conjunto predeterminado de condiciones. De esta forma, un SLO consistiría compara el valor de la métrica con el valor umbral indicado utilizando la condición especificada.

Continuando con la idea de la *librería de métricas*, es lógico esperar de esta implementación una *librería de condiciones* predefinidas, evitando una manipulación directa en la implementación de los SLOs.

2.2.3. Controlador de Ejecución

El estado de la implementación del controlador de ejecución no sigue el diseño original y esta implementado como un único componente primitivo. Este componente primitivo es en realidad un *wrapper* de un objeto controlador denominado *ReconfigurationController* perteneciente al proyecto de GCMScript. El *ReconfigurationController* provee las mismas funcionalidades esperadas del controlador de ejecución, salvo por el hecho de que es un objeto controlador y no un componente controlador. En consecuencia, la interfaz del controlador de ejecución posee los mismos métodos que la interfaz del *ReconfigurationController*:

`load(fileName)`: Permite cargar definiciones a través de un archivo que contenga código GCMScript y las deja disponible para futuras invocaciones.

`getGlobals()`: Retorna el conjunto de variables globales definidas actualmente en este interprete.

`execute(source)`: Ejecuta un fragmento de codigo GCMScript.

Este controlador contiene en su interior un interprete de GCMScript. Por defecto, se define en este interprete la variable `$this` que hace referencia al componente administrado, lo que sirve de cómodo punto de partida para las reconfiguraciones realizadas a través de este controlador.

Debido a que cada componente puede contener sus propias librerías de reconfiguración cargadas en sus controladores de ejecución, es importante mencionar la existencia de un una acción especial de GCMScript denominada `remote-execute`. Esta acción que permite enviar comandos GCMScript para ser ejecutados en los controladores de ejecución de otros componentes. Esta acción permite, por ejemplo, conservar el nivel de abstracción introducido por los componentes compuestos, como se muestra en la Figura 2.8:

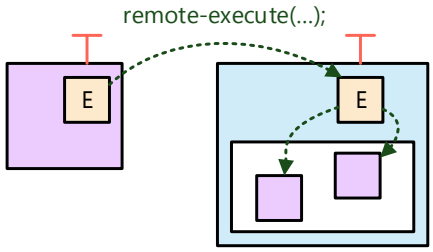


Figura 2.8: Ejecución remota de comandos GCMScript.

Capítulo 3

Implementación de Componentes Auto-Adaptables

En este capitulo se presentará el trabajo realizado durante la implementación de los controladores MAPE para proveer auto-adaptabilidad a los componentes GCM. Se tomará como base la implementación actual presentada en el capitulo anterior y se irá completando hasta lograr un ciclo autonómico completamente funcional.

3.1. Una API en forma de objetivos administrativos

Unas de las principales ventajas detrás de las ideas de la computación autonómica es permitir concentrarnos en *qué* es lo que queremos, abstrayéndonos de la necesidad de especificar el *cómo* lo haremos. Esto se logra gracias a que solo se deben especificar el conjunto de objetivos administrativos deseados y el administrador autonómico se encargará de que esos objetivos se cumplan. Mientras tanto, uno es libre en la implementación de representar estos objetivos como estime conveniente.

En la implementación actual, estos objetivos quedan definidos a través de tres elementos; las *métricas*, los *SLOs* y los componentes *Planners*. A excepción de las métricas, estos elementos no proveen el nivel de flexibilidad esperado para la definición del comportamiento auto-adaptable de los componentes. Particularmente, en el caso de los *Planners*, la manipualción de estos elementos en tiempo de ejecución implicaría necesariamente una modificación en la arquitectura del controlador de planificación, presentando el mismo problema que este *framework* intenta resolver.

El problema de los SLOs radica en la perdida de generalidad. La relación uno-a-uno impuesta entre métrica, condición y valor umbral deja fuera a un sin número de situaciones potencialmente deseables. Por ejemplo, es posible desear un SLO sobre una métrica condicionada al valor de otra métrica, o bien, un SLO cuya verificación requiera de los resultados de verificaciones anteriores.

Por otro lado, las estrategias de reconfiguración representadas por los *Planner* pueden variar considerablemente entre caso y caso, haciendo muy difícil diseñar un conjunto de *Planners*

genéricos. Esto obliga a completar la implementación del controlador de planificación cada ves que este *framework* deba ser utilizado, demandando conocimientos muy específicos sobre la implementación de este *framework* y haciéndolo muy poco utilizable en la práctica.

Para simplificar la definición de estos objetivos, soportar situaciones de mayor complejidad y mejorar la usabilidad de este *framework* en general, se propone una nueva especificación para la definición de los objetivos administrativos. De la misma forma que las métricas son implementados como objetos independientes, en esta propuesta se representan los SLOs a través de un objeto denominado *Regla* y las estrategias de reconfiguración a través de un objeto denominado *Plan*.

La definición de SLOs a través del objeto *Regla* no entorpece de modo alguno la definición de una *librería de reglas* predefinidas. En situaciones especiales, el uso de *Reglas* facilitaría la implementación de SLOs de mayor complejidad.

A su vez, la definición de estrategias de reconfiguración a través del objeto Plan nos permite prescindir de la implementación del controlador de planificación. Sin embargo, como se verá mas adelante, fue necesario rediseñar el controlador de planificación para reemplazar los componente *Planner* por estrategias de reconfiguración adjuntables en forma de *Planes*. Aunque la complejidad inherente de las estrategias de reconfiguración no permite la definición una librería de planes, se mostrará más adelante que esta definición puede ser utilizada para proveer estrategias sencillas sin la necesidad de implementar efectivamente un *Plan*.

Los elementos propuestos en esta sección pretenden ser lo suficiente mente flexibles para permitir una completa manipulación del comportamiento auto-adaptable de los componentes. Por lo tanto, la API propuesta para este *framework* de monitoreo y reconfiguración se resume en la implementación y manipulación de estos elementos. Un diagrama de clases de estos elementos se puede observar en la Figura 3.1:

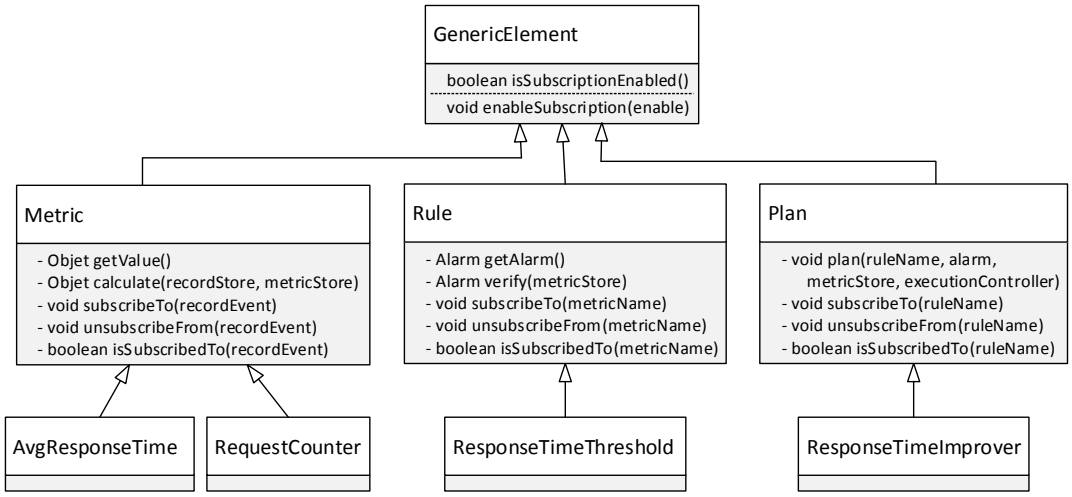


Figura 3.1: Diagrama de clases de los objetivos de adaptabilidad

A continuación, se presentará la implementación de estos tres elementos que conforman estos objetivos; las métricas, las reglas y los planes.

3.1.1. Métricas

Para las métricas se conservó la estructura general de versión existente al comenzar este trabajo y se realizó solo un cambio importante: los recursos necesarios para calcular el valor de una métrica ahora se entregarán como parámetros del método `calculate`. Actualmente, `Metric` posee una referencia a *RecordStore* en forma de una variable de instancia cuyo valor es asignado al momento de cargar la métrica al controlador de monitoreo. Para destacar la prescencia de este recurso ahora es entregado como parámetro del método `calculate`, junto con él se añadió también una referencia al *MetricStore* para que las métricas puedan basar sus cálculos en el valor de otras métricas.

```
// Calcula el valor de la métrica
Object calculate(RecordStore recordStore, MetricStore metricStore);

// Retorna el ultimo valor calculado de la métrica
Object getValue();
```

Listado 3.1: Principales métodos de una métrica

El resto de los métodos se encargan de administrar las suscripciones a los eventos de cambio en el valor de los registros, los *RecordEvents*. Estos métodos permite añadir, remover y consultar suscripciones, y permiten también deshabilitar completamente el sistema de suscripciones. Cuando las suscripciones se deshabilita se ignorará cualquier suscripción realizada y la métrica se dejará de ser recalculada automáticamente.

```
// Suscripción a cambios en los registros para ser recalculada automáticamente
void subscribeTo(RecordEvent e);
void enableSubscription(boolean enable);
```

Listado 3.2: Principales métodos de suscripción en una métrica

3.1.2. Reglas

Al igual que las métricas, las reglas se implementaron como un objeto Java. Toda regla deberá extender a la clase abstracta `Rule` cuyos principales métodos se listan a continuación:

```
// Verificar que la regla se cumpla
Alarm verify(MetricStore metricStore);

// Retorna la ultima alarma emitida
Alarm getAlarm();
```

Listado 3.3: Principales métodos de una regla

El método `verify` recibirá una referencia al *MetricStore* a través de la cual podrá consultar el valor de las métricas que necesite para verificar que la condición que representa se cumpla. Este método retornará una alarma que consisten en el enumerado `Alarm`, que por defecto

provee cuatro posibles valores: *Ok*, *Warning*, *Violation*, *Error*. El método `getAlarm`, por su parte, esta pensado para retornar la ultima alarma emitida por esta regla.

Adicionalmente la clase `Regla` provee métodos para suscribirse a métricas; cada vez que alguna de las métricas suscritas cambie de valor, la regla se volverá a verificar automáticamente. Estos métodos son los mismos métodos utilizados en la implementación de las métricas, pero esta vez la suscripción se debe realizar indicando el nombre asignado a la métrica al momento de ser cargada en el controlador de monitoreo.

```
// Suscripción a cambios en los valores de las métricas
void subscribeTo(String metricName);
void enableSubscription(boolean enable);
```

Listado 3.4: Principales métodos de suscripción en una regla

Estas reglas hacen posible la definición de SLOs sin la necesidad para el usuario de implementar manualmente una regla. Esto se hace posible proporcionando un conjunto de reglas predefinidas para criterios de comparación comunes. Estas reglas pueden ser solicitadas a una *librería de reglas* utilizando los siguientes parámetros: nombre de la métrica, nombre del criterio comparativo, valor umbral de la métrica. Un ejemplo de uso de esta librería se lista a continuación:

- *("foo", "greaterThan", 10)*: Desde estos parametros es posible generar una regla que verificará el valor de la métrica `foo` y retornará una alarma si su valor es mejor o igual a `10`. Por defecto esta regla se configurará para habilitar sus suscripciones y estar suscrita a la métrica `foo`.

3.1.3. Planes

Al igual que las reglas y las métricas, los planes se implementaron como un objeto Java. Todo plan deberá extender la clase abstracta `Plan` cuyo método principal se lista a continuación:

```
// Planea una serie de acciones dado que 'ruleName' emitió 'alarm'.
void plan(String ruleName, Alarm alarm, MetricStore metricStore,
          ExecutionController execution);
```

Listado 3.5: Principal método de un plan

Dada una alarma y el nombre utilizado para registrar la regla que emitió esta alarma, el método `plan` se encargará de determinar la serie de acciones necesarias a tomar para llevar el sistema de vuela al estado deseado. Este método recibirá como parámetro una referencia al *RecordStore* para solicitar información a las métricas, y una referencia al *ExecutionController* a traves del cual podra solicitar que se ejecuten las acciones que considere necesarias.

La clase `Plan` también provee métodos de suscripción, esta vez para suscribirse a reglas; cada ves que una de las reglas suscritas emita una alarma se ejecutará automáticamente el método `plan`.

```
// Suscripción a la emision de alarmas de una regla
void subscribeTo(String ruleName);
void enableSubscription(boolean enable);
```

Listado 3.6: Principales métodos de suscripción en una regla

Normalmente, las estrategias de reconfiguración consisten en la implementación de algoritmos específicos que varían en cada arquitectura. Por esta razón, no es posible implementar una librería de planes, sin embargo, es posible proveer una implementación de un plan genérico básico que ejecute despues de cada alarma una determinado comando GCMScript. La instanciación de un plan de este tipo requiere de al menos tres parametros; nombre de la regla, tipo de alarma y el comando GCMScript a ejecutar. Un ejemplo de esta tecnica se lista a continuación:

- *("foo", "VIOLATION", "action-example();")*: A partir de estos parámetros es posible generar un plan que trate de ejecutar el comando `action-example();` cada vez que reciba la alarma `VIOLATION` desde la regla `foo`. Por defecto este plan se configurará para habilitar sus subscripciones y estar suscrito a la regla `foo`.

3.2. Controladores MAPE

En esta sección mostraremos como se terminaron de implementar cada un de los controladores MAPE, con excepción del controlador de ejecución donde se reutilizó la versión ya existente sin ningún tipo de modificación.

3.2.1. Controlador de Monitoreo

.

La versión actual del controlador de monitoreo funciona correctamente y solo falta permitir a las métricas puedan consultar el valor de otras métricas, tanto locales como remotas. Tomando como base esta versión, se terminó de implementar el controlador y se mejoraron algunos aspectos con el fin de mejorar su usabilidad. A continuación se presentarán los cambios más relevantes realizados a este controlador:

Nuevo tipo de registro: `OUTGOING_VOID_RECORD`

Los registros representan el estado de una solicitud enviada o recibida por el componente administrado. Cada registro consiste en una serie de datos relacionados con dicha solicitud, como por ejemplo; el nombre del componente que emitió la solicitud o la marca de tiempo de cuando fue emitida. Las solicitudes en general comparten un conjunto de atributos comunes, sin embargo, dependiendo del ciclo de vida que posea cada solicitud, los eventos de relevancia registrables pueden variar.

En la Figura 3.2 se clasifican las solicitudes en tres tipos distintos segun sus ciclos de vidas. Las *solicitudes entrantes* al momento de ser recibidas son ubicadas en una cola de solicitudes y eventualmente se comenzarán a servir. Las *solicitudes salientes* son enviadas al componente servidor e inmediatamente se recibe un valor futuro a modo de respuesta, cuando la solicitud haya sido realmente servida se recibirá la respuesta real. Finalmente, las *solicitudes salientes void* son aquellas que no esperan respuesta alguna, por lo tanto una vez enviadas no queda nada más por registrar.

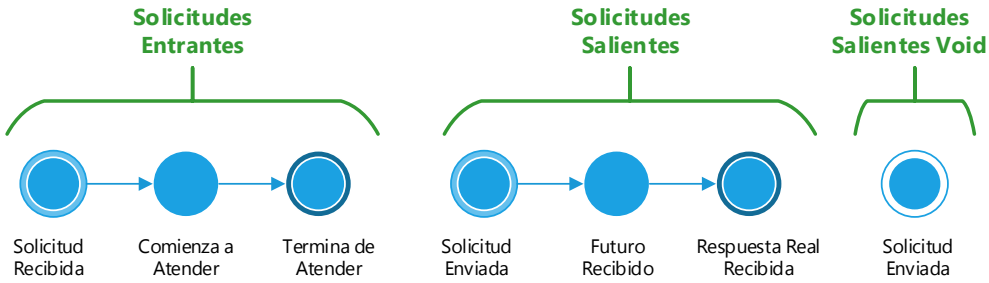


Figura 3.2: Clasificación de solicitudes GCM según ciclo de vida

Actualmente existen dos tipos de registros, los `INCOMING_RECORD` para solicitudes entrantes y los `OUTGOING_RECORD` para solicitudes salientes en general. En la implementación del registro se provee un método llamado `isVoid` que permite identificar aquellas solicitudes salientes que no esperan una respuesta, advirtiéndonos de que algunas marcas de tiempo nunca serán asignadas. Para evitar confusiones y facilitar el manejo de estos registros se añadió un nuevo tipo de registro denominado `OUTGOING_VOID_RECORD`, dedicado exclusivamente a las *solicitudes salientes void*.

Redefinición de los *RecordEvent*

Las métricas pueden ser calculadas automáticamente registrándose a alguno de los eventos de cambio en el valor de los registros *RecordEvent*. Actualmente existen cuatro tipos de *RecordEvent*: `NEW_INCOMING`, `INCOMING`, `NEW_OUTGOING` y `OUTGOING`.

Los tipos de *RecordEvent* actuales son poco intuitivos y entregan un pobre control sobre la situación. Por ejemplo, si deseamos recalcular una métrica cada vez que se comience a servir una solicitud debemos suscribir la métrica al evento `INCOMING`. Sin embargo, esto no solo recalculará la métrica cuando se comience a servir una solicitud, también se recalculará cuando se termine de servir, introduciendo un *overhead* innecesario en el sistema.

Para resolver este problema se redefinieron los tipos de *RecordEvent*, esta vez basandose en el ciclo de vida de las solicitudes. Siguiendo el diagrama de la Figura 3.2, la nueva categorización de queda entonces de la siguiente manera:

Incoming record	
`REQUEST_RECIVED`	: se recibe una solicitud entrante.
`SERVICE_STARTED`	: se comienza a servir una solicitud.
`SERVICE_ENDED`	: se termina de servir una solicitud.

Outgoing record	
`REQUEST_SENT`	: se envía una solicitud.
`FUTURE_RECEIVED`	: se recibe un futuro como respuesta.
`RESPONSE_RECEIVED`	: se recibe la respuesta real.

Outgoing void record	
`VOID_REQUEST_SENT`	: se envía una solicitud tipo void.

Calculando métricas en base a otras métricas

La nueva implementación de las métricas, introducida en la sección 3.1.1, permite que las métricas puedan acceder al *MetricStore* y basar sus calculos en el valor de otras métrica. Para consultar el valor a una métricas local el *MetricStore* posee el método `getValue(metricName)`, para consultar el valor de una métrica remota se agrega un nuevo método `getValue(metricName, interfacesPath)`.

```
Object getValue(String metricName);
Object getValue(String metricName, String[] interfacesPath);
```

Listado 3.7: Consulta de métricas usando *MetricStore*

El nuevo argumento `interfacesPath` corresponde a la ruta de interfaces que conecta al componente local con el componente remoto dueño de la métrica a consultar. Esta ruta se compone de los nombres de las interfaces utilizadas para conectar cada uno de los componentes intermediarios. Por ejemplo, como se muestra en la Figura 3.3, existen dos rutas de interfaces posibles entre el componente A y el componente C. La primera se indica con color verde y considera a B como un componente cualquiera: $\{b, c\}$. La segunda ruta se indica con color celeste y considera a B como un componente compuesto, ingresando en su implementación: $\{b, s1, s2, c\}$.

Este método fue posible de implementar debido a que las interfaces no funcionales utilizadas para conectar los controladores de monitoreo utilizan el mismo nombre que la interfaz funcional que las originó, más un sufijo. Continuando con el ejemplo de la Figura 3.3, la interfaz no funcional que conecta al controlador de monitoreo de A con el de B fue creada por que A posee una referencia de B a través de la interfaz b, y por lo tanto su nombre queda automáticamente definido como *b-remote-monitoring-controller*. Sabiendo esto, más el hecho de que los nombres de las interfaces no se pueden repetir en un mismo componente, se implementó el nuevo `getValue` de tal forma que navegara a través de las referencias a controladores de monitoreo remotos, utilizando la ruta de interfaces como guía.

Es importante destacar el hecho de que, cuando se define la ruta de interfaces, para ingresar a la implementación del componente compuesto se debe agregar explícitamente la interfaz que conecta al componente compuesto con su subcomponente. Sin embargo, para salir de

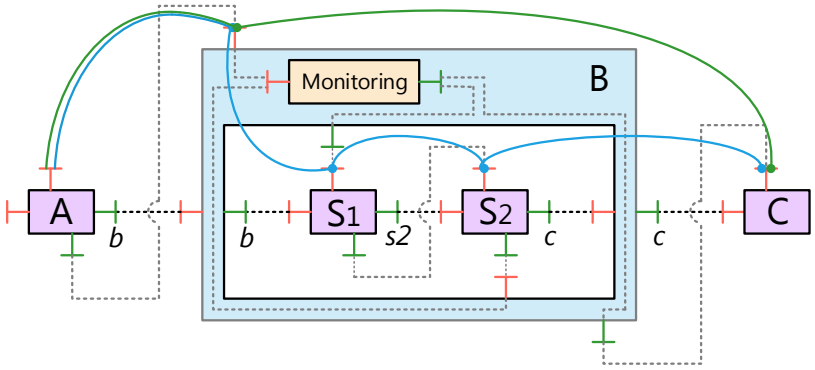


Figura 3.3: Nuevas referencias a controladores de monitoreo remotos dentro de un componente compuesto.

la implementación del componente compuesto, esto no es necesario. Esta diferencia se hace para ser consecuentes con la modularización impuesta por el componente compuesto. Los componentes deben ser entendidos como proveedores de servicios, de esta manera podemos entender un componente compuesto como una agrupación de servidores que colaboran entre sí para proveer y ser expuestos como un único meta-servicio. Bajo esta idea, y continuando con el ejemplo de la Figura 3.3, la diferencia en la ruta de interfaces se justifica de la siguiente manera:

- *Entrar al componente compuesto se solicita explícitamente* Los servicios de A dependen del meta-servicio de B y no directamente del servicio de S_1, por lo tanto la implementación interna de B no deberían ser del interés para los asuntos de A. Normalmente se espera que B monitoreé el subsistema que modulariza y resuma la información de utilidad en una serie de métricas que A pueda consultar sin necesidad de conocer la implementación de B.
- *Salir del componente compuesto es inmediato* Los servicios de S_2 dependen directamente de C y, aun que deba acceder a B para alcanzar a C, sus asuntos no tienen ninguna relación con B. Por lo tanto, por la misma razones que B intercepta y redirige las solicitudes funcionales de S_2 hacia C, el controlador de monitoreo de B intercepta y redirige las solicitudes no funcionales desde el controlador de monitoreo de S_2 hacia el de C.

Notificación de cambio en las métricas

En la propuesta original se menciona que el controlador de análisis sería notificado cada vez que una métrica es recalculada, pero en el diseño no se indica cómo. De la misma forma que el controlador de análisis envía alarmas al controlador de planificación, se añadio una nueva interfaz a través de la cual el controlador de monitoreo envía el nombre de la métrica recalculada al controlador de análisis. Esta interfaz se denominó *MetricEventListener* y es el *MetricStore* el encargado de notificar al contralor de análisis a traves de esta interfaz cada vez que una métrica sea recalculada.

3.2.2. Controlador de Análisis

Por razones practicas, se decidió implementar este controlador como un único componente primitivo encargado de almacenas las reglas, verificarlas automáticamente y notificar las alarmas emitidas al controlador de planificación. La razón de esta desición radica en que las actividades de análisis actuales carecen de una complejidad que justifique el uso de más de un componente, en este caso un único componente primitivo provee los mismos resultados y reduce los esfuerzos de desarrollo.

Como ya se definió en la sección 3.1, las reglas pueden suscribirse a métricas locales para ser verificadas automáticamente cada vez que el valor de esas métricas cambien. Para hacer esto posible se añadió a este controlador la interfaz *MetricEventListener*, a través de la cual recibirá desde el controlador de monitoreo el nombre de las métricas recalculadas recientemente. De esta forma, cada vez que se reciba una notificación de cambio en una métrica se verificarán nuevamente todas las reglas suscritas a dicha métrica. La nueva relación entre controlador de monitoreo y análisis se muestra en la Figura 3.4:

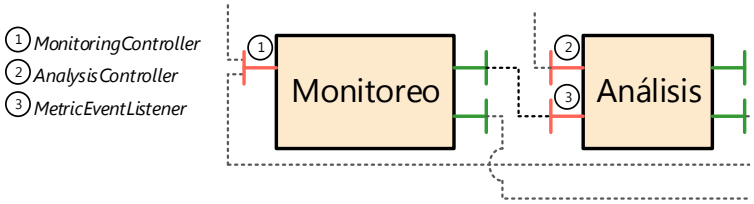


Figura 3.4: El *MetricChangeListener* notifica los cambios en las métricas

Durante una verificación de una regla se provee de una referencia al controlador de monitoreo para acceder al valor de las métricas, y como resultado de la verificación se retorna una alarma. Las alarmas son representadas a través del enumerado *Alarm* dentro del cual hay cuatro posibles valores: `OK`, `WARNING`, `VIOLATION` y `ERROR`. Estos tipos de alarma poseen un valor numérico asociado denominado *severidad* para comparar la gravedad de las alarmas. Toda alarma emitida de severidad mayor a `OK` es notificada al controlador de planificación.

Para notificar la alarmas emitidas se utilizó la interfaz *AlarmListener* a través de la cual se envía un *AlarmEvent* que contiene la alarma emitida y el nombre de la regla que la emitió.

3.2.3. Controlador de Planificación

Debido a la introducción de los planes en los objetivos administrativos, el controlador de planificación ya no contiene los algoritmos de reconfiguracion como subcomponentes *Planner*, sino que son cargados forma de planes. Este controlador fue implementado como un único componente primitivo cuyas principales tareas son administrar los planes y ejecutar automáticamente los planes suscritos cuando se reciba una alarma.

Para recibir notificaciones desde el controlador de análisis se implementó la interfaz *AlarmListener*. A través de esta interfaz se reciben los *AlarmEvent* con el tipo de alarma emitida y el nombre de la regla que la emitió, para luego ejecutar todos aquellos planes que esten suscritos a las alarmas de dicha regla. Al momento de ejecutar un plan se provee de una

referencia al controlador de monitoreo y una al controlador de ejecución. El resultado final de la interacción entre estos controladores se muestra en la Figura 3.5

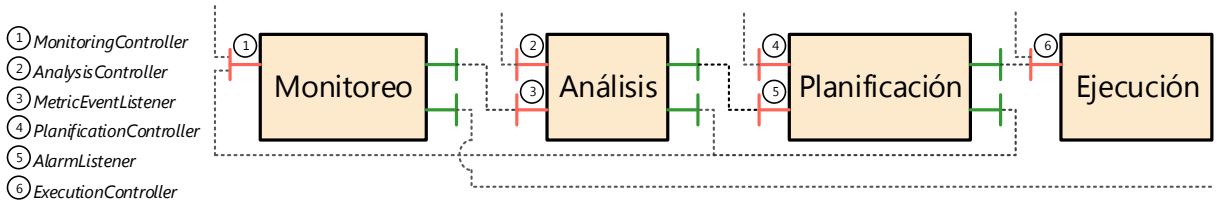


Figura 3.5: Los controladores MAPE completos

Capítulo 4

Administración desde GCMScript

En este capituló se presentará una extensión del lenguaje GCMScript para soportar e interactuar con los nuevos controladores autonómicos. El objetivo de esta parte de la memoria consiste en permitir una completa administración del comportamiento autonómico de los componentes a través de este lenguaje.

4.1. Extensión del modelo de GCMScript

GCMScript es actualmente la forma más sencilla y rápida de realizar reconfiguraciones sobre una aplicación GCM. Este lenguaje esta diseñado para navegar a través de la arquitectura de la aplicación y localizar elementos de interés basándose en sus propiedades o ubicación dentro de la arquitectura. Una vez localizados estos elementos es posible definir y aplicar complejas reconfiguraciones sobre ellos.

Internamente, este lenguaje utiliza un modelo para representar la arquitectura la aplicación. Como se muestra en la Figura 1.4, este modelo está compuesto por dos elementos, los *nodos* y los *ejes*. Los nodos representan entidades presentes en el modelo GCM, como componentes e interfaces, y se utilizan para indicar los objetivos de la reconfiguración. Los ejes representan la relación que existe entre los nodos y se utilizan para buscar y seleccionar los nodos deseados. Por ejemplo, en la Figura 4.1 se muestra como el nodo que representa al componente compuesto posee un eje llamado *child* que lo conecta con cada uno de los nodos de sus subcomponentes, a su vez, los nodos de los subcomponentes poseen un eje llamado *parent* que los conecta con el componente compuesto.

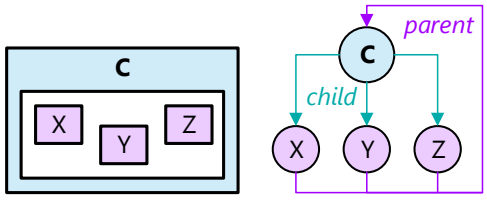


Figura 4.1: Ejemplo sencillo de la relación entre arquitectura y modelo de GCMScript

En la implementación de GCMScript, sin embargo, no existe una instancia persistente de

este modelo aplicado a toda la estructura de la aplicación, sino que se va creando a medida son requeridos durante la ejecución de algún comando de reconfiguración. Continuando con el ejemplo de la Figura 4.1, supongamos que tenemos una referencia al nodo del componente C en la variable `$c` y deseamos obtener una referencia al nodo del subcomponente X. Esto se logra con la expresión `$c/child::X`. Para resolver esta solicitud GCMScript llama al controlador *ContentController* del componente C, un controlador especial que administra el contenido de un componente compuesto. Se le solicita al *ContentController* una referencia de cada uno de los subcomponentes de C y crea un nodo por cada uno de ellos. Luego, de los nodos recién creados selecciona solo aquellos cuyo nombre sea igual a `X`, que por regla solo puede ser uno, retornando el o los nodos seleccionados. Durante este proceso, ninguno de los nodo generados es guardado y todo lo que no haya sido apuntado en una variable de GCMScript se perderá.

De la misma forma que se utiliza implícitamente el *ContentController* para obtener un subcomponente de un componente compuesto, se extendió GCMScript para utilizar implícitamente los controladores MAPE durante la gestión de los objetivos autonómicos. Se definieron tres nuevos nodos en el modelo para representar a las métricas, las reglas y los planes, y junto con ellos se definieron también los ejes que permitirán ubicar esos nodos dentro de la arquitectura. De esta forma, se hizo posible configurar el comportamiento autonómico de un componente sin tener que relacionarse directamente con los controladores MAPE.

4.1.1. Integración de métricas, reglas y planes

Para integrar las métricas, reglas y planes en el modelo de GCMScript se añadió un nodo nuevo para cada uno de estos elementos. Estos nodos estarán relacionados directamente con el nodo del componente administrado a través de ejes que llevarán el mismo nombre. El diagrama del modelo actualizado muestra en la Figura 4.2.

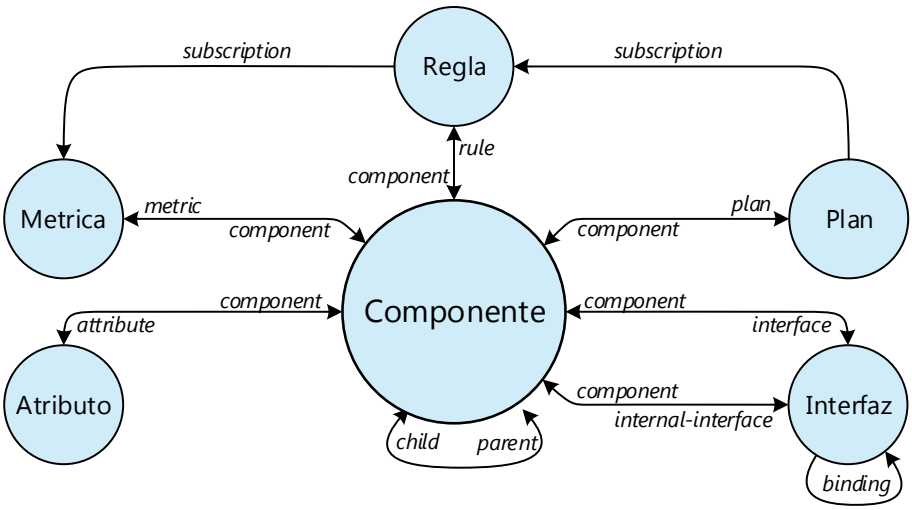


Figura 4.2: El Modelo de GCMScript extendido para soportar métricas, reglas y planes

Debido a que las métricas, reglas y planes no tienen valor por si mismos, si no que solo una vez que son cargados dentro de sus respectivos controladores, los nodos de este tipo fueron

construidos a partir del nombre con que fueron registrados y a una referencia al componente administrado. Estos dos recursos son suficientes para llamar al controlador autonómico pertinente y solicitar las acciones deseadas para el elemento representado.

Como se mostró en la sección 1.1.3 a todo nodo posee un conjunto de propiedades que pueden ser consultadas y, en algunos casos, modificadas. En la Tabla 4.1 se muestran las propiedades definidas para los nuevos nodos de métrica, regla y plan.

Nodo	**Propiedad**	Desripción	**Editable**
métrica	name	Nombre de la métrica	No
métrica	value	Último valor calculado	No
métrica	enable	Indica si las suscripciones están habilitadas	Sí
regla	name	Nombre de la regla	No
regla	alarm	Última alarma emitida	No
regla	enable	Indica si las suscripciones están habilitadas	Sí
plan	name	Nombre del plan	No
plan	enable	Indica si las suscripciones están habilitadas	Sí

Tabla 4.1: Tabla de propiedades para nodos de métrica, regla y plan.

Para obtener una referencia de estos nodos y consultar sus propiedades se deben utiliazar sus respectivos ejes. Por ejemplo, supongamos que deseamos obtener el valor de una métrica llamada `avgResponseTime`, ubicada en un componente referenciado por la variable de GCMScript `$foo`. El valor de la métrica es retornado por la siguiente expresión:

```
value($foo/metric::avgResponseTime);
```

Como se muestra en la Figura 4.3, el primer paso para la resolución de esta expresión consisteen obtener los todos los nodo de tipo métrica; la implementación del eje `metric` solicitará al componente `$foo` una referencia al *MonitoringController*, llamará al método `getRegisteredNames()` para obtener una lista con los nombres de todas las métricas cargadas en el controlador y creará un nodo de tipo métrica por cada uno de los nombres obtenidos.

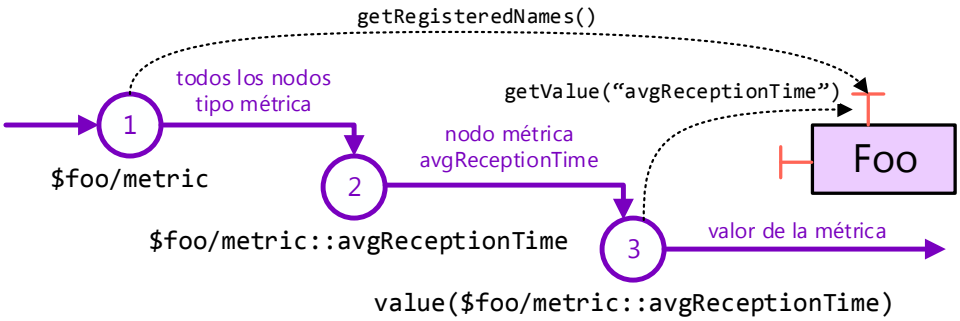


Figura 4.3: El Modelo de GCMScript extendido para soportar métricas, reglas y planes

El siguiente paso consiste en filtrar la métrica deseada; del conjunto de nodos métrica obtenidos se filtrarán todos aquellos con nombre igual a `avgResponseTime`, debido a que el nombre debe ser único se retornará una o ninguna métrica. Finalmente, el ultimo paso será consultar

el valor de la métrica; nuestro nodo llamara al controlador del monitoreo del componente y consultará el valor de la métrica que representa utilizando el método `getValue(metricName)`.

4.1.2. Suscripciones

Finalmente, se provee un eje adicional denominado `subscription` para representar las suscripciones entre nuestros elementos autonómicos. Desde planes hacia relgas y desde reglas hacia métricas, con este eje no solo podemos obtener una referencia a los elementos que estamos registrados, sino que también podemos también agregar o remover suscripciones. En la implementación se GCMScript existe un tipo especial de ejes denominados *modificables*, un eje modificable expone las acciones `add-` y `remove-` para implementar la adición y remoción de la relación que conecta al par de nodos involucrados. En este caso, las suscripciones se puede agregar y remover a gusto, por lo tanto se implementó este eje como un eje modificable.

Supongamos que contamos con un nodo de plan en la variable `$fooPlan`, a continuación se listará un ejemplo de de uso del eje suscripción:

```
-- Obtener todas las mtricas de las que depende indirectamente fooPlan
$planx/subscription::*/subscription::*;

-- Suscribir el plan fooPlan a una regla llamada "fooRule"
add-suscription($fooPlan/component::*/rule::fooRule);
```

Listado 4.1: Ejemplo de uso del eje `subscription`

Al igual que en el ejemplo de la Figura 4.3, para resolver todas estas solicitudes la implementación de `subscription` y la de todos los elementos presentados en esta sección utilizan los controladores autonómicos implementados en el capitulo 3.

4.2. Una API para los objetivos administrativos

Hasta este punto hemos integrado en el modelo de GCMScript a nuestros objetivos administrativos. Esta extensión por si sola nos permite explorar la configuración autonómica actual y modificar las subscripciones de reglas y planes, pero para para permitir una completa administración de estos objetivos se necesitaron algunas funciónes y acciónes adicionales.

Se debe tener presente que la finalidad de GCMscript es modificar la configuración un recurso y no intervenir en su ejecución. Por lo tanto, acciones como calcular una métrica o forzar la ejecución de un plan fueron dejados deliberadamente sin soporte.

Los siguientes grupos de funciones y acciones fueron implementados de forma nativa en GCMScript:

Suscripciones en las métricas

En la extensión del modelo no se pudo incluir las suscripciones de las métricas, en su lugar se implementó esta una acción independiente denominada `subscribe-metric` que recibe como parámetro el nodo de la métrica en cuestión y el nombre del *RecordEvent* a suscribir.

```
subscribe-metric($metricNode, "REQUEST_SENT");
unsubscribe-metric($metricNode, "REQUEST_SENT");
is-subscribed-metric($metricNode, "REQUEST_SENT");
```

Listado 4.2: Ejemplo: subscripción de métricas

Para mantener una similitud en la sintaxis de la API añadieron acciónes con el mismo formato de nombres para las suscripciones reglas y planes, donde en ves de `metric` se usó `rule` y `plan` respectivamente. Estas acciónes fueron implementadas a través de una librería de GCMScript, donde internamente se llamó a la acción nativa `add-subscription`.

Añadiendo y removiendo elementos

Se añadieron también acciones para agregar y remover elementos a los controladores MAPE. Para agregar un elemento se necesitó una referencia al componente de destino y el nombre de clase que implementa el elemento en cuestión. Gracias que los ejes `metric`, `rule` y `plan` no son modificables los nombres de acciónes como `add-metric` y `remove-plan` estan disponibles y se utilizaron para denominar estas acciones.

```
add-rule($ruleNode, "cl.examples.rules.MyRule");
remove-rule($ruleNode);
```

Listado 4.3: Ejemplo: añadiendo/removiedo métricas

Obteniendo información

Finalmente, se añadieron algunas funcions de utilidad para simplificar la, tal vez más recurrente, tarea de estudiar el estado del sistema. La idea es resumir la configuración actual imprimiendo una tabla informativa, el contenido mostrado consiste en el nombre del elemento, su valor actual (métricas y reglas), si esta habilitada o no, y finalmente una listado de sus suscripciones.

```
print-metrics(\$fooMetric);
print-rules(\$fooRule);
print-plans(\$fooPlan);
```

4.3. Una nueva consola interactiva

Actualmente existe una consola interactiva para FScript, el predecesor de GCMScript para el modelo de componentes Fractal. Esta consola se conecta a un interprete de comandos de FScript para poder interactuar y reconfigurar una aplicación Fractal.

En este trabajo contamos con los controladores de ejecución, que en su interior contienen su propio interprete de GCMScript. Probablemente, distintos controladores de ejecución posean distintas librerías de reconfiguración cargadas. Para aprovechar estos recursos, se implementó una nueva consola interactiva para GCMScript capaz de conectarse e interactuar con el controlador de ejecución de un componente, como se muestra en la Figura 4.4.

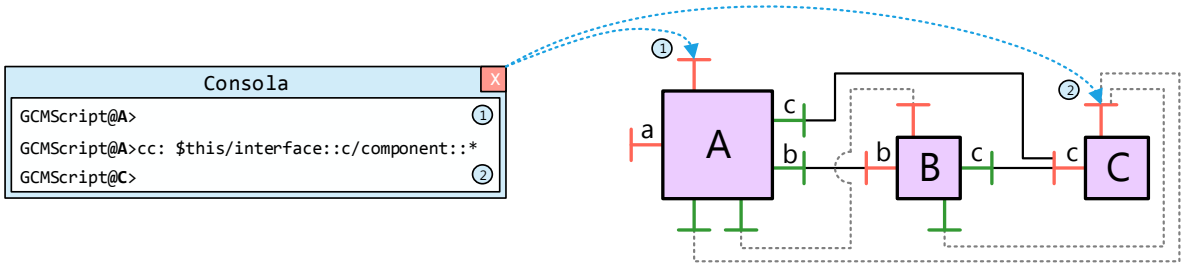


Figura 4.4: Consola interactiva para aplicaciones GCM

Los componentes de GCM/ProActive son expuesto en la red a través de una dirección RMI. Esta dirección puede ser pasada como parámetro para la nueva consola para que pueda encontrar al componente e iniciar una sesión utilizando su controlador de ejecución.

El componente con el que se interactúa puede ser cambiado durante después de haber sido iniciada la consola, esta vez, ayudándonos de GCMScript. Se implementa para la consola un comando denominado *change componente* (`:cc`) que recibe como parámetro un nodo de componente del modelo de GCMScript. En el ejemplo de la Figura 4.4 comenzamos con una consola iniciada en el componente A, en este punto se utiliza el interprete de A quien utiliza la variable `$this` para referenciar al componente A. Luego utilizamos el comando `:cc` para cambiar la consola de posición hacia el componente C, el argumento es una expresion de GCMScript que será resuelta por el interprete de A antes de mudar la consola de componente.

La finalidad de esta consola es facilitar el acceso a la API para la administración de los objetivos administrativos implementada en la sección anterior. De esta forma, modificar la configuración autonómica del sistema se simplifica a la tarea de abrir una consola y ejecutar algunos comandos.

Capítulo 5

Experimentación

En este capítulo se realizarán experimentos para probar la eficacia de los recursos implementados. En la sección 5.1 se mostrará un problema de rendimiento detectado en las consultas a valores de métricas remotas, se estudiará el caso, se presentará una solución y se realizarán las pruebas correspondientes. Luego, en la sección se presentarán situaciones problemáticas que pueden ser mejoradas utilizando computación autonómica, se indicará como se utilizó el trabajo de esta memoria para resolver estos problemas y mostrarán los resultados obtenidos.

5.1. Optimización del monitoreo remoto

Luego se realizar algunas pruebas de monitoreo se detectaron problemas de rendimiento en los casos donde se solicita información a controladores de monitoreo remotos. Si el componente remoto que se desea consultar sufre de una alta demanda, entonces existe una alta probabilidad de obtener el valor de la métrica remota con algo de retraso. Este retraso es suficientemente significativo como para que el valor de la métrica quede obsoleto, experimentalmente también se comprobó que este retraso se puede acumular y generar valores inválidos e inesperados.

En esta sección se estudiará este caso, se propondrá una solución, y se experimentará para demostrar la efectividad de la solución propuesta.

5.1.1. Problemas con el retraso en el monitoreo remoto

En la implementación de GCM, todo componente posee una cola donde se almacenan las solicitudes recibidas. Estas solicitudes se van retirando para ser servidas de forma serial y en orden FIFO, como se muestra en la Figura 5.1. Las solicitudes a controladores o subcomponentes no son una excepción, aunque el destino sea distinto, deben esperar su turno en la cola como todas las demás.

Al solicitar el valor de una métricas remota necesariamente tendremos que esperar por nuestro turno en la cola de cada uno de los componentes intermediarios. Por ejemplo, continuando con la Figura 5.1, cuando el componente A solicite el valor de una métrica en D deberá

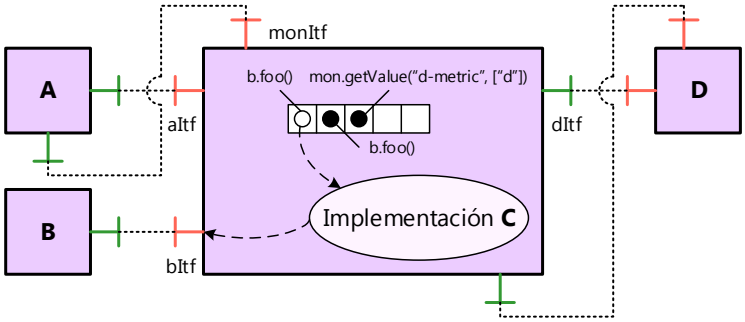


Figura 5.1: Encolado de una solicitud al controlador de monitoreo.

esperar que C se desocupe antes de poder comunicarnos con D. Esta situación puede causar dos importantes problemas en el proceso de monitoreo:

1.- *Una larga espera puede generar valores obsoletos y/o invalidos.*
Supongamos que tenemos las métricas m, p, y q tal que m depende de p y de q. Si p y q corresponden a métricas remotas, una espera demasiado larga podría causar que sus valores lleguen con una diferencia de tiempo significativa. Durante este tiempo los valores reales podrían haber sido actualizado varias veces, generando todo tipo de valores inesperados para m.

2.- *Durante la espera los controladores de monitoreo se bloquean.*
Los controladores de monitoreo son tambien componentes y poseen su propia cola de solicitudes. Mientras estos controladores esperan, otras métricas registradas a eventos *RecordEvent* no podrán ser recalculadas, retrasando todo el sistema de monitoreo.

Supongamos ahora que el la frecuencia con que A y B envían nuevas solicitudes a C es mayor a la frecuencia con que C resuelve dichas solicitudes. Mientras estas condición se mantenga el tamaño de la cola de solicitudes de C crecerá con el tiempo, esto causará un aumento constante en el retraso con que llegarán los valores de las métricas de D a A y, por lo tanto, un empeoramiento constante en las condiciones de monitoreo causadas por los problemas (1) y (2) descritos anteriormente. Esta situación no supone un problema para la parte funcional la aplicación, su comportamiento es el esperado, sin embargo, para el monitoreo significa una perdida total de su representatividad.

El problema comienza por que ambos controladores de monitoreo A y C no atenderán otra solicitud hasta obtener la respuesta de C, como se muestra en la Figura 5.2:

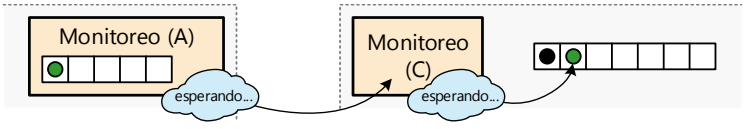


Figura 5.2: Los controladores de monitoreo se bloquean esperando.

Mientras esto sucede, A y B continuan enviando solicitudes funcionales a C, las que tomas su posición en la cola. El controlador de A por su parte también comienza a acumular tareas pendientes, como se observa en la Figura 5.3:

Como consecuencia, la próxima consulta que haga el controlador de monitoreo de A deberá

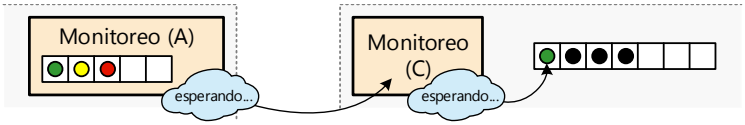


Figura 5.3: Acumulación de solicitudes pendientes.

esperar un tiempo mayor al anterrior, retrasando el monitoreo tanto en A como en C, y empeorándo la situación en cada iteración, como se muestra en la Figura 5.4.

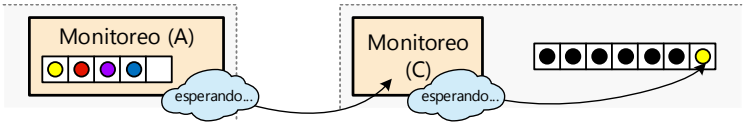


Figura 5.4: Las condiciones empeoran para la siguiente interación.

5.1.2. Solución y Experimentación

Para solucionar este problema se permitió que las solicitudes realizadas a los métodos `getValue()` del controlador de monitoreo sean ejecutadas inmediatamente y en paralelo con cualquier solicitud que se este atendiendo. Esta condición fue impuesta en la interfaz no funcional del controlador de monitoreo, en el *MonitoringManager* y en el *MetricStore*, permitiendo evadir así los retrasos en la consulta de valores a métricas remotas en todos sus niveles.

El método `getValue()` de las métricas esta pensado para retornar el último valor calculado de la métrica. Por lo tanto, el tiempo de respuesta de este método es despreciable y su ejecución no influye de manera alguna en otros asuntos relacionados con el monitoreo.

Para implementar esta solución reutilizó un recurso heredado de los objetos activos de la librería ProActive denominado *immediate service*. Los servicios inmediatos nos permiten imponer que toda solicitud realizada a un método especifico deberá ser ejecutada inmediatamente y en paralelo con cualquier otra solicitud que se este ejecutando en ese momento.

Caso de prueba

Para probar esta solución se construyó una aplicación para crackear contraseñas cifradas utilizando MD5 utilizando fuerza bruta. Como se muestra en la Figura 5.5, la aplicación cuenta con un componente *Cracker* encargado de crackear las contraseñas, un componente *Store* donde se almacenan los resultados, y un componente compuesto *CrackingService* que modulariza todo lo anterior:

Este sistema fué monitoreado para calcular el número de solicitudes recibidas pendientes por servir. Para esto se añadió una métrica *ReceivedCounter* en el componente *Cracker* para contar el número de solicitudes recibidas, una métrica *ServedCounter* en el componente *Store* para contar las solicitudes servidas, y una métrica en *PendingRequests* en el componente *CrackingService* que consultará el valor de *ReceivedCounter* y le restará el valor de *ServedCounter*.

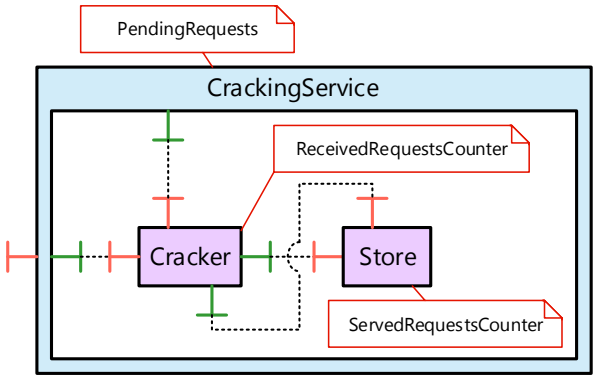


Figura 5.5: Experimento

El método utilizado para obtener las solicitudes pendientes es deliberadamente ineficiente con el fin de estresar el proceso de monitoreo. Junto con esto, se fuerza también la condición de que frecuencia con que el sistema recibe solicitudes sea mayor a la frecuencia con la que las resuelve, para esto se envió cada 2 segundos una solicitud de crackeo que, medido experimentalmente, toma 2,6 segundos en ser resuelta.

Resultados

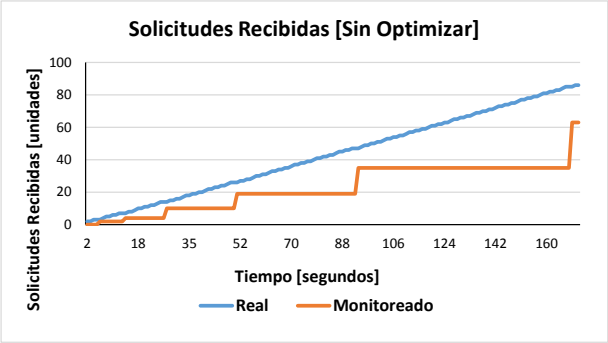
En la Figura 5.6 se muestra el resultado de una ejecución normal, sin los *immediate services*. El gráfico de la izquierda (5.6a) muestra el número de solicitudes recibidas en el componente *Cracker* tal como son percibidas desde la métrica *PendingRequests*. Se puede observar como la frecuencia con que *PendingRequest* obtiene el valor de *ReceivedRequestsCounter* va disminuyendo exponencialmente a medida que pasa el tiempo. Adicionalmente, se puede observar como el valor de *ReceivedCounter* se va alejando cada vez más de la realidad. Esto se produce debido a que las solicitudes de *PendingRequest* bloquean al controlador de monitoreo de *Cracker*, retrasando el cálculo de *ReceivedCounter*.

El gráfico de la derecha (5.6b) muestra el número de solicitudes pendientes percibidas por *PendingRequests*. Se puede observar como retraso en el cálculo de *ReceivedCounter* afectó negativamente el resultado de *PendingRequests*; un valor de las solicitudes recibidas desactualizado se ve igualado por el valor de las solicitudes servidas actualizado, obteniendo una diferencia constantemente cercana a cero.

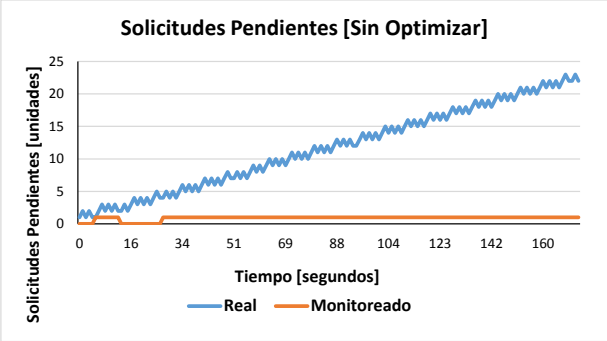
Finalmente, en la Figura 5.7 se muestra el resultado de una ejecución utilizando los *immediate services*. En estos graficos se puede ver como los servicios inmediatos evitaron todos los problemas de congestión en la comunicación, permitiendo un monitoreo eficiente que refleja la realidad.

5.2. Caso de uso: crackeador distribuido autonómico

Para probar la efectividad de los componentes autoadaptables implementados a lo largo de esta de memoria se extendió el crackeador de contraseñas de la sección anterior para funcio-

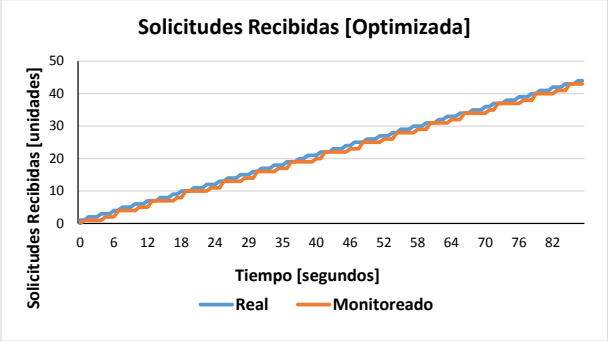


(a) Solicitudes recibidas percibidas desde *PendingRequests*.

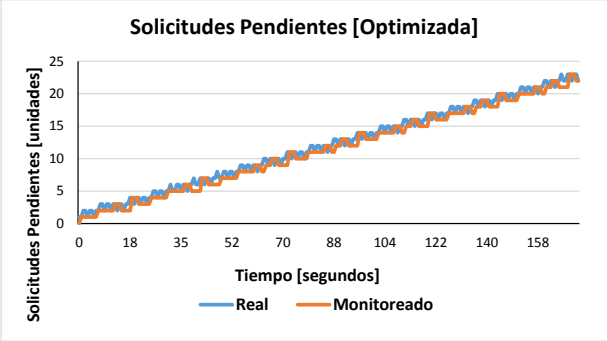


(b) Número de solicitudes pendientes.

Figura 5.6: Resultado del experimento sin utilizar *immediate services*.



(a) Solicitudes recibidas percibidas desde *PendingRequests*.



(b) Solicitudes pendientes calculadas por *PendingRequests*.

Figura 5.7: Resultado del experimento utilizando *immediate services*.

nar de forma distribuida. En esta sección se identificarán situaciones problematicas donde este crackeador distribuido necesitaria ser reconfigurado y luego se mostrará como utilizar métricas, reglas y planes para definir el comportmaiento autonómico que realizaría esas reconfiguraciones automaticamente. Finalmente, se realizarán los experimientos pertinentes que mostrarán la efectivadad de la soluciones propuestas.

Arquitectura del crackeador

La nueva versión del crackeador se pude observar en la Figura 5.8, nuevamente la aplicación se expone como único componente compuesto, esta vez, denominado *Cracker*. Las solicitudes de crackeo son recibidas internamente por el componente *Dispatcher*, cuyo rol consiste en dividir el trabajo en tres tareas T_1, T_2, y T_3. Cada una de estas tareas representa un conjunto de palabras que deberán ser probadas hasta dar con la contraseña buscada. Por ejemplo, considerando una contraseña de largo máximo 6 y un alfabeto de 36 caracteres, el número total de palabras a probar son $W = \sum_{i=0}^{6} 36^i$, entones el *Dispatcher* asignará las primeras

$W/3$ palabras a T_1, las segundas $W/3$ a T_2 y el resto a T_3.

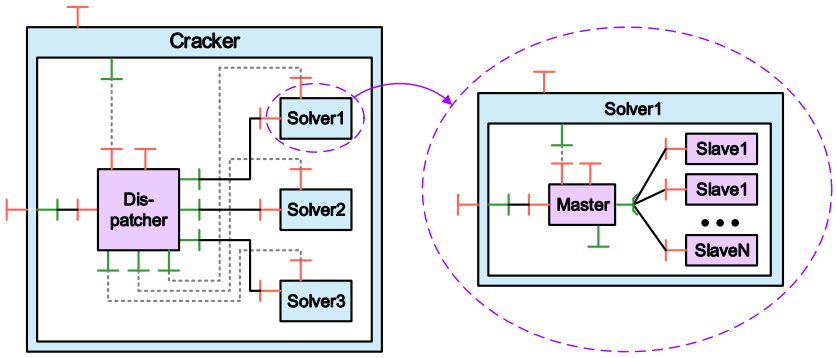


Figura 5.8: Crackeador de contraseñas distribuido.

Cada una de las tareas generadas por el *Dispatcher* es enviada a su *Solver* distinto para ser calculadas. Los *Solvers* son definidos como como un componente compuesto encargado de modularizar al subsistema responsable de buscar la contraseña. Dentro de un *Solver*, la tarea T_i es recibida por el componente *Master*. Este componente subdividirá la tarea en las subtareas $t_1^i, t_2^i, ..., t_N^i$, donde N es el número de Slaves disponibles. Finalmente, los *Slaves* son quienes realmente buscan la contraseña probando como posible contraseña el conjunto de palabras t_j^i asignado.

Configuración

Este diseño esta pensado para desplegar cada *Solver*, y todos sus sub-componentes, en una maquina distinta. Un despliegue de esta forma pretende aprovechar al máximo los recursos disponibles en dos sentidos; primero, es posible añadir tantos *Slaves* como unidades de procesamiento tenga la maquina, segundo, es posible asignar tareas de tamaños distintas para maquinas con capacidades distintas.

Para configurar el número de *Slaves* se provee un *script* de GCMScript pre-cargado en el controlador de ejecución de los *Solvers*. El *script* provee los métodos `add-slave()` y `remove-slave()` para añadir y remover *Slaves* respectivamente. Estos métodos pueden ser invocados a través la consola interactiva presentada la sección 4.3; posicionando la consola directamente sobre el *Solver* en cuestión, con el comando `:cc`, o enviando una solicitud de ejecución remota a través del comando `remote-execute()` como se mostró en la sección 2.2.3.

Para configurar el tamaño de la tarea que se enviará a cada *Solver* se utiliza el *AttributeController* para exponer en componente *Dispatcher* un atributo especial denominado *DistributionPoint*. Este atributo consiste en un punto (x, y) en el dominio $0 \leq x \leq y \leq 1$ que define una partición del número total de palabras a probar en el sistema, como se muestra en la Figura 5.9:

El punto de distribución puede ser modificado llamando directamente al *AttributeController*, o bien a utilizando de GCMscript.

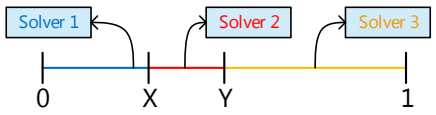


Figura 5.9: Punto de distribución *DistributionPoint*.

5.2.1. Asignación autonómica de recursos

Supongamos que necesitamos satisfacer alguna política de calidad de servicio, por ejemplo, que el tiempo de respuesta del *Cracker* sea menor o igual un segundo. Este problema se puede resolver rapidamente modificando manualmente el número de *Slaves* hasta satisfacer las demandas de rendimiento impuestas, sin embargo, nada nos asegura que esta perdurara en el tiempo; cambios impredecibles en el ambiente podrían invalidar la configuración actual.

Para una solución autonómica se definen los siguientes elementos: una métrica *AvgResponseTime* para monitorear los tiempos de respuesta del componente *Dispatcher* y de los *Solvers*, una regla *MaxResponseTime* para verificar que el tiempo de respuesta de *Dispatcher* no sobrepase el maximo especificado, y un plan *ResourcesRequest* para solicitar más recursos a los *Solvers*. Estos elementos se muestran en la Figura 5.10:

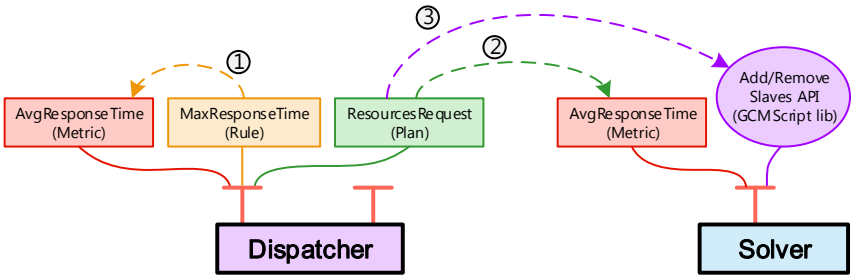


Figura 5.10: Configuración para la asignación autonomica de recursos.

Cada vez que que la métrica *AvgResponseTime* del *Dispatcher* sea recalculada la regla *MaxRequestTime* verificará que su valor no exceda el valor exceda el máximo especificado (1), de de no cumplirse esta regla se emitirá una alarma que accionará la ejecución del plan *ResourcesRequest*. Este plan consultará el valor de la métrica *AvgResponseTime* de cada uno de los *Solvers* (2), esta información será utilizada para escoger al *Solver* con peor rendimiento. Finalmente, este plan enviará a través del controlador de ejecución los comandos necesarios para solicitar que un nuevo *Slave* sea agregado en el *Solver* escogido (3). Después de cada adición exitosa el plan *ResourcesRequest* ingorará cualquier alarma recibida desde la regla *MaxRequestTime* para permitir que los cambios surjan efecto.

NOTA: La implementación de estos elementos se pueden observar en el capítulo A.

Experimento

Para probar esta solución autonómica se desplegó una instancia del crackeador utilizando un solo *Slave* por *Solver*. A cada *Solver* se asignó un tercio del trabajo durante todo el expe-

rimento (punto de distribución $(\frac{1}{3}, \frac{2}{3})$). A esta instancia se le conectó un cliente que envía constantemente solicitudes de creackeo, cliente que espera recibir una respuesta antes de volver enviar una nueva solicitud. Las solicitudes enviadas corresponden a contraseñas cifradas previamente preparadas para tardar aproximadamente 4,5 segundos en ser crackeadas. Desde este punto, se realizarán las siguientes actividades:

1.- Solicitar un tiempo de respuesta no mayor a 1 segundo.

2.- Cambiar la solicitud a un tiempo de respuesta no menor a 3 segundos.

Estas solicitudes fueron concretadas utilizando la consola interactiva de GCMScript para iniciar los elementos definidos anteriormente. En el caso de la solicitud (2) se utilizó un plan y una regla homologos a *MaxRequestTime* y *ResourcesRequest* pero en el sentido contrario; remueven *Slaves* desde *Solver* con mejor rendimiento.

Finalmente, el despliegue de la aplicación se realizó sobre el siguiente conjunto de maquinas:

0) *Cracker* y *Dispatcher*: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
1) *Solver 1* Intel(R) Xeon(R) CPU X5647 @ 2.93GHz
2) *Solver 2* Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz
3) *Solver 3* Intel(R) Xeon(R) CPU E5-2403 0 @ 1.80GHz

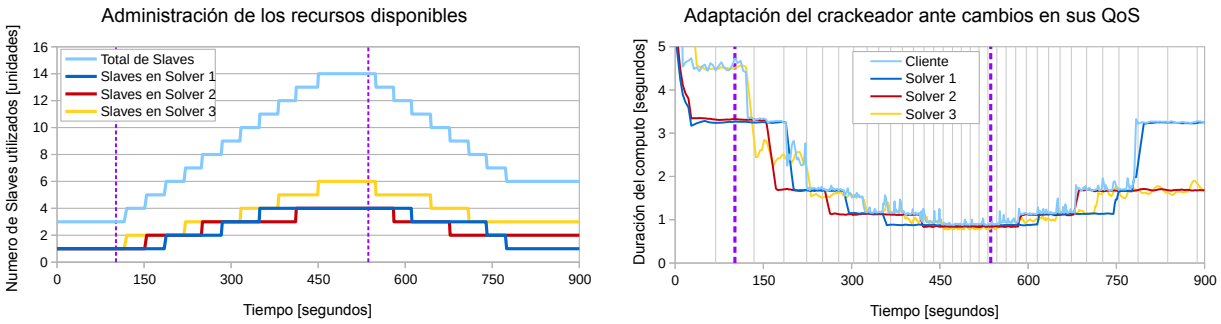
Resultados

En el gráfico de la Figura 5.11a se puede como el experimento inició con solo un *Slave* por cada *Solver*. Luego, después de solicitar un tiempo de respuesta menor o igual a un segundo el número de *Solvers* comenzó a crecer. En cada iteración se seleccionó el *Solver* con peor rendimiento, y el objetivo final se alcanzó utilizando 6 *Slaves* en el *Solver 3* y solo 4 en el resto, lo que indica una debilidad de esa maquina con respecto a las otras dos. Alcanzado este punto cambió la solicitud para exigir que los tiempos de respuesta fueran como minimo 3 segundos. Esto provocó una reducción en el número de *Solvers* hasta cumplir con el objetivo, termiando con 3 *Slaves* en el *Solver 1*, lo que ratifica la debilidad de esta maquina con respecto a las demás.

En el gráfico de la Figura 5.11b se muestran los tiempos de respuesta obtenidos a lo largo del experimento. En todo momento el tiempo de respuesta percibido por el cliente lo impone el *Solver* con peor rendimiento. Es importante notar como al finalizar el experimento termina siendo el *Solver 1* quien impone el tiempo de respuesta del crackeador, siendo esta la maquina con mejor rendimieto. La diferencia final en los rendimiendo de los *Solvers* indica un claro derroche de recursos en los *Solvers 1* y *2*, situación que puede ser mejorada con una modificacion en el tamaño de las tareas enviadas a cada *Solver*.

5.2.2. Balanceo autonómico de carga

En el problema anterior pudimos observar como la distribución constante del trabajo en partes iguales se tradujo en una medición poco precisa de los rendimentos de la maquina, lo que llevó al sistema de asignación autonómica de recursos a tomar una decisión poco



(a) (b)

Figura 5.11: Resultados de la asignación de recursos autonómica.

óptima. En general, toda distribución del trabajo que no sea proporcional al rendimiento de los *Solvers* instroducirá un derroche de recursos de computo. Para solucionar este problema es necesario asignar el *DistributionPoint* optimo en el componente *Dispatcher*, sin embargo, esta no es una tarea fácil. Este punto puede variar muy rapidamente y es muy sensible a los eventos de su ambiente; imprevistos como cuellos de botella en la red, adición y remoción de *Solvers*, o la utilización de las maquinas por parte de terceros son algunos de los problemas que podrían dejar obsoleto el punto de distribución calculado.

Para una solución autonómica a este problema se definen los siguientes elementos: una métrica *AvgResponseTime* para calcular los tiempos de respuesta de los *Solvers*, una métrica *OptimalBalance* para calcular el punto de distribución optimo, una regla *BalanceStatus* para verificar que el punto de distribución no haya variado demasiado, y finalmente un plan *BalanceUpdate* para actualizar punto de distribución. Estos elementos se muestran en la Figura 5.12.

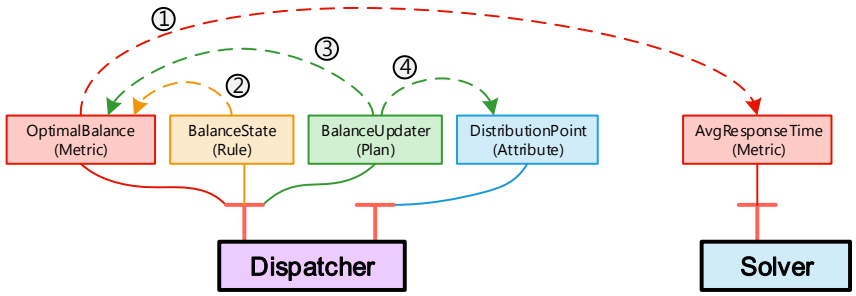


Figura 5.12: Balancer

Cada vez que la métrica *OptimalBalance* sea recalculada consultará los valores de las métricas *AvgResponseTime* de los *Solvers* (1) y utilizará sus valores para calcular el punto de distribución óptimo en ese instante. Luego, la regla *BalanceStatus* consultará el valor de *OptimalBalance* (2) y verificará su valor no haya variado en una distancia mayor a d con respecto al ultimo punto de distribución. Si la distancia entre estos puntos es mayor a d, entonces se considera que la diferencia en los rendimientos es importante y amerita una actualización del punto de distribución, emitiendo una alarma. Esta alarma indicará al plan *BalanceUpdate* que deberá tomar el valor de *OptimalBalance* (3) y asignárselo al *Dispatcher* (4).

Experimento

Para probar este balanceador autonómico de las cargas asignadas a cada *Solver* se desplegó una instancia del crackeador y de su cliente exactamente de la misma forma como se hizo en el experimento anterior (5.2.1). La única diferencia fueron son los elementos autónomicos presentes.

Para observar el comportamiento de este balanceador se dividió el experimento en 5 fases, cada una de una duración de 180 segundos de ejecución:

- **Fase 1. No auto-optimizado:**
  La ejecución se realizó sin ningún tipo de balanceo autonómico. La distribución de cargas fué la distribución por defecto: $|T_1| = |T_2| = |T_3|$.
- **Fase 2. Auto-balanceado:**
  El balanceo automático de cargas activado.
- **Fase 3. Cambio en el ambiente (1):**
  Se añadieron 2 *Slaves* al *Solver3*.
- **Fase 4. Cambio en el ambiente (2):**
  Se añadieron 3 *Slaves* más al *Solver3* y 2 al *Solver2*
- **Fase 5. Cambio en el ambiente (3):**
  Se removieron 5 *Slaves* desde el *Solver3*, se removieron 2 *Slaves* desde el *Solver2* y se añadieron 5 *Slaves* al *Solver1*.

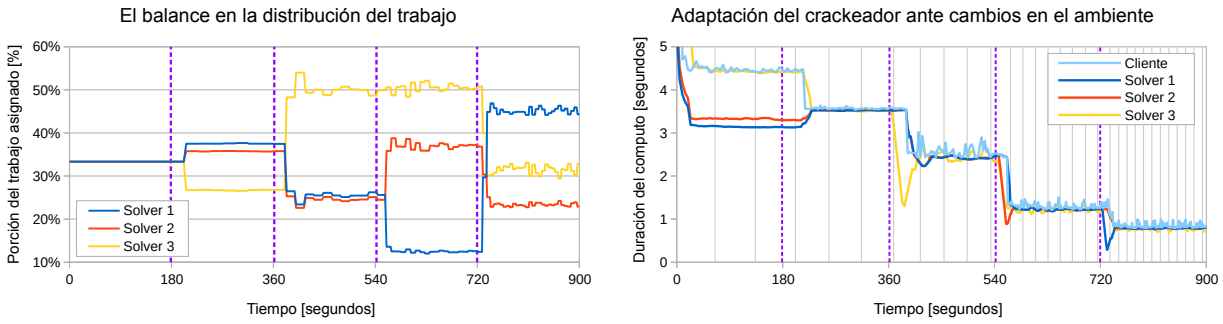
Estos cambios fueron realizados de forma manual, utilizando la consola interactica de reconfiguraciones.

Resultados

En La Figura 5.13a se puede observar como rápidamente se redistribuyeron las cargas de trabajo asignado a cada *Solver*. En paralelo, en la Figura 5.13b se puede observar el impacto que tuvo estas nuevas asignaciones en los tiempos de ejecución.

Inicialmente, durante la primera fase el tiempo percibido por el cliente fue impuesto por el *Solver 3*, por ser el *Solver* con peor rendimiento. Una vez activado el balanceo autonómico, se disminuyó la carga para el *Solver 3* pero se aumentó la de *Solver 1*, lo que equiparó los tiempos de respuestas de los *Solver*, minimizando el tiempo de respuesta percibido por el cliente.

En las siguientes fases se puede observar como incialmente se produce un fuerte impacto en los tiempo de ejecución. En la transicion de la fase 2 a la fase 3 el *Solver 1* pasa de contar con 3 *Solvers* a contar con 6, esto se traduce en una brusca disminución en su tiempo de respuesta. Este fenomeno es rapidamente reparado por el balanceador, quien le otorga casi el doble de carga que tenía anteriormente, nivelando sus tiempos de respuesta con el de los demás *Solvers*.



(a) (b)

Figura 5.13: Resultados del balanceo autonómico de carga.

5.2.3. Crackeador autonómico

Finalmente, para finalizar con este experimento, se mesclaron ambas soluciones para proveer un crackeador capaz de aumentar o disminuir su capacidad de carga de forma balanceada. Para combinar estas soluciones fue necesario reemplazar el criterio de selección de *Solvers* por parte del plan *ResourcesRequest*; para seleccionar al *Solver* con peor rendimiento se consultará a la métrica *OptimalBalance* y se escogerá aquel *Solver* con menor porcentaje de carga asignado.

Las prueba realizada fue la misma que se realizó en la sección **??** y bajo las mismas condiciones. En primer lugar se solicitó un tiempo de respuesta menor o igual a uno, luego, cumplido este objetivo, se exigió un tiempo de respuesta de no menos de 3 segundos.

Resultados

En la Figura 5.14 se puede observar como los tiempos de respuestas de los solvers fueron forzados igualarse después de cada adición o remoción de un *Slave*, alcansando rapidamente una estabilidad.

La combinación de ambas características introdujo eficiencia en la utilización de recursos. En la Figura 5.15b se puede observar como finalmente fueron eliminados todos los *Slaves* para cumplir con la condición de tener un tiempo de respuesta de no menos de tres segundos. Es decir, se disminuyó la cantidad de recursos utilizados con respecto al experimento de la sección 5.2.1 mientras cumplian con los mismos objetivos.

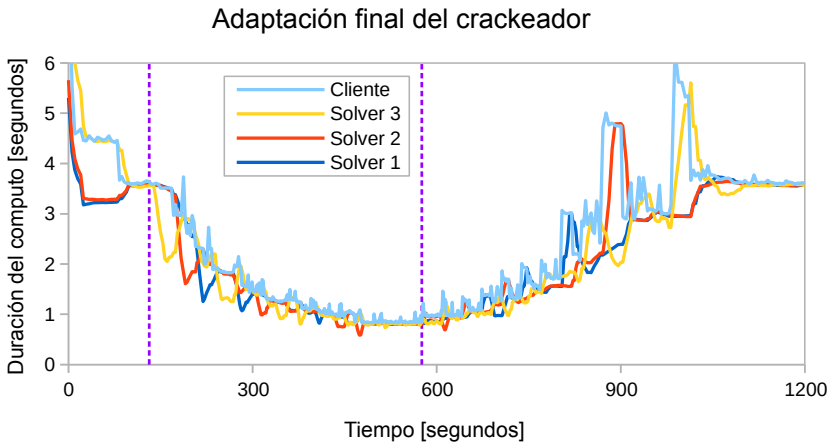
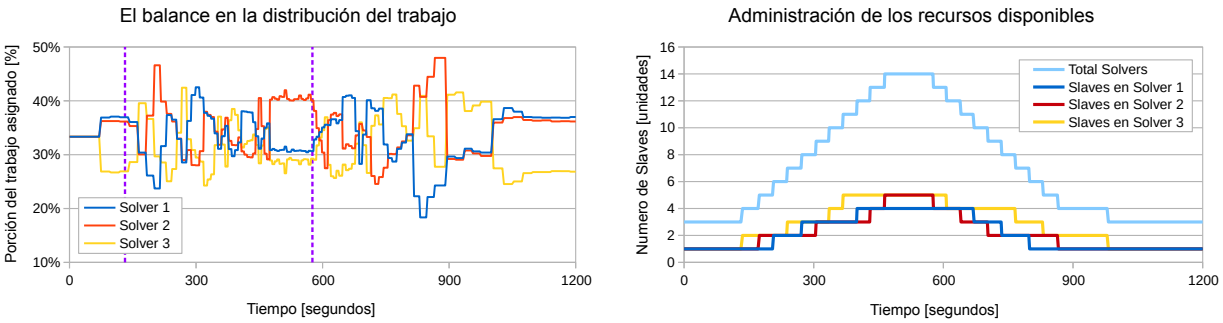


Figura 5.14: Comportamiento adaptativo del crackeador ante cambios en los QoS



(a) (b)

Figura 5.15: Procesos adaptativos por separado.

Conclusiones

En este trabajo de memoria se retomó la implementación de un *framework* para la programación de componentes auto-adaptables bajo el contexto del modelo de componentes distribuidos GCM. Este *framework* se basa en las ideas de la computación autonómica para integrar en en la implementación de los componentes el bucle de control autonómico MAPE. Cada una de las fases del bucle debieron ser implementadas a través de un componente controlador distinto.

En este framework, para otorgar un comportamiento auto-adaptable en un componente es requerida la definición de conjunto de objetivos administrativos para orientar al bucle de control autonómico MAPE en su tarea de mantener al sistema en el estado deseado. Para facilitar la manipulación de este comportamiento auto-adaptable se propuso una API que facilita la definición de estos objetivos administrativos. La solución propuesta tomó la forma de tres elementos distintos (*Métricas*, *Reglas* y *Planes*) que en su conjunto definen completamente el comportamiento auto-adaptable del componente.

El bucle de control autonómico diseñado para este framework inicia sus actividades con la recolección de eventos generados durante la ejecución del componente administrado. La información contenida en estos eventos representa el trafico de solicitudes que experimenta el componente. Esta información es traducida a una representación interna, denominada *registros*, y es almacenada para servir de base para la medición y análisis del estado del sistema. En la implementación realizada durante esta memoria se identificó con mayor precisión la información representada por estos registros. Para esto, se introdujo un nuevo tipo de registro para diferencias aquellas solicitudes que no esperan una respuesta del servidor, luego, se propuso un nuevo conjunto de eventos para notificar con mayor exactitud las actualizaciones realizadas a estos registros.

Para enriquecer el conocimiento sobre el estado del sistema, obtenido durante fase de monitoreo, este *framework* introduce el concepto de monitoreo remoto: el controlador de monitoreo es capaz de consultar el valor de las métricas pertenecientes a controladores de monitoreo de otros componentes. Es este trabajo se implementó dicha característica, habilitando este tipo de consultas a través de la API propuesta. De esta forma, se permitió que una métrica pueda basar su valor en el valor de otras métricas pertenecientes a otros componentes del sistema. Posteriormente, se estudió el impacto de estas consultas remotas en el proceso de monitoreo, se identificaron situaciones conflictivas bajo ciertas condiciones de estrés y, finalmente, se propuso una solución mas la debida experimentación que demuestra su efectividad.

Volviendo al bucle de control autonómico, una vez obtenidos los antecedentes suficientes para validar una reconfiguración sobre el sistema, los cambios son aplicados a través de GCMS-

cript. GCMScript provee un modelo para manipular a un alto nivel los elementos de GCM, permitiendo acceder fácilmente a las funcionalidades ofrecidas por las interfaces de configuración. Con el propósito de permitir una manipulación simple, rápida y eficaz de la API propuesta (*Métricas*, *Reglas* y *Planes*) se extendió GCMScript para soportar en su modelo a los controladores MAPE. De esta forma, se hizo posible modificar el comportamiento autonómico de los componentes en tiempo de ejecución a través de GCMScript. Como herramienta de utilidad practica, se implementó además una consola interactiva capaz de conectarse directamente con un componente, comunicándose en lenguaje GCMScript a través del controlador de ejecución de dicho componente.

Finalmente, todas estas herramientas fueron puestas a prueba en la implementación de un crackeador de contraseñas distribuidos. Se mostró como a través del uso de métricas, reglas y planes fue posible proveer un comportamiento auto-adaptable a este crackeador en dos sentidos; en la capacidad de reconfigurarse autonómicamente para distribuir su trabajo en las proporciones óptimas cada vez que cambiaban las condiciones en su ambiente distribuido, y en la capacidad de modificar su arquitectura autonómicamente para cumplir con la calidad de servicio exigidos.

Bibliografía

[1] Oasis. reference model for service oriented architecture v1.0. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`.

[2] M Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE: Web Information Systems Engineering, 2003*, volume 3, pages 3–12, 2003.

[3] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[4] A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX Conference on System Administration*, LISA '02, pages 185–188, Berkeley, CA, USA, 2002. USENIX Association.

[5] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[6] Cristian Ruz. *Autonomic monitoring and management of component-based services*. PhD thesis, Université de Nice, Sophia Antipolis, 2011.

[7] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm: A grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1):5–24, 2009.

[8] European telecommunications standards institute, technical specification. ts 102 830: Grid; grid component model (gcm); gcm fractal management api.

[9] Bruneton e, coupaye t, stefani j. the fractal component model specification, 2004. `http://fractal.objectweb.org/specification/index.html/`.

[10] The proactive parallel suite. `http://proactive.inria.fr/`.

[11] R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[12] Pierre-Charles David, Thomas Ledoux, Thierry Coupaye, and Marc Léger. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, Volume 64(Numbers 1-2 / février 2009):45–63, December 2008.

[13] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology, 2001.

[14] IBM Corp. *An architectural blueprint for autonomic computing.* IBM Corp., USA, October 2004.

[15] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[16] Cristian Ruz, Françoise Baude, and Bastien Sauvan. Using components to provide a flexible adaptation loop to component-based soa applications. *International Journal on Advances in Intelligent Systems*, 5(1 and 2):32–50, 2012.

Apéndice A

Ejemplo de uso: Asignación Autonómica de Recursos

En este capitulo se mostrará como fue utilizado el *framework* implementado, así como también la consola interactiva para GCMScript, para realización del experimento de la sección 5.2.1: la asignación autonómica de recursos en el crackeador distribuido.

El código adjuntado en este espacio es una colección de los elementos más importantes al tema que nos compete. El código completo empleado para los experimentos, así como también el de la implementación del *framework* presentado, se encuentra a publicado en la dirección: *https://github.com/mnip91/autonomic-component-system.git*.

A.1. Utilización de la API: métricas, reglas y planes

El primer paso para otorgar el comportamiento deseado al creackeador autonómico consiste en la definición de los objetivos autonómicos. Como se describió en la sección 5.2.1, estos objetivos quedan definidos por tres entidades: la métrica *AvgResponseTime*, la regla *MaxResponseTime*, y el plan *ResourcesRequest*.

A.1.1. Metrica: *AvgResponseTime*

Esta métrica tiene como fin calcular el tiempo promedio de respuesta que emplea un componente en atender sus solicitudes entrantes. Para calcular este promedio, se solicita los últimos N registros finalizados para luego retornar el promedio simple entre los tiempos de respuesta registrados en estos registros.

```
public class AvgRespTimeMetric extends Metric<Long> {

    // Valor de esta métrica
    private long avgTime = 0;

    // Esta condición impone que solo aquellos registros finalizados
```

```
    // (osea, que representan solicitudes finalizadas) serán estudiadas
    private final RCondition<IncomingRecord> finishedCondition =
        new RCondition<IncomingRecord>() {
            @Override
            public boolean evaluate(IncomingRecord record) {
                return record.isFinished();
            }
        };

    // Constante que nos indicará la cantidad de registros a estudiar
    private final int nOfRecords = CrackerConfig.RECORDS_SAMPLE_SIZE;

    public AvgRespTimeMetric() {
        // se habilitará esta métrica para que sea recalculada
        // automáticamente tras cada actualización de los registros suscritos
        setEnabled(true);

        // se desea recalcular esta métrica cada vez que termine de atender
        // una solicitud, por lo tanto, se suscribe al evento pertinente
        subscribeTo(RecordEvent.REQUEST_SERVICE_ENDED);
    }

    @Override
    public Long getValue() {
        // se retorna el ultimo valor calculado
        return avgTime;
    }

    @Override
    public Long calculate(RecordStore recordStore, MetricStore metricStore) {
        // Se utiliza el record store para obtener los últimos nOfRecords
        // registros que ya hayan finalizado
        List<IncomingRecord> records =
                recordStore.getIncoming(finishedCondition, nOfRecords);

        long sumOfTimes = 0;
        for (IncomingRecord record : records) {
            sumOfTimes += record.getServiceEndedTime() - record.getReceptionTime();
        }
        avgTime = records.size() == 0 ? 0 : sumOfTimes/records.size();
        return avgTime;
    }
}
```

Listado A.1: Implementación de la métrica *AvgResponseTime*

Regla: *MaxResponseTime*

Esta regla tiene como fin emitir una alarma si es que el valor de la métrica *AvgResponseTime* sobrepasa el valor máximo permitido.

```
public class MaxResponseTimeRule extends Rule {

    // ultima alarma emitida
    private ACSAlarm alarm = OK;

    // valor umbral de la métrica AvgResponseTime
    private long threshold = CrackerConfig.MAX_RESPONSE_TIME;

    // nombre con el que se registra la métrica AvgResponseTime
    private String metricName = CrackerConfig.AVG_RESP_TIME_METRIC_NAME;

    public MaxResponseTimeRule() {
        // se suscribe a la métrica AvgResponseTime para verificar que esta
        // regla se cumpla cada el valor de esta métrica se actualice.
        // NOTA: por defecto, esta métrica no estará habilitada. Su
        // habilitación se realizará manualmente en el futuro vía GCMScript.
        subscribeTo(metricName);
    }

    @Override
    public ACSAlarm getAlarm(MonitoringController monitoringController) {
        return alarm;
    }

    @Override
    public ACSAlarm verify(MetricStore metricStore) {
        // Se solicita el valor de la métrica y se verifica la regla
        Wrapper<Long> respTimeWrapper = monitoringCtrl.getValue(metricName);

        if ( !respTimeWrapper.isValid() ) {
            alarm = ERROR;
        } else if (respTimeWrapper.unwrap() > threshold) {
            alarm = VIOLATION;
        } else {
            alarm = OK;
        }

        return alarm;
    }
}
```

Listado A.2: Implementación de la regla *MaxResponseTime*

Plan: *ResourcesRequest*

Este plan tiene por objetivo solicitar más recursos, es decir *Solvers*, a los *Solvers* para aumentar el rendimiento del sistema. La tarea de añadir y/o eliminar *Slaves* esta implementada directamente en GCMScript (más en la sección A.2) como una serie de métodos expuestos a través del controlador de ejecución de los *Solvers*. Por lo tanto, este plan se limita a seleccionar al *Solver* con peor rendimiento para luego solicitar, a través de su controlador de ejecución del *Solver* escogido, que se añada un nuevo *Slave*.

```
public class ResourcesRequestPlan extends Plan {

    // Nombre usado para registrar la métrica AvgRespTimeMetric
    private String avgTimeMetric = CrackerConfig.AVG_RESP_TIME_METRIC_NAME;

    public ResourcesRequestPlan() {
        // Se suscribe a la regla MaxResponseTime para ejecutar este
        // plan cada vez que dicha regla sea violada.
        subscribeTo(CrackerConfig.MAX_RESP_TIME_RULE_NAME);
    }

    @Override
    public void doPlanFor(String ruleName, ACSAlarm alarm, MonitoringController
            monitorCtrl, ExecutionController executionCtrl) {

        if (alarm == VIOLATION) {
            // El plan solo se ejecutará si hubo una violación de la regla

            if (stillSleeping()) {
                // Después de haber solicitado más recursos, este plan se
                // dormirá por un tiempo, dejando así un margen de tiempo
                // para que el sistema se adapte a los nuevos recursos.
                // Mientras tanto no se hace nada.
                return;
            }

            // Para saber a que Slave solicitar recursos se solicitan sus
            // tiempos de respuestas. Se solicitarán recursos al que tenga
            // peor desempeño, es decir, mayor tiempo de respuesta.
            Wrapper<Long> s0 = monitorCtrl.getValue(avgTimeMetric, pathTo(0);
            Wrapper<Long> s1 = monitorCtrl.getValue(avgTimeMetric, pathTo(1);
            Wrapper<Long> s2 = monitorCtrl.getValue(avgTimeMetric, pathTo(2);

            if (areValidValues(s0, s1, s2)) {

                Queue<Pair> queue = new PriorityQueue<>();
                queue.add(new Pair(0, s0.unwrap()));
                queue.add(new Pair(1, s1.unwrap()));
                queue.add(new Pair(2, s2.unwrap()));

                addSlave(queue, executionCtrl);
```

```
            }
        }
    }

    // Este método intentará añadir un Slave a un Solver. Primero tratará con
    // el de peor desempeño, si no es posible probara con el siguiente, y así.
    private void addSlave(Queue<Pair> queue, ExecutionController executionCtrl) {

        for (Pair pair : queue) {
            // Se utiliza el ExecutionController para ejecutar un comando GCMScript.
            // Este comando consiste una orden de ejecución remota: se ejecutará
            // la expresión add-slave($this) en Solver indicado
            Wrapper<Boolean> additionResult = executionCtrl.execute(
                    "remote-execute(\$this/sibling::Solver" + pair.index
                    + ", \"add-slave(\$this)\")");

            if (additionResult.isValid()) {
                if (additionResult.unwrap()) {
                    // Si el Slave se añadió correctamente, se hace dormir a
                    // a este plan para que el sistema pueda adaptarse al cambio
                    goToSleep();
                    System.out.println("[ACTION] Slave added on Solver" +
                        pair.index);
                    return;
                }
            }

            // no se pueden agregar más Slaves a este Solver, continuar
        }
    }

    // Resto de la clase, utilidades...
}
```

Listado A.3: Implementación del plan *ResourcesRequest*

A.2. Administración de recursos vía GCMScript.

Al momento de instanciar un componente es posible cargar uno o varios archivos con definiciones de funciones y acciones GCMScript para su uso futuro. Esta característica nos permite, por ejemplo, definir una acción encargada de añadir *Slaves* a un *Solver* y dejarla disponible para uso uso durante la ejecución del crackeador distribuido. De esta forma, es posible definir una completa API de reconfiguración a través de GCMScript, accesible en tiempo de ejecución a través de una consola interactiva o directamente desde el controlador de ejecución.

A continuación, se listarán las principales acciones definidas para la adición de *Slaves* en los

Solvers:

```
// Añade un slave al solver indicado
action add-slave(solver) {
    return add-slaves(\$solver, 1);
}

// Añade slaves al solver indicado
action add-slaves(solver, number) {

    // Se impone que el número máximo de Slaves sea 6
    if (slaves-number(\$solver) + \$number > 6) {
        fail();
    }

    solverState = state(\$solver);
    if (\$solverState == "STARTED") {
        stop(\$solver);
    }

    recursively-add-slaves(\$solver, \$number);

    if (\$solverState == "STARTED") {
        start(\$solver);
    }
    return true();
}

// GCMScript aún carece de suficientes estructuras de control.
// Este método utiliza recursion para añadir tantos Slaves como se desee.
action recursively-add-slaves(solver, number) {

    if (\$number <= 0) {
        return true();
    }

    // Creación del nuevo Slave
    signature = "cl.niclabs.scada.acs.examples.cracker.solver.component.Slave";
    node = \$solver/deployment-gcmnode::*;
    slave = gcm-new(\$signature, \$node);

    // Asignación del nombre adecuado (numeración correcta)
    oldSlaves = \$solver/child::*[starts-with(name(.), "Slave")];
    set-name(\$slave, get-next-available-name(\$oldSlaves));

    // Añadidura del Slave al Solver
    add(\$solver, \$slave);
    bind(\$solver/child::Master/interface::multicast-slave-itf,
        \$slave/interface::slave-itf);
```

```
    set-value(\$solver/child::Master/attribute::partitionsNumber,
        slaves-number(\$solver));

    return recursively-add-slaves(\$solver, \$number - 1);
}
```

Listado A.4: solver-lib.fscript

A.3. Manipulación manual del experimento

Una vez el crackeador distribuido este ejecutándose, se procede con la manipulación manual del comportamiento autonómico: es decir, solicitar un tiempo de respuesta no mayor a uno, y luego, modificar esta solicitud para exigir un tiempo de respuesta no menor a 3.

Para simplificar esta tarea se cargo previamente una serie de acciones de GCMScript al controlador de ejecución del componente *Dispatcher*. A continuación, se lista las acciones relacionadas con este experimento de la libreria GCMScript para el componente *Dispatcher*:

```
// acción utilizada para solicitar un tiempo de respuesta no mayor a 1 seg
action enable-max-response-time(dispatcher) {
    set-state(\$dispatcher/rule::minResponseTime, "DISABLED");
    set-state(\$dispatcher/plan::resourcesRefuse, "DISABLED");

    // habilitación de la regla y plan pertinente
    set-state(\$dispatcher/rule::maxResponseTime, "ENABLED");
    set-state(\$dispatcher/plan::resourcesRequest, "ENABLED");
}

// acción utilizada para solicitar un tiempo de respuesta no menor a 3 seg
action enable-min-response-time(balancer) {
    set-state(\$dispatcher/rule::maxResponseTime, "DISABLED");
    set-state(\$dispatcher/plan::resourcesRequest, "DISABLED");

    set-state(\$dispatcher/rule::minResponseTime, "ENABLED");
    set-state(\$dispatcher/plan::resourcesRefuse, "ENABLED");
}
```

Listado A.5: dispatcher-lib.fscript

De esta manera, la operación manual del experimento se limita a abrir una consola de GCMScript, posicionarla sobre el componente *Dispatcher*, y ejecutar alguna de estas acciones a gusto.

NOTA: la regla *MinResponseTime* y el plan *ResourcesRefuse* corresponden a una variante de la regla *MaxResponseTime* y el plan *ResourcesRequest*, utilizadas para exigir un tiempo no menor a 3 segundos y, en caso de que esto no se cumpla, disminuir el número de *Slaves*. La implementación de estos elementos es muy similar a las aquí presentadas y pueden observarse en el repositorio apuntado en el inicio de este capítulo.