Una Plataforma para el Control a Eventos Discretos de Sistemas de Software mediante Modelos en Tiempo Real: Aplicación al Consumo de Recursos en Web/Java EE

Guillermina Galache[1,3] y Rodrigo Castro[2,3]

[1]Departamento de Computación, Facultad de Ingeniería.
Universidad de Buenos Aires, Argentina.
ggalac@fi.uba.ar
[2]Departamento de Computación, Facultad de Ciencias Exactas y Naturales.
Universidad de Buenos Aires, Argentina.
rcastro@dc.uba.ar
[3]Área de Investigación, Desarrollo e Innovación (I+D+i), Epidata Consulting.
Buenos Aires, Argentina.

Abstract. El aseguramiento de calidad de servicio en sistemas informáticos mediante auto-adaptación (self-healing) ofrece ventajas particulares cuando se aplican técnicas de control a lazo cerrado. Sin embargo, se carece de herramientas robustas que faciliten combinar prácticas de ingeniería de control e ingeniería de software. En este trabajo se presenta DECSS (Discrete Event Control of Software Systems), una herramienta genérica y flexible para incorporar rápidamente capacidades de self-healing a sistemas de software y su hardware asociado. DECSS implementa control a lazo cerrado basado en modelos de simulación interactuando en tiempo real con el sistema supervisado, facilitando el diseño de modelos multiformalismo y eliminando riesgos en la transición hacia controladores productivos. Se implementó exitosamente un control sencillo para limitar el consumo de CPU y memoria de una aplicación Web sobre Java EE. Los ensayos expusieron dinámicas inesperadas de la aplicación controlada, y DECSS mostró flexibilidad para adaptar el control utilizando los nuevos conocimientos adquiridos.

Keywords: self-heailng, modelado, eventos discretos, control en tiempo real

1 Introducción

La importancia de las disciplinas de aseguramiento de la calidad de servicio y de la confiabilidad en sistemas informáticos crece acorde al incremento de la complejidad de los mismos y su adopción masiva. Al mismo tiempo, se incrementan las dificultades para implementarlas exitosamente dentro de las restricciones de competitividad en términos de tiempo y costos de desarrollo impuestos por el mercado.

Surge entonces la necesidad de diseñar estrategias que permitan a los sistemas administrarse en forma autónoma y automática en la fase de producción, partiendo exclusivamente de especificaciones de objetivos de calidad definidos en la fase de diseño.

Esto implica que posean la capacidad de reaccionar ante fallas propias y/o cambios en su entorno de ejecución que impliquen un desvío del objetivo de calidad estipulado, sin necesidad de la intervención humana y de forma transparente para el usuario.

Diversos esfuerzos realizados tanto en el ámbito académico como industrial proponen enfoques para obtener lo que se conoce como sistemas self-healing [1,2]. En lugar de intentar considerar a priori todas las posibles condiciones de falla de un sistema, se acepta la no idealidad del mismo y el débil conocimiento previo del comportamiento de sus usuarios, y se le agrega una capa de supervisión que detecta anomalías y reacciona controlando parámetros de operación buscando contrarrestar el estado de falla. Esta estrategia es mucho más eficiente en términos de escalabilidad (ya que no depende del tamaño del sistema) y posibilita que las estrategias de control puedan ser reutilizadas en diversos sistemas de software que compartan ciertas propiedades en su arquitectura.

Entre los principales enfoques existentes puede mencionarse Model-based Adaptation (Garlan y Schermerl [3]) en donde se utilizan modelos de la arquitectura del sistema en tiempo de ejecución representados como grafos de componentes interactuantes. Otro enfoque distinto se basa en modelos de lazos de realimentación siguiendo la línea de la Teoría de Control (Hellerstein y otros [4], Fig. 1).

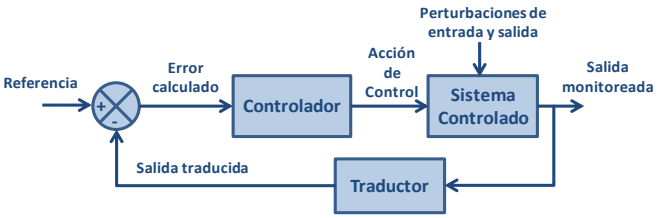


Fig. 1. Diagrama en bloques de un lazo de control a lazo cerrado.

Todo sistema self-healing puede plantearse conceptualmente como 4 etapas centrales expresables en términos de Teoría de Control: *a) monitoreo*, con el fin de la recolección de datos acerca de la ejecución del sistema y de su entorno; *b) traducción y cálculo del error*, en donde los datos recolectados son adaptados y comparados con los objetivos de calidad, estableciendo los desvíos o "errores"; *c) control*, dedicado al análisis y cálculo de un curso de acción para contrarrestar los desvíos y *d) actuación*, para la ejecución de acciones de control sobre parámetros manipulables del sistema.

El control de sistemas de software (y redes de comunicaciones) en tiempo real es una disciplina joven que aplica técnicas clásicas de control sobre sistemas de cómputo considerando a éstos últimos como los objetos de control [4]. Típicamente, el objetivo de control es mantener métricas de performance (*delay, jitter, throughput, utilization*) dentro de rangos aceptables especificados por requerimientos de calidad de servicio. Las acciones de control son eventos discretos que modifican la asignación de los recursos disponibles (*pools, buffers, queues, slots*) a las entidades que compiten por ellos (*tasks, packets, jobs, requests*). Si bien estos sistemas de control son típicamente diseñados con técnicas ad–hoc, la Teoría de Control es cada vez más utilizada por su robustez teórica y metodológica [5] que permite garantizar propiedades de performance y estabilidad del sistema controlado ante condiciones impredecibles. Estas características no están disponibles con otras técnicas como métodos formales o teoría de colas.

A pesar de la vasta aceptación de estas ventajas por parte de comunidades relacionadas con la calidad de sistemas de software (ver [6] para una revisión sistemática de más de 160 artículos científicos) las implementaciones propuestas consisten en prototipos destinados a casos específicos de laboratorio, o bien, en módulos de productos comerciales sin posibilidad de reutilización en otros sistemas. Las pocas excepciones que han ofrecido soluciones genéricas y reutilizables [7,8,9] carecen de la posibilidad de definir formalmente la implementación de los controladores. Los autores consideran a esta situación uno de los principales motivos que obstaculizan la adopción de técnicas de self-healing basadas en Teoría de Control en proyectos de software.

En este artículo se presenta DECSS (Discrete Event Control of Software Systems), una nueva plataforma genérica y reutilizable para incorporar rápidamente capacidades de self-healing a sistemas de software y su infraestructura de hardware. DECSS se ajusta al paradigma de lazo cerrado de la ingeniería de control y se sustenta en un formalismo matemático robusto para la implementación de controladores mediante modelado y simulación de eventos discretos en tiempo real.

En la Sección 2 se presentan conceptos preliminares acerca de modelado y simulación en tiempo real. En las Secciones 3 y 4 se define el problema y se presenta a DECSS como solución. La Sección 5 describe su implementación técnica y en la Sección 6 se presenta el caso de prueba utilizado para validar su funcionamiento.

2 Conceptos Preliminares

2.1 El formalismo DEVS

DEVS (Discrete Event Systems Specification) [10] es un formalismo genérico para describir sistemas de eventos discretos que realizan un número finito de cambios en intervalos finitos de tiempo. Un modelo DEVS procesa una trayectoria de eventos de entrada y, según esa trayectoria y sus propias condiciones iniciales, produce una trayectoria de eventos de salida. Un sistema modelado con DEVS se describe como una composición jerárquica de modelos comportamentales (atómicos) y/o estructurales (acoplados). Formalmente, un modelo atómico DEVS se define por la estructura M en la Fig.2

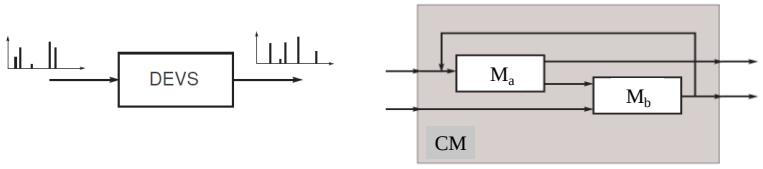


Fig. 2. Modelo DEVS atómico (izq.) y Modelo DEVS acoplado (der.)

M queda definido por tres conjuntos y cuatro funciones dinámicas: el conjunto **X** de los posibles valores que puede adoptar un evento de entrada, el conjunto **Y** de los posibles valores de los eventos de salida, el conjunto **S** de los posibles valores de estado interno, una función de transición interna **δint**, una función de transición externa **δext**, una función de salida **λ** y una función de avance de tiempo **ta**. Una gran variedad de sistemas

complejos pueden pensarse como acoplamiento de sistemas más simples, donde los eventos de salida de algunos subsistemas se convierten en eventos de entrada de otros. Puede demostrase formalmente [10] que el acoplamiento de modelos DEVS (**CM** en la Fig. 2) siempre puede expresarse como un nuevo modelo DEVS atómico, permitiendo que los sistemas complejos sean estructurados de manera modular y jerárquica.

Cada posible estado **s** ∈ **S** tiene asociado un avance de tiempo calculado mediante la función **ta(s)** consistente en un número real no negativo que indica el lapso durante el cual el sistema permanecerá en un determinado estado en ausencia de eventos de entrada. Cuando expira dicho lapso, el sistema produce un evento de salida dependiente del estado en el cual se encontraba (el evento emitido adopta el valor **λ(s)**) e inmediatamente cambia a un nuevo estado calculado por **δint(s)**. Se dice entonces que el sistema realizó una *transición interna*. Si antes de que finalice el lapso de avance de tiempo llega un evento externo con valor **x**, el sistema cambia inmediatamente a un nuevo estado **s'= δext(s,e,x)** que dependerá del estado previo **s**, del tiempo transcurrido **e** en dicho estado y del evento externo recibido **x**.

La generalidad del formalismo DEVS permite modelar cualquier controlador discreto y/o continuo siguiendo metodologías ad-hoc o basadas en la Teoría de Control. De este modo, diversas fases en un proceso de diseño, verificación y validación de controladores -que demanda la combinación de subsistemas a tiempo discreto y/o eventos discretos y/o continuos- puede realizarse de forma consistente en una única herramienta y bajo un único marco teórico, reduciendo notablemente la posibilidad de errores [11].

2.2 Control mediante simulación de eventos discretos en tiempo real

Una de las ventajas principales de la especificación DEVS es la de ser intrínsecamente orientada a la simulación, por lo cual la implementación de simuladores para modelos DEVS es notablemente sencilla [10]. DEVS facilita además la ejecución de modelos en modo "embebido" en un hardware de propósito específico, como por ejemplo, un servidor de control que supervise aplicaciones de software distribuidas en una red.

Se adopta aquí el concepto de "simulación embebida" del dominio de los sistemas embebidos en donde los simuladores asumen un rol similar al de una máquina virtual de modelos [11,12]. El objetivo de una simulación embebida pasa de la concepción clásica de *"simular"* modelos (no necesariamente en tiempo real) al concepto de *"ejecutar"* dichos modelos *interactuando en tiempo real* con sistemas externos al simulador (simulación tipo System-In-The-Loop). Esta modalidad tiene la ventaja de permitir seguir una metodología de desarrollo basada en modelos: inicialmente, el modelo de un controlador es desarrollado de manera aislada del sistema real a controlar (utilizando un modelo equivalente del mismo). Luego, a medida que el proyecto avanza, las sucesivas versiones del controlador pueden ser verificadas operando contra el sistema real sin necesidad de adaptar código a plataformas específicas, ya que el controlador es ejecutado por el mismo simulador en el cual fue desarrollado, aunque esta vez, operando en modo System-In-The Loop.

Sólo se requiere redireccionar las salidas y entradas del controlador hacia *el sistema real* propiamente dicho, en lugar de hacia *un modelo del sistema real*.

3 Definición del Problema

La gran mayoría de las empresas han incorporado la utilización de sistemas informáticos como componente crítico de sus negocios. Aun más, muchas los utilizan como único medio para brindar sus servicios y los escenarios más exigentes se requiere que los sistemas presten servicio las 24 horas los 365 días del año. Sin embargo, se destinan escasos (y a veces nulos) recursos para control del cumplimiento de los objetivos de calidad durante la ejecución de los sistemas. La administración de los mismos se suele limitar al monitoreo y mantenimiento manual reactivo ante la ocurrencia de fallas.

Una de las causas de este aparente desacople radica en la falta de herramientas que permitan combinar la capacidad de diseñar controladores confiables y robustos con la capacidad de poder implementarlos en la práctica con esfuerzos moderados. En este contexto, las comunidades de la ingeniería de control e ingeniería de software tienen aún importantes esfuerzos de convergencia por realizar.

Se define entonces la necesidad contar con mejores herramientas para *simplificar el proceso de implementación* de controladores para self-healing (aplicando las mejores prácticas de la ingeniería de software) y al mismo tiempo permitan *diseñar formalmente e implementar controladores* de diversa complejidad (aplicando las mejores prácticas de la ingeniería de control).

4 Solución Propuesta: DECSS (Discrete Event Control of Software Systems)

Se adopta el enfoque de la ingeniería de control a lazo cerrado aplicado a sistemas informáticos, para cuyos algoritmos de control se exigirá que:

- Su especificación se realice mediante un lenguaje de modelado formal
- Su desarrollo obedezca a una metodología basada en modelos de simulación
- Su ejecución responda a una simulación en tiempo real del tipo System-in-the-Loop

Estas propuestas se apoyan en la vasta experiencia de la ingeniería de control de sistemas físicos, sólidamente establecida durante décadas en las industrias de procesos.

Se propone entonces a DECSS (Discrete Event Control of Software Systems), una plataforma para diseñar e implementar controladores a lazo cerrado (ad-hoc o basado en Teoría de Control) utilizando modelos bajo el formalismo DEVS y ejecutándolos a modo System-In-The-Loop para supervisión de aplicaciones de software.

Con el fin de hacer a DECSS fácilmente adoptable, flexible y extensible, se proponen tecnologías estándar y abiertas para las etapas de *monitoreo* y *actuación*, minimizando la barrera de adopción en proyectos de software partiendo de la pequeña y mediana escala. Acorde a estas premisas, se seleccionó a Nagios [13] y JMX [14] como tecnologías de monitoreo y actuación maduras para operar sobre aplicaciones de software y sus recursos de infraestructura (CPU, memoria, red, etc.).

Para la etapa de *control* DECSS incorpora el simulador de eventos discretos en tiempo real PowerDEVS [15], un entorno de modelado y simulación DEVS (con orientación al control) para propósitos de investigación, educación y aplicación industrial.

El formalismo DEVS permite modelar y combinar sistemas discretos (tiempo discreto y/o eventos discretos) y sistemas continuos (aproximados con tanta precisión como se requiera [10, 16]). Esta característica distintiva posibilita la inmediata adopción de metodologías matemáticamente robustas para el diseño e implementación de controladores híbridos basados en modelos [12], desde sencillas máquinas de estado hasta técnicas avanzadas de control no lineal.

En el presente trabajo se desarrollaron las piezas de software necesarias para poder integrar modelos DEVS ejecutando en la herramienta PowerDEVS con las tecnologías de monitoreo y actuación seleccionadas. Así, los controladores basados en DEVS controlan los parámetros de infraestructura que finalmente impactan en el desempeño y calidad de servicio de la aplicación de software bajo estudio.

4.1 Arquitectura

En la Fig. 3 se muestra el esquema de la infraestructura desarrollada para DECSS en donde pueden verse sus principales componentes.

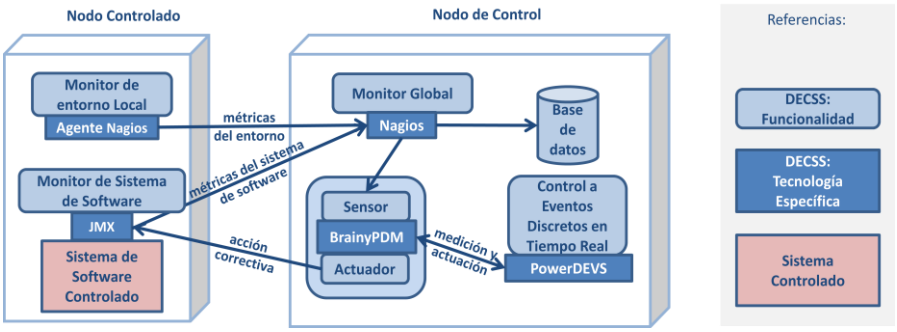


Fig. 3. – Diagrama de la arquitectura e infraestructura desarrollada para DECSS

A la izquierda se encuentra el Nodo Controlado, donde se ejecutan el sistema de software controlado y un Monitor del Entorno Local que recolecta métricas propias del nodo (consistente en Agentes Nagios, tantos como tipos de métricas se desee obtener). A la derecha, se encuentra el Nodo de Control con las tecnologías necesarias para ejecutar los algoritmos de control supervisando uno o más nodos externos. Allí se encuentra el Monitor Global que recolecta las métricas provenientes de los monitores de entornos locales y del sistema de software (obtenidas vía la tecnología JMX).

Las métricas recolectadas son enviadas al Sensor quien se encarga de adaptar el formato de las mediciones y transmitirlas al control en tiempo real. Este último es el encargado de calcular si existen desvíos apreciables entre los valores deseados para las métricas y sus valores actuales, determinando si se debe aplicar una acción correctiva, en cuyo caso calcula su valor y se lo comunica al Actuador quien lo aplica vía JMX.

5 Implementación

5.1 Monitores

Tanto para el rol de Monitor del Entorno Local como para el de Monitor Global se utilizó Nagios [13], una herramienta de monitoreo altamente utilizada en el mercado.

Nagios está construido sobre una arquitectura de agentes/servidor, es decir que está conformado por varios agentes (plugins) que recolectan información en hosts remotos y la envían a un servidor que centraliza la recepción y procesamiento de las mediciones. Los plugins son pequeños programas o scripts (escritos en cualquier lenguaje) que pueden correr tanto localmente o en forma remota al hardware del servidor. Estos plugins no forman parte de Nagios, siendo usualmente desarrollados por terceros.

Para la recolección de métricas del sistema de software a controlar se utilizó el plugin check_jmx (obtenido de [17]) desarrollado en Java, que permite monitorear atributos JMX desde Nagios. Para ser utilizado en DECSS se le agregó una sección *performance data* (información que puede adicionarse al código de respuesta y a la descripción de estado que debe devolver todo plugin). Esta información puede enviarse a aplicaciones externas, permitiendo elegir libremente cual (o cuales) se desea utilizar para el procesamiento y/o almacenamiento de las métricas [18]. Esta flexibilidad es una de las principales ventajas de Nagios, permitiendo personalizaciones sin tener que modificar el componente *core* del mismo.

Para la recolección de métricas del entorno se pueden utilizar tanto plugins oficiales [19] de Nagios como desarrollados por terceros [17], según la necesidad.

5.2 Sensor y Actuador

Tanto el sensor como el actuador fueron implementados utilizando BrainyPDM [20], un módulo Java autónomo cuya función principal es procesar y almacenar información de *performance data* recibida desde Nagios. Para poder ser utilizado en DECSS se realizaron adaptaciones a BrainyPDM que serán explicadas a continuación.

Sensor.
En BrainyPDM el procesamiento y análisis de la información recibida de Nagios está basada en plugins (llamados *procesadores*). En un archivo XML se configuran cuales están disponibles indicando para cada uno la clase que lo implementa, los parámetros comunes que definen su comportamiento (e.g., el *host* y el *service* de los cuales el procesador procesará las métricas) y en algunos casos parámetros propios del procesador.

Se pueden desarrollar plugins propios o utilizar alguno de los que BrainyPDM trae ya implementados, entre los cuales se encuentra el llamado DefaultPerformanceData que utiliza una expresión regular para determinar que formato de performace data procesará. Este plugin fue el utilizado en DECSS, configurándolo para procesar mensajes con la siguiente estructura: {label=value[UOM];[warn];[crit];[min];[max]}

Cuando recibe información de Nagios, BrainyPDM determina cual plugin la procesará en base al *host* y *service* de procedencia y a los parámetros de cada uno. A este

proceso se le agregó una funcionalidad para enviar los datos recibidos hacia el control externo en tiempo real. Para ello, se siguió el criterio utilizado para los procesadores: se definieron plugins (llamados *transmisores*) encargados de enviar métricas al sistema de control. Se configura en un archivo XML los transmisores disponibles y se indican para cada uno de ellos la clase que lo implementa, parámetros comunes que definen su comportamiento (e.g., métricas a transmitir) y parámetros propios de cada transmisor.

Luego que la información recibida desde Nagios es procesada, si la misma contiene la sección de performance data además de almacenarla en la base de datos correspondiente se obtienen todos los transmisores configurados para transmitir las métricas recibidas, según host, servicio y nombre de la métrica de la cual se trate, y se envía los datos a través de ellos.

Se desarrolló así un transmisor denominado TransmisorUDP, que envía las métricas a través del protocolo UDP a la dirección IP y el puerto configurado como parámetros propios. En caso de ser necesario otro tipo de comunicación con el control externo, se debe desarrollar un nuevo transmisor (haciéndolo heredar de la clase TransmisorBase) e implementar la nueva lógica deseada para la transmisión.

Actuador.

La funcionalidad del actuador es una extensión realizada a BrainyPDM para que pueda ser utilizado como parte de DECSS. Para su diseño se siguió el mismo criterio usado para la transmisión de métricas al control externo: en un archivo XML se configuran los actuadores disponibles, encargados de recibir del control las acciones correctivas y de aplicarlas. Para cada uno se configuran sus parámetros, la clase que lo implementa y la URL en donde se encuentran los MBeans sobre los cuales accionará.

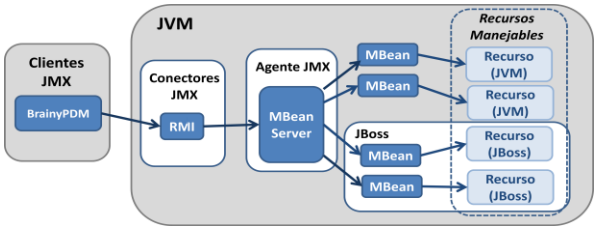


Fig. 4. Esquema del proceso de actuación

Se implementó un actuador denominado ActuadorByUDP, que recibe la información vía sockets de red mediante el protocolo UDP. Esto flexibiliza el reemplazo del sistema de control (la tarea se reduce a reconfigurar el actuador correspondiente) o desarrollar un actuador nuevo (heredando de la clase abstracta ActuadorBase). Con este diseño se logran las características deseadas de flexibilidad y extensibilidad para el acoplamiento entre los módulos -a priori heterogéneos- del actuador y controlador.

La acción de control se realiza a través de la tecnología JMX, lo que implica ejecutar alguna operación o modificar algún atributo en un MBean remoto (Fig.4).

Para que un MBean pueda ser accedido por DECSS se debe configurar en un archivo XML su Object Name y los métodos que se quieren exponer, asignándoles identificadores unívocos que serán usados por el sistema de control para indicar al actuador

cual MBean/método debe ejecutar mediante un mensaje con la siguiente estructura: {MBean1 | operacion | arg1,…,argn | | MBean2 | operacion | arg1,…,argn | | …}

5.3 Control a lazo cerrado en Tiempo Real

DECSS prevé la adopción de distintos sistemas de control sin que ello implique modificaciones en el resto de la infraestructura (monitoreo y actuación) gracias al uso de plugins para los sensores y actuadores (descriptos más arriba).

Para utilizar PowerDEVS como ejecutor del sistema de control se le debió incorporar funcionalidades de conexión con sistemas externos (en este caso con los plugins a través de sockets UDP). Para manejar el envío de acciones de control se incorporó la función *sendNET()*, invocable desde cualquier modelo DEVS atómico (Fig. 5b). Para la recepción de datos se implementó un mecanismo de suscripciones, mediante el cual un modelo atómico DEVS debe registrase como interesado en ser notificado de la llegada de datos por un determinado puerto UDP (Fig. 5a). Aquellos modelos atómicos suscriptos son notificados desde el motor de simulación utilizando la función *externalInput()* (Fig. 5b) emulando la ocurrencia de un evento externo **δext(s,e,x)** (Fig. 2).

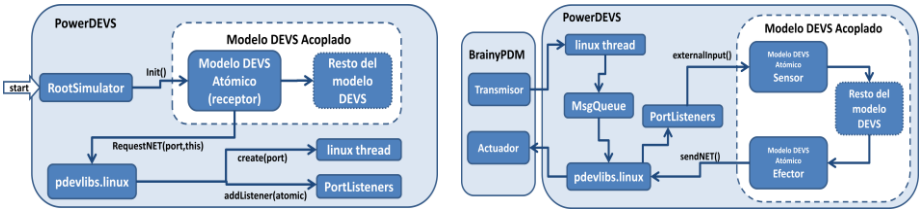


Fig. 5. Esquema de procesos. (a) Inicialización. (b) Recepción y envío de datos.

6 Prueba de Concepto

En esta sección se valida el funcionamiento esperado de DECSS en un caso de aplicación práctica. Se seleccionó como sistema a controlar a JBoss 5.1.0.GA [21], una plataforma de aplicaciones Web basada en Java EE (Java Enterprise Edition, o JEE) vastamente adoptada en la industria, nativamente administrable vía JMX.

Para la estrategia de control a lazo cerrado, se presenta un caso sencillo de tipo "exploratorio" basado en una máquina de estados. Actualmente no se dispone de sistemas de software que provean una documentación de su respuesta dinámica en términos de modelos matemáticos requeridos por la Teoría de Control. Consecuentemente, se recurre a ejercicios preliminares de identificación, para ganar conocimiento acerca de la dinámica desconocida del sistema y de las respuestas ante distintas acciones de control.

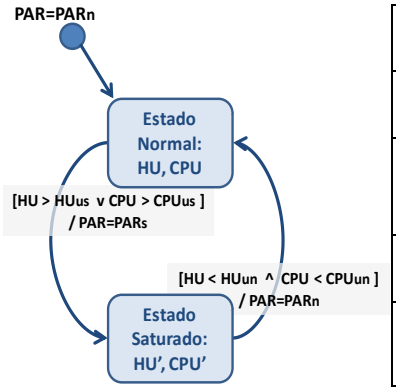
Estos ejercicios conducen a la obtención de modelos matemáticos a partir de los que se diseñan controladores mediante Teoría de Control [4]. Si bien esos procedimientos quedan fuera del alcance del presente trabajo, DECSS permite de modo transparente el reemplazo incremental de controladores sencillos por otros más avanzados.

6.1 Escenario de Prueba

Se plantea un escenario tomado de la experiencia práctica en instalaciones productivas: JBoss se encuentra en estado de carga intensa pero su servidor web embebido (JBoss Web Server) tiene aún capacidad para seguir atendiendo pedidos. Se plantea la hipótesis que nuevas peticiones adicionales cargarán aún más al sistema pudiendo causar su colapso, impidiendo dar servicio hasta que una intervención humana lo estabilice nuevamente (típicamente, mediante un reinicio de todo el sistema).

Control.
Se propone detectar cuando JBoss se encuentre sobrecargado o bien próximo a estarlo, y entonces actuar reduciendo la capacidad de recepción de nuevas peticiones. Las métricas elegidas para medir la carga son: porcentaje ocupado de memoria Heap de la JVM y porcentaje de uso de CPU por parte de JBoss (i.e., viendo a la JVM de JBoss como un proceso del sistema operativo). Para las variables de control, se eligen dos atributos que determinan el comportamiento de JBoss Web Server respecto del mecanismo de recepción de pedidos de clientes [22]:

- **maxThreads**: cantidad máxima de threads de procesamiento de sistema operativo que serán creados y asignados a procesar los pedidos de cada cliente.
- **acceptCount**: determina la longitud máxima de la cola en donde se encolan las nuevas peticiones *cuando todos los threads de procesamiento se encuentran ocupados*. Cuando esta cola se llena toda nueva conexión será rechazada.



HU **CPU**	% de memoria Heap Utilizada y % de CPU Utilizada en el **estado normal.**
HU' **CPU'**	% de memoria Heap Utilizada y % de CPU Utilizada en el **estado saturado**.
PAR	Valores de las variables de control: PARi={maxThread$_i$, acceptCounti}, con i ϵ {n=**normal**, s=**saturado**}
HU_{us} **CPU_{us}**	Umbrales a partir de los que se considera al sistema en **estado saturado**.
HU_{un} **CPU_{un}**	Umbrales de restablecimiento a partir de los que se considera al sistema en **estado normal**.

Fig. 6. Máquina de Estados para la estrategia de control aplicada.

El controlador consiste en una máquina de estados sencilla (Fig. 6) que traslada a JBoss entre dos estados: uno denominado **saturado** (o sobrecargado) que ocurre cuando el porcentaje de uso de CPU o el porcentaje ocupado de memoria Heap de la JVM superan los umbrales de saturación definidos, y otro denominado **normal**, que ocurre cuando ambas métricas están por debajo de los llamados umbrales de restablecimiento.

En la Fig.7 se muestra el diagrama en bloques del sistema de control implementado en la interfaz gráfica de PowerDEVS. Se observan tres modelos DEVS atómicos: el

Sensor (recepción de métricas), el Procesador (lógica de control) y el Efector (Aplicación de acciones de actuación, cuando el procesador así lo determine).

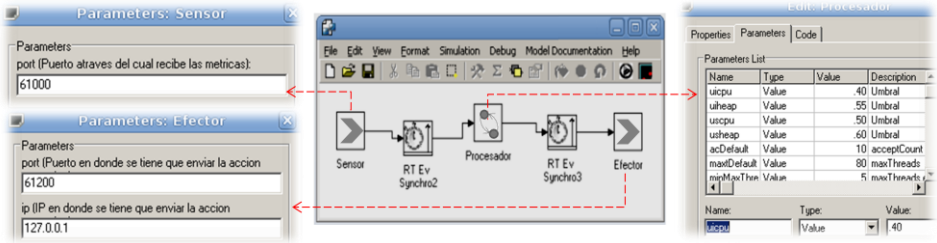


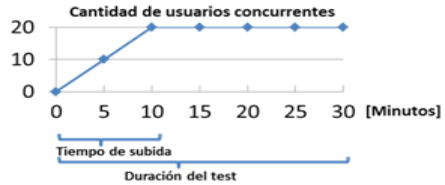
Fig. 7. Modelos DEVS implementados en la interfaz gráfica de modelado de PowerDEVS (centro) y pantallas de configuración de parámetros (izquierda y derecha).

Monitoreo.
Para obtener mediciones de utilización de recursos del proceso JVM se eligieron los plugins de Nagios check_process_resources [17] y check_by_ssh [19] que utilizados en conjunto permiten conocer el uso de memoria y de CPU de un proceso remoto. El porcentaje de memoria Heap utilizada se calcula en el sistema de control, partiendo de disponer mediciones de memoria Heap *libre* y de memoria Heap *total* (vía JMX).

6.2 Casos de prueba

Se utilizó la aplicación Duke's Bank [23] que emula transacciones bancarias online, a la cual se le generó carga utilizando JMeter 2.5.1 [24], ejecutando un script que consta de 5 operaciones: login a la aplicación, listado de las cuentas, consulta de movimientos de dos cuentas, extracción de una de las cuentas y logout. Los parámetros de carga del test son mostrados en la Fig. 8.

JMeter generará carga sobre la aplicación durante 30 minutos simulando el acceso de 20 usuarios concurrentes ejecutando las transacciones definidas en el script. Los usuarios se incorporan progresivamente durante los primeros 10 minutos del test hasta alcanzar la población de régimen.



- *Población total de usuarios*: 20
- *Tiempo de subida*: 10 minutos
- *Duración del test*: 30 minutos
- *Think Time entre pedidos consecutivos*: Distribución Gaussiana (valor medio: 300 ms., desviación estándar: 100 ms.)

Fig. 8. Diseño de la carga de prueba y su población de usuarios concurrentes.

Caso de Test 0 (Línea de Base).
Los gráficos de la Fig. 9 muestran el comportamiento del sistema bajo carga y "a lazo abierto" (sin la utilización de DECSS). Se observa en (a) que el consumo de CPU se mantiene por encima del 60% durante toda la prueba, y en (b) que el porcentaje de

memoria Heap libre oscila entre el 56% y el 10% aproximadamente evidenciando la acción del *Garbage Collector* de la JVM en 2 ocasiones. En (d) y (c) se muestran la cantidad de threads creados (currentThreadCount) y ocupados (currentThreadBusy) respectivamente, los cuales aumentan progresivamente con la cantidad de usuarios que se van creando, hasta llegar al valor de la población en régimen.

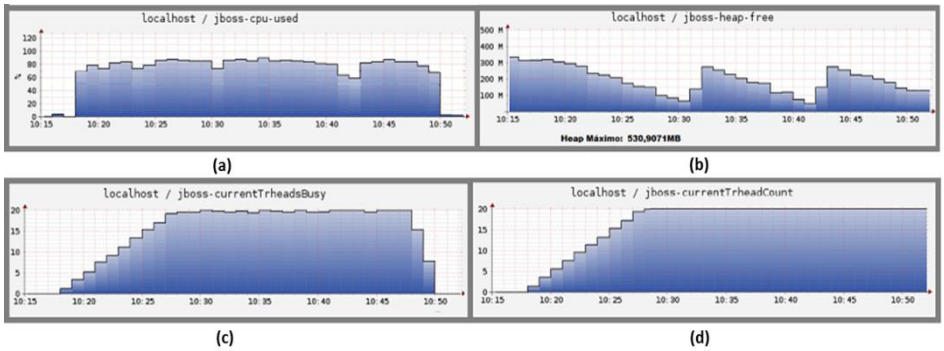


Fig. 9. Caso de Test 0 (ejecución sin DECSS). (a) Consumo de CPU (b) Memoria Heap libre (c) Cantidad de threads ocupados (d) Cantidad de threads creados.

En el marco del escenario de prueba se considera que un uso de CPU superior al 60% es una indicación de sobrecarga de JBoss. Luego, con el fin de mantener ese porcentaje en valores menores, se realizaron dos casos de test exploratorios aplicando DECSS con distintos parámetros de control, utilizando la carga de la Línea de Base.

Caso de Test 1.

Parámetros del controlador y resultados obtenidos:

Umbrales	Variables de control
Saturación: HUus=60%, CPUus=55%	**Estado normal**: PARn: maxThreadn=80, acceptCountn=10
Restablecimiento: HUun=55%, CPUun=50%	**Estado saturado**: PARs: maxThreads=5, acceptCounts=1

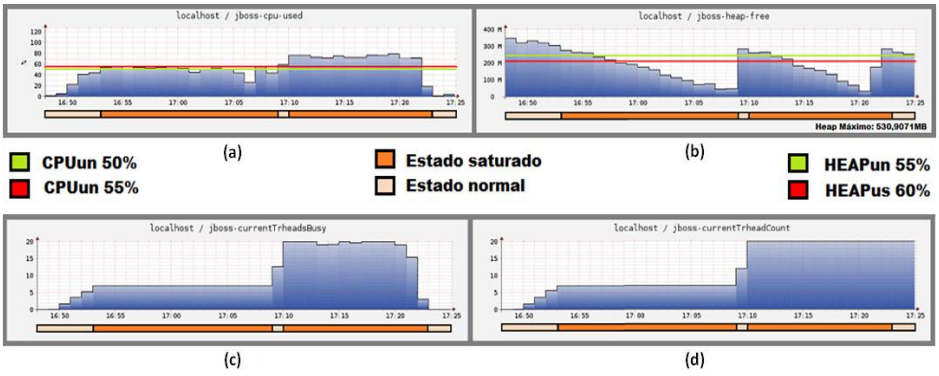


Fig. 10. Caso de Test 1 (DECSS aplicado). (a) Consumo de CPU (b) Memoria Heap libre (c) Cantidad de threads ocupados (d) Cantidad de threads ocupados.

En este caso el JBoss entró en su estado saturado en dos oportunidades (Fig. 10 a). En el primer período se observa un consumo de CPU menor al 60%, mientras que en el segundo se tiene un comportamiento similar al caso del sistema a lazo abierto.

Esta asimetría es claramente no deseable. Al regresar al estado normal -luego del primer ciclo en estado saturado- el controlador vuelve a subir el parámetro maxThread, permitiendo a JBoss crear nuevos threads. Sin embargo, cuando maxThread es restringido nuevamente -segundo ciclo en estado saturado- JBoss no disminuye la cantidad de threads como se esperaría que ocurra (ver Fig. 10d), lo cual es considerado como un bug de la aplicación. En cuanto al consumo de memoria Heap no se observaron diferencias se observa un comportamiento similar al caso sin DECSS aplicado (Fig. 10b).

Caso de Test 2.

Para evitar el problema detectado en el Caso de Test 1, se modificaron los parámetros y la estrategia de control: al cambiar de estado no se modifica el valor de maxThread (i.e., maxThreadn=maxThreads) y se agregó un acción a la transición de normal a saturado que fuerza la ejecución del *Garbage Collector* para liberar memoria Heap.

Parámetros del controlador y resultados obtenidos:

Umbrales	Variables de control
Saturación: HUus=60%, CPUus=50%	**Estado normal**: PARn: maxThreadn=5, acceptCountn=10
Restablecimiento: HUun=55%, CPUun=45%	**Estado saturado**: PARs: maxThreads=5, acceptCounts=1

En este caso hubo múltiples cambios de estado del JBoss, el cual operó durante todo el test acorde a los objetivos de calidad estipulados, es decir, respetando los umbrales para el consumo de CPU y el porcentaje de memoria Heap utilizada (Fig. 11a y 11b). Sin embargo, esto sucede en detrimento de la flexibilidad del control, debiendo mantener siempre un valor bajo (muy restrictivo) para la cantidad de pedidos posibles de ser atendidos concurrentemente (Fig. 11c y 11d).

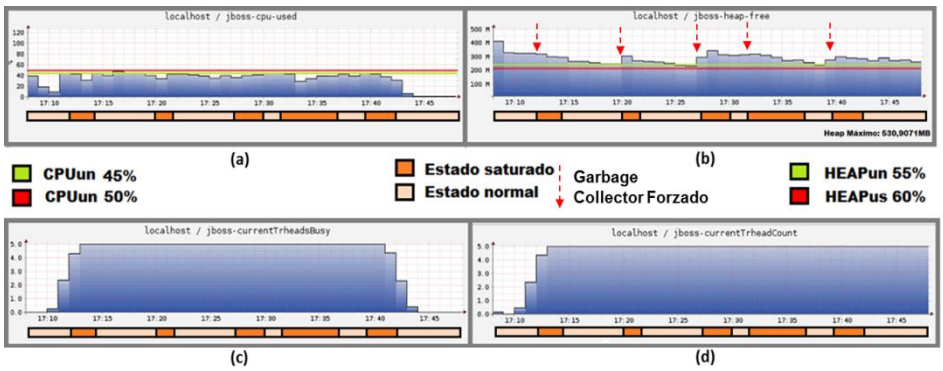


Fig. 11. Caso de Test 2 (DECSS aplicado). (a) Consumo de CPU (b) Memoria Heap libre (c) Cantidad de threads ocupados (d) Cantidad de threads creados.

Discusión de Resultados.
Se logró el objetivo del escenario de prueba, es decir, mantener el porcentaje de uso de CPU y de memoria Heap dentro de límites establecidos. Las anomalías detectadas en el Caso de Test 1 son frecuentes en los productos de software: el comportamiento real difiere del esperado según la documentación. Esta característica afecta la aplicabilidad de estrategias de self-healing basadas en modelos, y pone de relevancia la importancia de realizar ensayos preliminares sistemáticos para caracterizar la dinámica del sistema.

DECSS facilitó considerablemente esta tarea: ya en el Caso de Test 2 se logró el objetivo deseado mediante un sencillo cambio de parámetro y la inclusión de un comando de actuación extra dirigido hacia componentes específicos del sistema controlado (forzado de Garbage Collector de la JVM), mostrando la flexibilidad de la herramienta.

Si bien no fue parte del foco de los tests, debe mencionarse la evolución de los tiempos de respuesta de las transacciones en los 3 experimentos. Lo esperado es que con DECSS aplicado (cantidad restringida de threads posibles de ser creados) hubiese conexiones *rechazadas por timeout* y que aquellas conexiones *aceptadas* muestren un tiempo de respuesta menor al caso sin DECSS. Sin embargo, no se observaron timeouts y los tiempos de respuesta fueron mayores con DECSS.

Una explicación razonable es la dependencia con otros parámetros que afectan el comportamiento de recepción de pedidos de JBoss (muy probablemente *keep alive timeout*) que no formaron parte del control, y que deberán ser investigados.

7 Conclusiones y Trabajo Futuro

En este trabajo se presentó DECSS (Discrete Event Control of Software Systems), una plataforma flexible y extensible para la incorporación de capacidades de Self-healing a sistemas de software. DECSS facilita el diseño de controladores híbridos y permite su implementación con mínimos esfuerzos, promoviendo la combinación de mejores prácticas de la ingeniería de software y de la ingeniería de control.

Su arquitectura modular basada en plugins y sockets de red permite una sencilla adaptación de las tecnologías de monitoreo, control y/o actuación acorde a distintas necesidades, aceptando incluso varios sistemas de control cooperando en tiempo real.

DECSS implementa controles acorde al paradigma de simulación tipo System-In-The-Loop: los modelos de los controladores pueden ser desarrollados y ensayados mediante una simulación tipo standalone, para luego ser redireccionados en forma transparente hacia el sistema real (ejecución en tiempo real). Esta estrategia elimina los riesgos de errores de traducción o reimplementación de la lógica de control. La adopción del formalismo DEVS permite combinar distintos controladores híbridos (a tiempo discreto, eventos discretos o continuos) bajo una única herramienta multiformalismo.

DECSS permite el ensayo y estudio del funcionamiento de sistemas de software ante cambios de parámetros en tiempo real posibilitando el análisis sistemático de su dinámica. Al tratarse de una plataforma extensible, facilita su adopción para investigación en self-healing o para diagnósticos integrales con bajo costo y alto valor agregado gracias al reuso de conocimiento encapsulado en los modelos.

DECSS fue utilizado exitosamente para desarrollar un controlador sencillo (tipo on-off) cuyo diseño se basó en heurísticas obtenidas de la experiencia práctica con el sis-

tema controlado. El próximo paso consistirá en utilizar DECSS para aplicar técnicas de identificación de sistemas y desarrollar controles avanzados usando Teoría de Control.

8 Referencias

1. D. Ghosh, et al., Self-healing systems - survey and synthesis. Decision Support Systems, Vol. 42, pp. 2164-2185 (2007).
2. Koopman, P., Elements of the Self-healing System Problem Space. In: Proceedings of the Workshop on Software Architectures for Dependable Systems WADS03. (2003)
3. Garlan D., Schmerl B.: Model-based Adaptation for Self-healing Systems. In: WOSS ' 02: Proceedings of the first workshop on Selfhealing. (2002)
4. Hellerstein J.L., Diao Y., Parekh S., Tilbury D.M.: Feedback Control of Computing Systems. IEEE John Wiley & Sons, Inc. (2004)
5. Arzen, K. et. al.: Conclusions of the ARTIST2 roadmap on control of computing systems. SIGBED Rev., Vol.3 Nº 3, pp.11–20 (2006).
6. Patikirikorala P., Colman A., Han J., Wang, L.: A Systematic Survey on the Design of Self-Adaptive Software Systems using Control Engineering Approaches. In: 7th Intl. Symposium on Software Eng. for Adaptive and Self-Managing Systems, Zürich, Switzerland. (2012)
7. Goel A., Steere D., Pu C., Walpole J.: Swift: A feedback control and dynamic reconfiguration toolkit.(1999)
8. Li B. ,Nahrstedt K.: A control-based middleware framework for quality of service adaptations. (1999)
9. Zhang R., Lu C., Abdelzaher T., Stankovic J.: Controlware: a middleware architecture for feedback control of software performance. In: International Conference on Distributed Computing Systems, pp. 301-310. (2002)
10. Zeigler, B., Kim, T., Praehofer, H: Theory of Modeling and Simulation. 2nd. edition. Academic Press, New York. (2000)
11. Castro, R.; Kofman, E., Wainer, G.: A DEVS-based End-to-end Methodology for Hybrid Control of Embedded Networking Systems, 3rd. IFAC Conference on Analysis and Design of Hybrid Systems, Zaragoza, Spain. (2009)
12. Wainer, G., and Castro, R.: DEMES: a Discrete-Event methodology for Modeling and simulation of Embedded Systems. Modeling and Simulation Magazine, April, 2011. Society for Modeling and Simulation International, San Diego, CA, USA. (2011).
13. Nagios, http://www.nagios.org/
14. JMX, http://docs.oracle.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf
15. Bergero, F., Kofman, E.: PowerDEVS. A Tool for Hybrid System Modeling and Real Time. Simulation: Transactions of the Society of Modeling and Simulation.Vol.87 Nº1-2 pp.113–132 (2010)
16. Cellier, F., Kofman, E: Continuous System Simulation. Springer, New York. (2006)
17. Nagios Echange Plugins, http://exchange.nagios.org/directory/Plugins
18. Nagios Performance Data, http://nagios.sourceforge.net/docs/3_0/perfdata.html.
19. Official Nagios Plugin, http://nagiosplugins.org/.
20. BrainyPDM, http://sourceforge.net/projects/brainypdm/
21. JBoss, http://www.jboss.org/
22. Jamae, J., Johnson, P.: JBoss in Action. Manning Publications Co. Cap. 5 (2009)
23. Sun Microsystems, The Java EE 5 Tutorial, http://docs.oracle.com/javaee/5/tutorial/doc/
24. Apache JMeter, http://jmeter.apache.org/.