ICT-UNPA-149-2016
ISSN: 1852-4516
Aprobado por Resolución N° 0875/16-R-UNPA

Calidad de servicio (QoS) en procesos: Escenarios de procesamiento con aspectos

Quality of service (QoS) in processes: Processing scenarios with aspects

[1] Lic. Héctor Manuel Soto Pérez, hsoto@uarg.unpa.edu.ar
[2] Dra. Sandra Casas, sicasas@unpa.edu.ar

[1] Unidad Académica Río Gallegos - Universidad Nacional de la Patagonia Austral,
[2] GISP, Instituto de Tecnología Aplicada, U.A.R.G- Universidad Nacional de la Patagonia Austral, Río Gallegos -Santa Cruz - Argentina

RESUMEN

Aplicar QoS a la gestión de procesos plantea satisfacer una serie de garantías para cumplir con las expectativas en la ejecución de las aplicaciones. En los sistemas Linux, el tratamiento de la QoS en las aplicaciones esta diferenciado por su tipo. Así una aplicación de tiempo real (TR) consta de un tratamiento privilegiado respecto a una aplicación tradicional. Esto se traduce en una mayor priorización y disponibilidad de recursos para satisfacer las demandas de las aplicaciones de TR. En cuánto a las aplicaciones tradicionales no se han incorporado mayores mecanismos para brindar garantías de QoS a las mismas. Por eso se presenta un estudio que busca identificar y analizar los requerimientos de QoS desde un enfoque aspectual. El estudio plantea una serie de escenarios, los cuales definen casos de uso basados en aspectos sobre los requerimientos de QoS para aplicaciones tradicionales Linux. Se describen conceptos sobre los requerimientos de QoS y su posterior modelado con aspectos, tal que la tarea de cumplimentar los requerimientos de QoS en programas existentes sea mucho más simple y flexible. Además se describen algunos aspectos de la aplicación de la QoS en sistemas paralelos, en distribuidos y en sistemas en red.
En resumen el presente trabajo estudia conceptos básicos de calidad de servicio y algunos de sus marcos de aplicación siguiendo un enfoque aspectual. Planteando posibles casos de uso genéricos para aplicar mejoras en la prestación de la QoS mediante aspectos en la planificación y gestión de recursos de aplicaciones tradicionales.

Palabras clave: Calidad de Servicio; Programación Orientada a Aspectos; Procesos; Thread; Sistemas Distribuidos; Sistemas Paralelos; Casos de uso.

ABSTRACT

Apply QoS to meet process management raises a series of guarantees to meet expectations in the execution of applications. On Linux systems, the management of QoS applications is differentiated by type. Thus, a real-time application (RT) comprises a preferential treatment compared to a traditional application. This translates into greater prioritization and availability of resources to meet the demands of applications RT. As for traditional applications have not been incorporated major mechanisms to provide QoS guarantees to them. Is presented a study that seeks to identify and analyze QoS requirements from an aspectual approach. The study raises a number of scenarios, which define use cases based on aspects on traditional QoS requirements for Linux applications. Concepts are described QoS requirements and subsequent modeling with aspects, such that the task of completing the QoS

requirements in existing programs is much simpler and more flexible. Also they describe some aspects of the implementation of QoS in parallel systems, distributed and networked systems.
In summary, the present study examines basic concepts of quality of service and some of its application frameworks following a aspectual approach. Considering potential use cases generic to implement improvements in the provision of QoS through aspects in planning and resource management of traditional applications.

Key words: Quality of service; Aspect Oriented Programming; processes; thread; Distributed systems; Parallel Systems; Use cases.

1. INTRODUCCION

Los sistemas operativos basados en el Kernel Linux son de interés por la posibilidad de configurar y personalizar su comportamiento. El planificador de procesos Linux reconoce dos clases de tareas: **tareas de tiempo real y tareas comunes.** Las tareas se diferencian por el tratamiento que reciben, así una aplicación de tiempo real (TR) consta de un tratamiento privilegiado respecto a una aplicación del tipo tradicional. Esto se traduce en una mayor priorización y disponibilidad de recursos para las aplicaciones de TR. En cuanto a las aplicaciones tradicionales no se han incorporado mayores mecanismos para brindar garantías de QoS a las mismas.

Cumplir con las expectativas en la ejecución de las aplicaciones plantea satisfacer una serie de garantías para QoS. Proporcionar tales garantías de QoS en el contexto de un sistema operativo de propósito general multiprogramado es factible. Un camino posible es modificar el código base de las aplicaciones para que implementen los mecanismos para proporcionar QoS. Tal modificación no es sencilla y tiene consecuencias no deseadas como la invasión de código para el manejo de QoS en el código base de la aplicación. Esto último, genera problemas de mantenibilidad, reuso y flexibilidad en el código de la misma.

Por eso se presenta un estudio que busca identificar y analizar los requerimientos de QoS desde un enfoque aspectual. Cuyo objetivo es proporcionar garantías de QoS para alcanzar los niveles de rendimiento deseados en el contexto de un sistema operativo de propósito general multiprogramado. El enfoque aspectual provee una alternativa para realizar los cambios necesarios sin requerir la reescritura sustancial del código. Además, el enfoque aspectual tiene como ventaja el poder introducir nuevos comportamientos al código original sin necesidad de modificar su estructura base y sobre todo evitando mezclar funciones adicionales con el objetivo del programa. De esta manera se mantienen encapsuladas las funciones que no conforman la funcionalidad principal del sistema o lógica de negocio; conocidas como c*rosscutting concerns* (Jackson A. y otros, 2009)(Schauerhuber A. y otros, 2006). Previendo así las dificultades de mantenibilidad, reuso y flexibilidad por reescritura y/o modificación de código.

Para poder dimensionar las garantías de QoS necesarias, en la primera Sección se explican conceptos relacionados a la QoS y garantías típicas sobre las mismas. El comportamiento de las garantías de QoS en un sistema influyen en su efectividad y rendimiento. Por lo que en la misma Sección se analizan métricas en relación a las garantías de QoS.

En vista al desarrollo de un Framework que aporte garantías de QoS en la plataforma Linux en la Sección 2, se analiza un sistema operativo basado en ese Kernel para enumerar los mecanismos existentes para implementar la planificación y gestión de recursos. Además se describe conceptos de QoS en la planificación aplicada a servicios, Grid y Cloud.

Para comprender las principales características de la Programación Orientada a Aspectos (AOP) y su posible utilización en el modelado de requerimientos de QoS en una aplicación, la Sección 3 aborda conceptos básicos de ese enfoque.

La Sección 4 plantea una serie de casos de uso con escenarios genéricos sobre los cuáles aplicar las garantías de QoS utilizando un enfoque aspectual en el contexto de un sistema operativo de propósito general multiprogramado. Los casos de uso se centran en garantizar los requerimientos de QoS durante la ejecución de los procesos y usar el enfoque aspectual para introducir las modificaciones requeridas. Los casos de uso planteados utilizan aspectos y abarcan diferentes condiciones de ejecución de procesos.

1.1 La Calidad de Servicio (QoS)

En este apartado se estudiará el concepto de calidad de servicio (QoS), sus características y alcances, las condiciones para lograr una buena implementación en un sistema paralelo y/o distribuido. Además se definen conceptos relacionados a las garantías de calidad como las métricas para QoS en distintos entornos.

Desde el punto de vista del usuario, se dice que la percepción de QoS en general esta asociada a los requerimientos que se deben satisfacer para cumplir con las expectativas sobre la ejecución de las aplicaciones (Xiao X., 2008), servicios y redes. En otras palabras la incorporación de QoS se realiza para obtener una percepción satisfactoria de aplicaciones, servicio o redes.

Una definición viable para la QoS en principio debería hacer distinción entre aquella que es aplicada al ámbito de las comunicaciones para integrar componentes y/o servicios sobre una red, y aquella que es aplicada sobre la gestión de recursos (Hyden E.A., 1994).

En el ámbito de las redes de comunicación LAN y WAN la QoS esta asociada al manejo apropiado del tráfico de red, tal que se garantice la entrega de los paquetes de datos oportunamente. En ese sentido las redes de ordenadores están empezando a ofrecer garantías de calidad de servicio con respecto al retardo de paquetes y el ancho de banda de conexión. Estas garantías de QoS son de poca utilidad si no pueden extenderse a las aplicaciones que se ejecutan en los puntos finales (Bruno J. et al., 1998).

Por otro lado, dentro del modelo de calidad de servicios basado en asignación de recursos, la QoS aparece en formas diferentes como parte de las interfaces entre las capas de un sistema. En la interfaz entre una aplicación y el sistema operativo, el sistema se ve como un proveedor de servicios y la aplicación se considera como un cliente del servicio. La aplicación puede entonces solicitar una determinada QoS al sistema operativo (Hyden E.A., 1994). En otras palabras, el sistema operativo debe tener la capacidad de suministrar recursos del sistema a las solicitudes de manera que se logren los niveles deseados de rendimiento de forma predecible, por ejemplo para múltiples aplicaciones multimedia (Bruno J. et al., 1998). En términos generales los recursos (memoria caché, memoria, tiempo de procesamiento, dispositivos de entrada y/o salida, bufferes) disponibles del host y su uso eficiente determinan

el éxito al cumplir los requerimientos de los procesos que demandan un cierto nivel de QoS. Los sistemas operativos con multiprogramación de procesos en general no ofrecen garantías de calidad de servicio ya que el rendimiento de una sola aplicación es, en parte, determinada por la carga total en el sistema (Bruno J. et al., 1998). Los sistemas operativos en tiempo real son capaces de entregar garantías de rendimiento, tales como demoras límite, pero requieren que las aplicaciones se puedan modificar para aprovechar el tiempo real (Bruno J. et al., 1998). Algunos enfoques existentes orientados a la aplicación tratan de mantener información sobre las demandas de la aplicación y realizar la reserva de recursos en el sistema final basada en el cálculo detrás de la necesidad (Bechler M., Ritter H., 2001)(Nahrstedt K. et al., 1998) (Wolf L. C., Steinmetz R., 1997).

La aplicación del enfoque por asignación de los recursos puede estar estrictamente orientada al usuario. Donde el uso compartido de los recursos en el sistema final y el sesgo de la distribución de los mismos se debe hacer en favor de las aplicaciones en las cuáles el usuario se centra actualmente (Bechler M., Ritter H., 2001). Al considerar como las aplicaciones manejan los recursos disponibles, es posible diferenciar a las a*plicaciones adaptativas* que tratan de adaptarse a la situación actual, pero no pueden controlar activamente los recursos y las *aplicaciones proactivas* que participan activamente sobre el efecto de la distribución de los recursos por adelantado (Bechler M., Ritter H., 2001).

Las plataformas con chip multiprocesador (CMP) son capaces de manejar múltiples hilos de ejecución sobre los núcleos, proveen una forma de incrementar la carga de trabajo simultáneo. La Figura 1 muestra los posibles escenarios en plataformas CMP, bajo multiprogramación, virtualización o arquitecturas heterogéneas. La introducción de la virtualización permite consolidar la carga de trabajo sobre una misma plataforma, además permite controlar el número de núcleos asignados a cada carga de trabajo. Aunque el soporte de hardware o software de hoy en día para controlar la asignación de recursos de la plataforma, tales como espacio de caché y ancho de banda de memoria para las cargas de trabajo individuales no está garantizado (Iyer R. et al., 2007).

La priorización de los diferentes recursos del sistema puede ser crítica a la hora de hablar de gestión eficiente. Dada las condiciones del sistema, establecer prioridades a la hora de asignar recursos garantizaría cumplir con las metas de QoS, siendo factible una distribución no equitativa asumiendo un detrimento de otros procesos en vista de mejorar el desempeño de la aplicación y/o sistema global (Iyer R. et al., 2007). La definición de las políticas de calidad de servicio varían en términos de los objetivos para cumplir con las condiciones necesarias de los requerimientos de QoS de la aplicación.

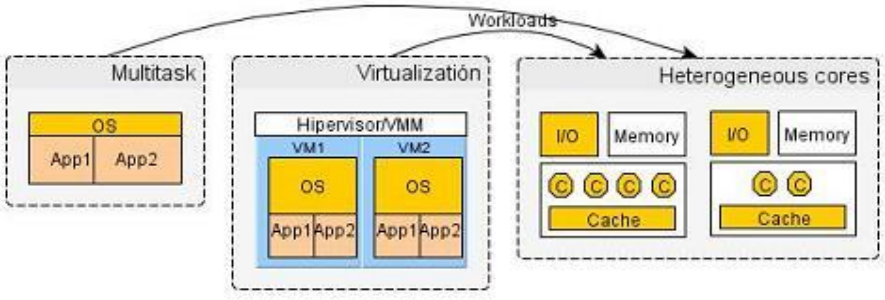

Figura 1: Modelo de sistema Workloads para CMP

La provisión de QoS mediante software combina las técnicas del sistema operativo para proporcionar un control estricto sobre la utilización de los recursos (por ejemplo, el uso del procesador o disco ancho de banda), y las técnicas que están más cerca de la capa de aplicación para satisfacer los requisitos de QoS que son más importantes para los clientes. En general, los enfoques de software existentes solo ofrecen garantías en el ámbito de una sola máquina o para una aplicación individual, y no proporcionan garantías de calidad a nivel de servicio global del sistema (Blanquer J., Batchelli A., 2004).

En el caso de las aplicaciones científicas, el procesamiento paralelo y distribuido toma importancia en situaciones donde un mayor poder computo es requerido. Como cuando la experimentación requiere la exploración de múltiples escenarios independientes. Así el apoyo a la QoS se vuelve importante en técnicas como la exploración de espacio de estado (barridos de parámetros completos), búsqueda guiada (por medio de técnicas de optimización no lineal) y el diseño de esquemas de experimentos que reducen el espacio de búsqueda. Una característica presente en algunas aplicaciones e-science que trabajan con adquisición y análisis de datos es que la carga de trabajo suele fluctuar por lo que el sistema debe adaptarse a las nuevas condiciones del entorno para poder cumplir con los requerimientos de QoS de la aplicación.

Normalmente los trabajos relacionados a e-science pueden ser encarados mediante la creación de múltiples procesos, y su posterior recolección en unidades de cómputo distribuidos (potencialmente) a través de un Grid o Cloud. Un problema asociado al uso de plataformas Grid es la falta de calidad de las garantías de servicio implicando que es difícil saber cuánto de los recursos se puede entregar, por lo tanto los tiempos de finalización del cómputo para experimentos son difíciles de determinar (Abramson D., 2011).

Existen enfoques para el manejo de QoS para servicios de Internet. Uno de ellos propone la utilización de reglas para la admisión en la entrada y conformación del tráfico. En particular, este enfoque propone considerar como una *caja negra* el cluster y los servicios, y utilizar técnicas de retroalimentación para controlar dinámicamente como cada una de las peticiones de los clientes debe ser enviada en el clúster (Blanquer J., Batchelli A., 2004). El modelo se presenta en la Figura 2 en el cual los clientes generan solicitudes, hacia un punto de acceso del sitio y dado un caso de sobrecarga u otro inconveniente las solicitudes pudieran ser descartadas antes de generar una respuesta exitosa o alcanzada su finalización (por tanto, no son devueltas al cliente). La ventaja presente en la caja negra es que no se requiere reconfiguración de hardware o reprogramación del software, para su implementación y aun así garantiza QoS efectivas para los servicios de Internet.

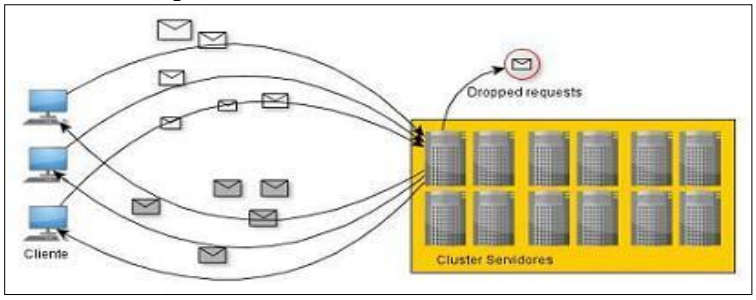

Figura 2: Modelo de sistema para servicios de Internet

Bajo el esquema anterior, Blanquer y Batchelli (2004) definen una clase de QoS como una tupla con tres cantidades: *el rendimiento garantizado, requisitos de cómputo para cumplir con una solicitud, y el tiempo de respuesta máximo permitido*. Así, una clase de QoS

describiría a la calidad predefinida para ser aplicada a una clase de servicio particular, cada clase se corresponde a una agrupación de solicitudes de acuerdo con una combinación de servicio (por ejemplo, URL, protocolo, puerto, etc.) o al perfil del cliente. La suma de estás clases conformarían la política de QoS.

Un problema asociado al uso de Grid o Cloud es la heterogeneidad de las arquitecturas, en los últimos tiempos los esquemas de virtualización han simplificado radicalmente el despliegue de las aplicaciones sobre estos entornos, haciendo posible ejecutar una variedad de aplicaciones que utilizan pilas de software específicas (Abramson D.,2011).

Del lado de las comunicaciones el modelo básico de servicio se refiere casi exclusivamente al tiempo de entrega de los paquetes. Los compromisos de QoS se hacen en base a los retrasos sufridos por los paquetes. Los llamados *servicios integrados* (IntServ), establecen que las aplicaciones necesitan interactuar de forma explícita con los routers de la red estableciendo reservas individuales (Flow Specs) por medio de un protocolo de señalización (RSVP) (IEFT RFC 1633). La arquitectura de *servicios diferenciados* (DiffServ), por el contrario, se ocupa de la escalabilidad, proporcionando así garantías de cumplimiento a los agregados de los flujos.

1.2. Garantías QoS

En general, las políticas de planificación realizables requieren que se corra a lo sumo un proceso a la vez sobre un recurso. Esto significa que si hay más de un proceso en espera para ejecutarse en el recurso, entonces uno o ambos de los procesos experimentarán (cola) un retardo (Bruno J. et al., 1998). La forma en que se ejecutan las aplicaciones impacta sobre los tiempos de procesamiento, así ejecutar una aplicación sobre un entorno dedicado versus uno compartido con otras aplicaciones de menor prioridad genera resultados dispares.

Para determinar o especificar las garantías de QoS es necesario establecer parámetros para medir su efectividad y rendimiento sobre la asignación de recursos. El *retraso límite* y la *equidad* son bien conocidos, y muchos algoritmos de planificación no proporcionarles los proveen (Bruno J. et al., 1998). Otro parámetro posible es el *servicio acumulativo* (Bruno J. et al., 1998).

El parámetro de *equidad* mide la capacidad del sistema para garantizar que los procesos "compartirán" los recursos en proporción a sus solicitudes, con independencia de su uso previo del recurso (Demers A. et al., 1989). Una política de planificación equitativa no penaliza a un proceso que utiliza un recurso inactivo más allá de sus solicitudes cuando otros dominios marcan como ocupado al recurso. La definición de la equidad se basa en otro modelo de servicio idealizado llamado *intercambio de procesador* (Goyal P. et al., 1996) Bajo el intercambio de procesador, cada proceso recibe un servicio proporcional a su fracción sobre un recurso.

El parámetro *retraso límite*, es proporcionado por una política de planificación. Donde el *tiempo de espera real* más *tiempo de servicio* para completar una fase definen una cantidad de tiempo constante como; $\boldsymbol{d}/\boldsymbol{\alpha_j}$ donde $\boldsymbol{d}$ es la duración de la fase y $\boldsymbol{\alpha_j}$ la fracción de un recurso asignado.

La *garantía de servicio acumulado*, complementa otros parámetros de QoS como el *retardo* y la *equidad*, implica que un proceso que está compitiendo por los recursos se ejecutará a una velocidad predecible, determinada por la fracción de los recursos que se reservan para ese

proceso, independientemente de la intensidad de la competencia por el recurso. La *garantía de servicio acumulado* significa que los retrasos de programación encontrados por un proceso en varios recursos no se acumulan durante la vida útil del proceso (Bruno J. et al., 1998).

Iyer at el. (Iyer R. et al., 2007), propone una serie de métricas para determinar la eficacia de una política de QoS basada en prioridades (aplicaciones de alta y baja prioridad): RUM (Resource Usage Metrics), RPM (Resource Performance Metrics) y OPM (Overall Performance metrics). RUM mide QoS de recursos asignados o en uso, así como su contribución a la calidad de servicio en general (manejo de múltiples recursos), RPM se centra en el rendimiento de los recursos proporcionados (no siempre se obtienen mejoras al proporcionar más recursos). Por último, OPM mide el rendimiento general, toma en cuenta que la contribución y rendimiento de un determinado recurso en una plataforma global depende altamente de características de la plataforma, así como características de la aplicación. En general, el rendimiento esperado para las aplicaciones de alta prioridad estará limitado por su participación en los modos dedicado y compartido de ejecución. En cambio para las aplicaciones de baja prioridad es necesario establecer restricciones mínimas a su performance en modo compartido.

En general, el rendimiento y los tiempos de respuesta son comúnmente expresados utilizando percentiles o promedios que se deben garantizar por intervalos de tiempo. Así las garantías pudieran ser temporales mientras dure el cómputo del proceso o extenderse más allá de la vida del proceso, estando predefinidas en el sistema. En el último caso, el sistema provee QoS para diferentes tipos de procesos o servicios; por lo tanto realizaría una planificación centrada en cumplir con los requerimientos de servicio de los procesos, en otras palabras lograr el cómputo de los procesos que demandan QoS a tasas aceptables. Para ello se espera que si la disponibilidad de recursos es suficiente el procesamiento se dé dentro los parámetros definidos en las políticas de QoS. Por el contrario, bajo condiciones extremas o de sobrecarga donde la disponibilidad de recursos es insuficiente, la eficacia para cumplir con las garantías de QoS se degradaría.

Una métrica que es posible definir es la *capacidad* (potencia de cálculo requerida). Está asocia rendimiento con los requisitos de cómputo, tal que la cantidad de requerimientos en un instante dado por la demanda de tiempo de cómputo para cada uno definirán la capacidad demandada (Blanquer J., Batchelli A., 2004) para el sistema. La capacidad es aplicable a sistemas de multiprocesadores como pueden ser un Cluster, Grid o Cloud.

En el ámbito de las redes de comunicación LAN y WAN el Protocolo de Internet de Calidad de Servicio (QoS IP) posee parámetros cuantificables que pueden ser aplicados para determinar la disponibilidad del servicio. Por ejemplo, la pérdida de paquetes pesados tiene un gran impacto, especialmente en la calidad de vídeo. Los parámetros básicos para las métricas de rendimiento son (Ahmed M. et al., 2007):
Retardo o latencia (Delay): Retardo de tránsito de extremo a extremo, es el tiempo transcurrido para que un paquete pase desde el emisor a través de la red al receptor.
Variación (Jitter): La variación del retardo de tránsito de extremo a extremo. En las redes de conmutación de paquetes jitter define la distorsión de los tiempos de llegada entre paquetes en comparación con los tiempos entre paquetes de la transmisión.
Rendimiento (Throughput): Medida de la velocidad a la que los datos pueden ser enviados a través de la red.

Tasa de pérdida de paquetes: Normalmente, cuando un camino a un dispositivo está sobrecargado y no puede aceptar datos de entrada en un momento dado se produce la pérdida de paquetes temporal.

Para garantizar QoS se requiere que el software de comunicación en combinación con el middleware logren un alto rendimiento y una baja latencia en el procesamiento de paquetes y expedición de los mismos. Cabe aclarar que parte de los factores que influyen en la capacidad del sistema para cumplir con las garantías de QoS están relacionados a condiciones externas del entorno operativo.

A medida que se aceleran las nuevas demandas en la comunicación multimedia a través de la transmisión de vídeo se hace necesario mejorar la provisión de QoS. En esa dirección, Internet Engineering Task Force (IETF) expresó la medición de calidad de servicio de red a través de indicadores llamados Indices de entrega de Medios (MDI) (IETF RFC 4445). MDI describe una serie de medidas que pueden ser utilizadas para evaluarla prestación del streaming para medios de comunicación y otras aplicaciones sensibles al tiempo y a la pérdida de paquetes. MDI consiste en dos componentes: El Factor de Retardo (DF) y la tasa de pérdida del medio MLR (Media Loss Rate). El factor de retardo es la máxima diferencia observada al final de cada paquete de flujos del medio, entre la llegada de datos y el despacho de datos al medio. La MLR es el número de los paquetes perdidos o flujo fuera de orden en un intervalo de tiempo seleccionado, donde los paquetes del flujo son los paquetes que llevan streaming de información de la aplicación. El rendimiento de una aplicación que reproduce una señal se mide a lo largo de dos dimensiones: la latencia y la fidelidad (IEFT RFC 1633).

Los requerimientos de QoS necesarios para aplicaciones IP están descriptos por La International Telecommunication Union (ITU), fijando objetivos de rendimiento para redes con aplicaciones comunes IP (Xiao X., 2008). En la Tabla 1 se describen las métricas comúnmente aplicadas para QoS.

Aplicación QoS	Métricas	Permite medir
Entorno de planificación compartido	Retraso limite Equidad Servicio Acumulativo	Efectividad y rendimiento Reserva de recursos para garantía de velocidad de procesamiento
Basada en prioridades	RUM RPM OPM	Grado eficacia de una política de planificación
En cluster, Grid o cloud	Capacidad	Grado de demanda de recursos a satisfacer
En Redes	Retardo o latencia Variación Rendimiento Tasa de pérdida de paquetes	Grado de funcionamiento de la red

Tabla 1: Métricas aplicables en QoS

La evaluación cuantitativa de métricas que poseen una interrelación para múltiples requerimientos de QoS puede ser difícil (Hoffert J. et al., 2010). En parte debido a las fluctuaciones que introducen los entornos dinámicos sobre los cuáles se ejecutan las aplicaciones.

2. PLANIFICADOR Y PROCESAMIENTO EN LINUX

En esta sección se estudiará la planificación en Linux, con vista de aplicar a los procesos condiciones de ejecución que promuevan las garantías necesarias para cumplir con requerimientos de QoS. Además se analiza la relación entre la asignación de recursos y sostenibilidad de las garantías de QoS. Por último, se describe en general algunas características de como se implementan controles para garantizar la QoS en sistemas basados en servicios, Cloud y Grid.

2.1. Planificación en sistemas Linux

GNU-Linux o conocido popularmente solamente como Linux es un sistema operativo de propósito general, que surgió en su momento como una alternativa al software privativo o propietario. Las posibilidades de configuración y personalización de su comportamiento lo transforman en una excelente plataforma para realizar experimentos con elementos tales como el planificador y/o la gestión de recursos.

El kernel de Linux tiene una estructura arquitectónica (Jones T., 2007) compuesta por los siguientes subsistemas, una interfaz para llamadas al sistema operativo (SCI System Call Interface), Manejo de Procesos (PM Process Management), Gestión de memoria (MM Memory management), Sistema virtual de archivos (VFS Virtual File System), Pila de red (Network Stack), Controladores de dispositivos (Device Drivers). En la Figura 3, se presenta un esquema que describe cada las principales funciones de cada uno de estos subsistemas y/o componentes.

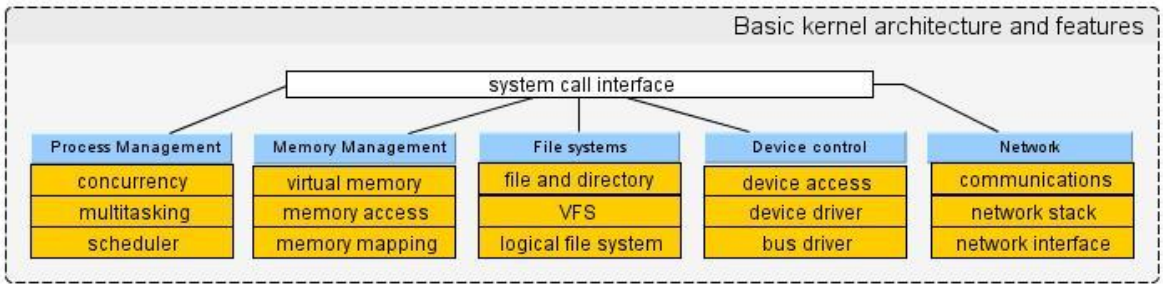

Figura 3: Modelo Kernel Linux

El PM tiene asignado el manejo de la concurrencia de los procesos, la gestión del planificador de procesos y provisión de mecanismos multitarea para compartir la CPU. La MM controla la operaciones que involucran accesos a la memoria, además provee manejo a esquemas de memoria virtual y su correspondiente mapeo. El VFS provee tanto soporte físico como lógico al sistema de archivos. Los controladores de dispositivo permiten manejar las entradas y salidas hacia los dispositivos y controlar los buses del sistema. La interfaz de llamadas del sistema provee un mecanismo para que las aplicaciones soliciten servicio al sistema operativo. Por último, el componente de comunicaciones permite vincular al host con la red subyacente. El kernel tiene código dependiente de la plataforma y una mayor parte que es independiente. El código dependiente es utilizado para los elementos que deben obtener una operación normal y eficiente sobre cierto hardware.

Dentro del núcleo de Linux, los mecanismos para la programación de los recursos de la CPU disponibles para los procesos que se ejecutan están optimizados principalmente para obtener alta eficacia, equidad, rendimiento y tiempos de respuesta bajos. Sin embargo, la calidad de los aspectos de servicio no fueron considerados para el desarrollo del planificador (Bechler M.,

Ritter H., 2001).Los tiempos del CPU son divididos en partes denominadas quantum, una para cada proceso ejecutable. Si un proceso que está en ejecución no llega a terminar en su porción de tiempo o el quantum expira, este es intercambiado por otro en su lugar. Las políticas del planificador asocian a los procesos una prioridad (valor) que determina su tratamiento a la hora de obtener tiempo de CPU (Bovet D. y Cesati M., 2006)(Bechler M., Ritter H., 2001). Siendo el manejo de prioridades dinámico, ajustando periódicamente las prioridades de los procesos.

En general, en Linux una aplicación que se va a ejecutar no dispone de un mecanismo de calidad de servicio asociado a la planificación de sus preferencias. La introducción de políticas de QoS para la planificación de los procesos puede tener dos aristas, ser estáticas o ser dinámicas. Las políticas de asignación de recursos de hardware pudieran ser estáticas, pero aún es posible que la política de priorización seguida por el software sea dinámica (Iyer R. et al., 2007). Un ejemplo, es que llegado a un umbral de ejecución se reajuste la priorización para favorecer el trabajo cooperativo con otros procesos. Las políticas dinámicas implican el monitoreo a intervalos constantes de las condiciones del entorno para poder adaptar la ejecución y/o asignación de recursos a los objetivos y restricciones de QoS.

Otro aspecto que influye en el comportamiento del planificador es sí el rendimiento de la tarea está asociado a los tiempos de espera por utilización de dispositivos de entrada y salida (I/O bound) o en su defecto por un uso intensivo de la CPU (CPU bound) (Bovet D. y Cesati M., 2006). El planificador Linux reconoce dos clases de tareas: **tareas de tiempo real y tareas comunes.** En comparación con las tareas comunes, las tareas en tiempo real son favorecidas principalmente por el planificador para hacer uso intensivo del CPU (Bechler M., Ritter H., 2001). El planificador de Linux soporta una larga serie de llamadas al sistema que permiten a los programadores cambiar las prioridades del planificador (Linux Man pages). En la Tabla 2 se describen las principales llamadas asociadas a la planificación de procesos.

Llamada al sistema	Descripción
nice()	Cambia la prioridad estática de procesos
renice()	Cambia la prioridad estática de procesos una vez en ejecución
getpriority()	Obtiene la máxima prioridad estática para un grupo de procesos
setpriority()	Establece la máxima prioridad estática para un grupo de procesos
sched_getscheduler()	Obtiene la política de planificación de un proceso
sched_setscheduler()	Establece la política de planificación y prioridad de tiempo-real de un proceso
sched_getparam()	Obtiene la prioridad de tiempo-real de un proceso
sched_setparam()	Establece la prioridad de tiempo-real de un proceso
sched_yiled()	El procesador es abandonado voluntariamente sin bloqueo
sched_get_priority_min()	Obtiene el valor de la mínima prioridad de tiempo-real para una política
sched_get_priority_max()	Obtiene el valor de la máximo prioridad de tiempo-real para una política
sched_rr_get_interval()	Obtiene el valor del quantum para la política Round Robin
sched_setaffinity()	Determina la máscara de afinidad a la CPU de un proceso
sched_getaffinity()	Obtiene la máscara de afinidad a la CPU de un proceso

Tabla 2: Llamadas del sistema al planificador

Los algoritmos para la planificación en Linux son de una de las siguientes clases: SCHED_FIFO para procesos de tiempo real bajo planificación First-In, First-Out (según su posición en la cola de ejecución) y SCHED_RR para procesos de tiempo-real bajo planificación Round Robin (basado en asignar tiempo en CPU) o SCHED_NORMAL para procesos estándar de tiempo compartido. Como se mencionó anteriormente cada proceso estándar tiene una prioridad estática asociada. Permitiendo establecer al planificador la priorización respecto a otros procesos estándar, los valores típicos se expresan en la Tabla 3.

El valor de la llamada del sistema *Nice* afecta al planificador de procesos de sistemas Linux permitiendo modificar prioridad estática de un proceso (Linux Man pages). Su valor oscila entre los -20 a 19. Un valor de *Nice* próximo al valor 19 implica que el proceso obtiene una menor asignación de tiempo de procesamiento o quantum y por el contrario si el valor esta próximo al valor -20 el proceso obtiene un mayor quantum. Es posible establecer una relación entre la prioridad y el quantum, tal que se da que a mayor prioridad mayor es el quantum y viceversa. Esto es representado por un valor diferente de tiempo base del quantum, el que es calculado por una formula en base a la prioridad estática.

Descripción	Prioridad estática	Valor Nice
Muy alta prioridad estática	100	-20
Alta prioridad estática	110	-10
Por defecto	120	0
Baja prioridad estática	130	+10
Muy Baja prioridad estática	139	+19

Tabla 3: Valores típicos de prioridad estática

Ahora bien, la utilización de prioridades implica que debe evitarse que los procesos con mayor prioridad estática bloqueen por completo a los de menor prioridad. Dado que generaría una situación de desequilibrio que llevaría a la inanición de los procesos de menor prioridad estática. Para ello el planificador usa una estrategia que da trato diferenciado a dos tipos de procesos: los procesos activos y los procesos expirados. Los procesos activos aún no han agotado su quantum de tiempo por lo que se les permite correr. Los procesos expirados agotaron su quantum y deben esperar a que todos los procesos activos expiren para volver a ejecutarse. El tratamiento que realiza el planificador de los procesos interactivos difiere de los procesos por lote. En general, un proceso por lote una vez agotado su quantum de tiempo pasará a estar expirado, mientras que una interactivo por lo general sigue siendo activo, se asigna un nuevo quantum y solo ante un proceso de mayor prioridad estática o una operación con tiempo de espera elevado se lo pasa a expirado.

2.2 Recursos de CPU

Para el sistema el tiempo de procesamiento es en sí un recurso que se asigna a cada proceso activo. Existen otros recursos que son asociados a un proceso en el sistema tales como el espacio de direcciones, pila, etc. Cada recurso tiene asociado un límite blando y uno duro. El límite blando es el valor que el Kernel de Linux utiliza como valor de referencia y lo hace cumplir para cada recurso, mientras que él límite duro actúa como tope para límite blando (Linux Man pages).

Cada proceso tiene asociado un conjunto de límites de recursos los que especifican la cantidad de recursos del sistema que pueden utilizar. Los límite de recursos de un proceso activo están almacenados dentro del descriptor de señales del proceso en el campo current → signal → rlim (Bovet D. y Cesati M., 2006). Es posible definir el tamaño máximo del espacio de direcciones del proceso, tamaño del archivo de volcado de memoria, tamaño máximo de la pila, tiempos máximos de CPU para el proceso, etc.

Un sistema Unix tradicional con el objeto de chequear permisos implementa el manejo de capacidades que pueden ser activadas o desactivadas independientemente. Se distingue entre dos categorías procesos privilegiados y procesos sin privilegios. Un proceso privilegiado tiene una identificación de usuario UID igual a 0 (superusuario o root) y uno sin privilegios tienen un UID distinto a cero (Linux Man pages). Los hilos de ejecución (threads) poseen capacidades que se agrupan en tres conjuntos que a su vez contienen 0 o más capacidades (Linux Man pages). El conjunto *Permitted* es el superconjunto que contiene y limita las capacidades que puede asumir el proceso. Además este conjunto limita las capacidades que pueden incorporarse al conjunto heredable de un thread que carece de la capacidad CAP_SETCAP en su conjunto efectivo. El conjunto *Inheritable* posee capacidades que son preservadas a través de una llamada *execve.* El conjunto *Effective* es el conjunto utilizado por el kernel para comprobar permisos para el thread.

Llamada al sistema	Descripción
getrlimit()	Obtiene límite de recursos
setrlimit()	Cambia límite de recursos
prlimit()	Combina y extiende la funcionabilidad de setrlimit y getrlimit

Tabla 4: Llamadas al sistema límite de recursos

Se mencionó que cada recurso está asociado a un límite blando y uno duro. Los procesos sin privilegios tienen configurado un límite blando entre el intervalo 0 y el limite duro, pudiendo modificar en forma irreversible su límite duro. Un proceso privilegiado (con la capacidad CAP_SYS_RESOURCE) puede modificar los límites arbitrariamente utilizando las llamadas correspondientes (Linux Man pages). En la Tabla 4 se describen las principales llamadas asociadas a la gestión de recursos. Los recursos se corresponden con alguno de los siguientes tipos (Linux Man pages) (Tabla 5):

Tipo de recurso	Descripción
RLIMIT_AS	Tamaño máximo de la memoria virtual del proceso (espacio de direcciones) en bytes.
RLIMIT_CORE	Tamaño máximo del archivo *core.*
RLIMIT_CPU	Tiempo limite de CPU en segundos.
RLIMIT_DATA	Tamaño máximo del segmento de datos del proceso.
RLIMIT_FSIZE	Tamaño máximo de archivo que el proceso puede crear.
RLIMIT_LOCKS	Un número sobre la combinación entre *flocks* y *fcntl* que puede obtener el proceso.
RLIMIT_MEMLOCK	Número máximo de bytes de memoria que pueden ser bloqueados en la memoria RAM.
RLIMIT_MSGQUEUE	Específica el límite en bytes asignables a colas de mensaje POSIX para el ID de usuario del proceso invocador.

RLIMIT_NICE	Especifica el límite máximo al que el valor *nice* del proceso puede ser elevado usando *setpriority* o *nice*.
RLIMIT_NOLIMIT	Especifica un valor mayor que el número máximo de descriptor de archivo que puede abrir el proceso.
RLIMIT_NPROC	Número máximo de procesos (threads en Linux) que se pueden crear para el ID de usuario real del proceso invocador.
RLIMIT_RSS	Establece el límite (en páginas) del conjunto de trabajo del proceso en memoria RAM.
RLIMIT_RTPRIO	Especifica un límite máximo a la prioridad de tiempo real que puede ser establecida para el proceso mediante *sched_setscheduler* y *sched_setparam*.
RLIMIT_RTTIME	Fija un límite en microsegundos de consumo de tiempo de CPU para un proceso bajo planificación de tiempo real sin hacer una llamada al sistema de bloqueo.
RLIMIT_SIGPENDING	Especifica el límite en el número de señales que pueden ser encoladas por el ID de usuario real del proceso invocador. Combina la cantidad de señales estándar y en tiempo real con el propósito de comprobar este límite.
RLIMIT_STACK	Tamaño máximo de la pila del proceso, en bytes. Al llegar a este límite, se genera una señal SIGSEGV, el proceso debe emplear una pila alternativa de señales.

Tabla 5: Tipos del argumento RLIMIT

Parte de las soluciones para dar soporte a la QoS en aplicaciones multimedia se basan en utilizar en el planificador prioridades y reservas. El planificador debe procurar la entrega oportuna de eventos y su tratamiento por parte de la aplicación (Poellabauer C. y Schwan K., 2005). Para evitar introducir retrasos innecesarios con lleven a una pérdida de rendimiento.

2.3 Planificación de servicios web

La planificación a nivel de servicios básicamente se centra en lograr satisfacer las solicitudes al ritmo que llegan. Para ello los estándares de la Internet Engineering Task Force (IETF) requieren de una adecuada configuración de la red para proporcionar la garantía de rendimiento necesario a las aplicaciones y/o servicios. Las aplicaciones distribuidas deben aprovechar las capacidades de las redes que soportan QoS. Para ello las aplicaciones deben interactuar con los servicios de calidad subyacentes en la red para negociar parámetros de QoS (Canonico R. y D'Arienzo M., 2002). Los paquetes de la comunicación pasan por un protocolo de señalización como RSVP (IEFT RFC 2205) o NSIS (IEFT RFC 5974), la señalización permite dar un tratamiento adecuado a cada paquete. De esta forma es posible proveer mecanismos de comunicación que puedan abordar los requerimientos de QoS de las aplicaciones distribuidas.

Existe una diferencia esencial entre como las aplicaciones ven afectado su rendimiento ante los retrasos en la entrega de paquetes. Los límites máximo y mínimo para las aplicaciones de tiempo real implican que los datos deben estar dentro de esos plazos o por el contrario pierden su valor. Otras aplicaciones llamadas "elásticas" siempre esperan por la llegada de los datos (IEFT RFC 1633). Las aplicaciones de tiempo real que reproducen datos constan de un punto de reproducción que indica hasta donde puede compensar un retraso en la entrega de paquetes. Los datos que lleguen antes de ese punto podrán ser utilizados. Los retrasos de los paquetes individuales afectan la fidelidad de la reproducción; una aplicación puede o bien cambiar el retardo de compensación con el fin de reproducir los paquetes atrasados o simplemente descartar los paquetes (IEFT RFC 1633). Por el otro lado, las aplicaciones del tipo "elásticas"

siempre van a poder optar entre esperar recibir la información o seguir adelante sin parte de los datos, los retrasos van afectando al rendimiento de la aplicación. Las técnicas de buffering para mejorar el rendimiento a veces es posible en las aplicaciones. Las aplicaciones a la hora de implementar un modelo de servicio que resulte garantizado y predictivo hacen su propio equilibrio entre fidelidad, latencia y costo (IEFT RFC 1633). Un elemento que se hace necesario conocer es el estado de los flujos en los routers que forman parte activa del proceso de reserva (Canonico R. y D'Arienzo M., 2002).

Los mecanismos que aprovechan el Quorum, implementan algún tipo de consenso entre los participantes como una solución viable para generar un manejo planificado de las solicitudes de servicio. Por ejemplo, Blanquer y Batchelli (2004) plantean un motor de QoS al ingreso del cluster, este motor clasifica las solicitudes y las asocia a una política de QoS definida que asignaría una parte garantizada de los recursos disponibles a cada clase. Además, controla el flujo de carga de trabajos que él Quorum entrega al cluster y complementa este con la capacidad de abandonar los mismos en caso de sobrecarga o degradación de respuesta. El motor QoS en sí controla el número máximo de peticiones que pueden estar pendientes en cualquier momento.

Para garantizar los recursos bajo Quorum, se emplea *Request precedence*, que consiste en implementar una partición de los mismos, tal que cumpla los parámetros mínimos aún bajo condiciones de sobrecarga o mal comportamiento de otras clases.

El sistema operativo subyacente debe dar soporte a la integración de las aplicaciones distribuidas, tal que puedan alcanzar los requerimientos de QoS deseados. En general este soporte se logra mediante las bondades de la capa middleware (políticas y servicios) y protocolos de red habilitados para QoS. Los componentes middleware para soportar QoS en aplicaciones distribuidas de tiempo real (DRE) presentan muchas características deseables, sin embargo, históricamente carecían de la capacidad de asignar los recursos de manera eficiente y que el sistema pueda adaptarse a las fluctuaciones de la carga de trabajo de entrada, la disponibilidad de recursos y las condiciones de funcionamiento (Shankaran et al., 2008).

Las aplicaciones distribuidas basadas en servicios web poseen una fase de selección de los servicios web para conformar el sistema, esta fase implica su emparejamiento con los requerimientos funcionales y no funcionales, las características de QoS demandadas antes de la creación del sistema de software distribuido (Gallege L.S. et al., 2013). La selección está influenciada por el grado de confianza del servicio respecto a su comportamiento y grado de cumplimiento de las políticas de QoS estipuladas.

2.4 Cloud y GRID

La mayor parte de las plataformas de ejecución en Cloud y Grid tienen componentes heterogéneos, la eficiencia y previsibilidad varia de acuerdo a la tecnología de interconexión y la plataforma de cómputo. Tal como la configuración de los servicios del middleware para dar soporte a la QoS en tiempo real. En ámbitos operativos fluctuantes dicho soporte debe procurar realizar los ajustes necesarios para mantener un desempeño dentro de los parámetros de QoS seleccionados.

Existen soluciones como Nimrod que proveen QoS a nivel de tareas, para elegir la asignación más barata que cumpla un plazo especificado por el usuario (Abramson D., 2011) y tomando en

cuenta que la disponibilidad de recursos en Cloud y Grid es dinámica. Así que el esquema de aplicaciones MTC (Many Task Computing) puede obtener soporte a la QoS sobre plataformas que son heterogéneas. En general, se utiliza un portal para definir los trabajos, especificando su planificación dinámica y el manejo del retorno de los resultados. Una vez integrados los resultados estos serian interpretados.

La necesidad de cumplir con requisitos no funcionales es un factor que incrementa la complejidad del software. Estos requisitos engloban a los atributos de QoS, disponibilidad, manejo de excepciones, etc. La lógica tras la implementación de los requisitos no funcionales a menudo complica la integración del software por imponer control de flujo no lineales (Hiltunen M. et al., 2006). Para alcanzar retrasos de extremo a extremo adecuados en la comunicación, no basta con garantizar el correcto funcionamiento de una aplicación multimedia. A menudo las acciones tardías de una aplicación y/o middleware incrementan los tiempos respuesta a eventos (Poellabauer C. y Schwan K., 2005). Bajo esas condiciones la integración de una aplicación en Cloud o Grid podría generar retrasos en la devolución de trabajos y afectar la escalabilidad de los servicios afectados.

2.5 Planteo del problema

Las garantías de QoS permiten cumplir con las expectativas del usuario (desarrollador o del sistema) sobre el rendimiento de una aplicación. Para los sistemas basados en el kernel Linux se mencionó que el planificador reconoce dos clases de tareas: **tareas de tiempo real y tareas comunes.** Normalmente la incorporación de QoS en el planificador requiere modificar código e invadir el funcionamiento del sistema operativo y/o en las aplicaciones.

Esto en general no es tarea sencilla y puede introducir efectos indeseados en el funcionamiento normal del sistema operativo. Como inestabilidad e incapacidad del sistema para llegar a un equilibrio en la carga de trabajo. En las aplicaciones delimitar el alcance de las modificaciones presenta retos al desarrollador. La introducción de cambios en el código de la aplicación genera errores y fallas en ejecución en partes probadas de la aplicación por falta de contención de los cambios. La dispersión del código adicional puede generar problemas de mantenibilidad y comprensión del código original.

Se plantea que el tratamiento para tareas comunes dado por el planificador puede ser modificado sin tener que reescribir totalmente el mismo, utilizando un enfoque aspectual para la incorporación de QoS basada en recursos y los ajustes necesarios sobre la prioridad y tiempo de procesamiento para cambiar la administración provista por el planificador. De tal forma que el manejo de las funciones de QoS este contenida en aspectos, que por definición se encuentran separados del código principal.

El principal beneficio de este enfoque es que permite evitar en general la reescritura y modificación de código. Además el enfoque aspectual permite encapsular los requerimientos adicionales como la QoS evitando que las modificaciones se terminen mezclando y diseminando por toda la aplicación. Adicionalmente la contención del código provista por el encapsulamiento permite asegurar la mantenibilidad y entendimiento tanto del código para ofrecer las garantías de QoS como del código original de la aplicación.

3. CONCEPTOS DE AOP

La definición de un sistema puede ser expresada como un conjunto de objetivos a satisfacer. Normalmente estos objetivos responden a necesidades funcionales (RF) y a otras complementarias asociadas requerimientos no funcionales (RNF).

En un sistema un *concern* puede ser definido como un área de interés particular. En general, se debe implementar un conjunto de concerns para estructurar una aplicación completa. Algunos de estos concerns forman parte de la funcionalidad base asociada a RF (denominados core concerns). Por otro lado, aquellos que no forman parte de la funcionabilidad principal del sistema o lógica de negocio son conocidos como cr*osscutting concerns* (Jackson A. y otros, 2009)(Schauerhuber A. y otros, 2006) estos se vinculan a los RNF. Los crosscutting concerns aportan características adicionales deseables, en general asociadas a restricciones técnicas y/o de calidad. Pero su implementación con técnicas de programación convencionales origina ciertas desventajas como la dispersión, repetición y mezcla de código de RNF con código de RF del sistema. Esto termina originando código pobre o de mala calidad, la ruptura de la modularización, pérdida de encapsulamiento y un serio problema para delimitar los alcances de las tareas y/o objetivos de los propios módulos. Algunos ejemplos típicos de CCC son logging, monitoreo, validación de datos.

La Programación Orientada a Aspectos (AOP) (Kiczales G., 1996) procura modularizar el código transversal asociado a los crosscutting concerns que por definición no se corresponde al código de los core concerns. La AOP propone el concepto de *aspecto* para encapsular lógicamente los crosscutting concerns, en otras palabras implementan el código de los crosscutting concerns. Cabe aclarar que en general solo una parte de los crosscutting concerns identificados en una aplicación se transformaran en un aspecto. Un aspecto actuá como un módulo aportando al código relacionado a los crosscutting concerns flexibilidad, encapsulamiento y abstracción. Los componentes distintivos de un aspecto son: JoinPoint, PointCut y Advice.

Un *JointPoint* es un punto en la ejecución de un programa que se encuentra bien definido. Como puede ser la llamada a un método/constructor o el acceso a atributos de un objeto. El *JointPoint* señala donde el comportamiento del aspecto será integrado al código base. Un *JointPoint* puede referenciar una función, un atributo, un tipo, una variable, o un punto del código referenciado por el *JointPoint* (Spinczyk O. y Urban M., 2012).

Un *PointCut* es la descripción de un conjunto de *JointPoints*, que determina en que condiciones un aspecto tendrá efecto sobre el código funcional. La descripción de un *PointCut* esta compuesta por un designador de pointcut (call-execution u otro) y una expresión de coincidencia (patrón) que se emplea para cotejar la misma contra un conjunto esecifico de *JointPoint*. Así se determina el conjunto apropiado de *JoinPoint* sobre el cual el aspecto actúa.

El *advice (aviso)* se asocia a los pointcuts para permitir que se inserte el comportamiento (código) específico del aspecto. La activación del código *advice* acepta tres modificadores de comportamiento, antes *(before)*, después *(after)*, o antes y después *(around)*. Estos modificadores especifican donde el código de punto de unión sera alcanzado. Se activan durante la ejecución cuando el código del JointPoint es alcanzado.

La separación de aspectos implica que la parte del código vinculada a los *crosscutting concerns* estará implementada en un lenguaje AOP y la otra parte del código cuya la funcionalidad está vinculada a los *core concerns* expresada en un lenguaje base de característica funcional.
Un *Weaver* (tejedor) es el responsable de entrelazar el código del aspecto con el código escrito en el lenguaje de programación convencional. La integración de los módulos (clases y aspectos) efectuada mediante un tejedor de aspectos permite combinar las instrucciones de los lenguajes, tal que el flujo de ejecución entre el lenguaje base y el lenguaje AOP sea posible. Esta integración se resuelve tanto en tiempo de compilación como en tiempo de ejecución dependiendo tanto de las capacidades del lenguaje AOP como de la necesidad del contexto de ejecución de la aplicación. Para AspectC++ este entrelazado se efectúa de forma estática en tiempo de compilación. El tejedor transforma los aspectos en clases de C++. En cierto sentido los lenguajes AOP extienden las funciones del lenguaje base, por ejemplo: AspectJ extiende para Java o AspectC++ (Spinczyk O., 2005) extiende para C/C++ (ISO/IEC 9899)(ISO/IEC 14882:2014(E)).

4. QoS EN APLICACIONES LINUX CON ASPECTOS

Se presentan alternativas que modelan casos de uso para brindar garantías de QoS que buscan identificar y analizar dichos requerimientos desde un enfoque aspectual. Para representar los casos de uso se utiliza como base el método AOSD/UC (Bakker J. et al., 2005)(Jacobson I. y Pan-Wei Ng. (2004)), donde los diagramas de caso de uso representan separadamente los concerns. En el estudio se plantea una serie de escenarios que aplican el enfoque aspectual por medio de casos de uso para modelar los requerimientos de QoS en aplicaciones tradicionales Linux. Los escenarios propuestos siguen un tratamiento general para establecer las necesidades de QoS en los procesos presentes en las aplicaciones. Esto implica que en principio sería posible para un desarrollador incorporar los mismos a cualquier aplicación existente. Cada escenario aborda comportamientos diferentes de procesamiento y manejo de procesos adicionando características de complejidad particulares. En la Figura 4 se grafican los modelos para la ejecución de procesos de los escenarios planteados.

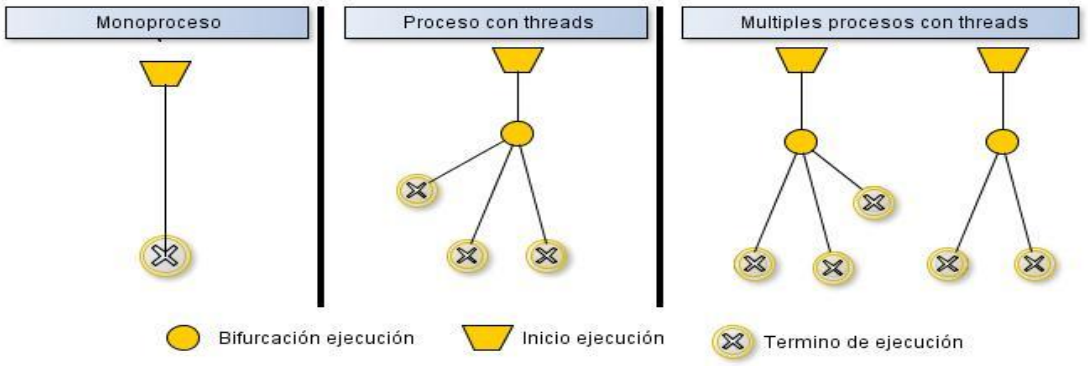

Figura 4: Modelos de procesamiento

Se establecen una serie de restricciones a los escenarios, primero la necesidad de disponer del código fuente para poder adaptar la aplicación y en segundo lugar en todos los casos el alcance de estos escenarios se limita a procesos residentes en un único host. Para llevar a cabo

la futura implementación del enfoque aspectual propuesto se introduce el concepto de módulo como contenedor de aspectos que permiten implementar las garantías de QoS. Estos módulos se integran a la aplicación mediante los JointPoint.

4.1 Escenario 1: Monoproceso simple

En este escenario se plantea una aplicación cuyo diseño se define mediante un proceso único en ejecución sin procesos hijos, residiendo la misma en un host. Entre sus requerimientos debe ser provista una cierta garantía de QoS para lograr su ejecución con éxito. En general, la expresión de estas garantías puede ser definida por medio de parámetros sobre los recursos disponibles tales como el tiempo de procesamiento, memoria, almacenamiento, etc. Para este escenario una priorización de la tarea por sobre la carga normal de trabajo es fundamental, esto implica garantizar las condiciones de procesamiento adecuado y acceso a los recursos requeridos por la aplicación. El suficiente acceso a otros recursos para cumplir con las necesidades propias de la aplicación es clave para evitar un rendimiento inadecuado asociado a bloqueos por recursos.

Para la definición de QoS se plantea declarar necesidades sobre los recursos del sistema, así en este escenario el tiempo de procesamiento necesario para cumplir con las expectativas esta expresado en un valor en función a un quantum y una prioridad en la política de planificación.

Si bien las aplicaciones TR cuentan con un tratamiento diferenciado en sistemas Linux no ocurre lo mismos con las aplicaciones tradicionales. Por lo tanto, priorizar los procesos de interés por sobre la carga de trabajo del sistema es parte de garantizar QoS para la aplicación. Los parámetros que determinan las condiciones iniciales del procesamiento con garantías de QoS deben estar definidos antes de la ejecución. En cualquiera de los casos los módulos que contienen los aspectos definidos para garantizar la QoS brindarían una solución que encapsula los *crosscutting concerns* relacionados con la QoS del código y la adecuación de su comportamiento según la definición de parámetros. La definición de las garantías de QoS pueden abarcar distintos segmentos de código de la aplicación, determinando donde se aplican o no las garantías de QoS. Se puede considerar que estos segmentos de código de la aplicación contienen determinados JointPoint que al ser alcanzados activan las garantías de QoS y eventualmente al alcanzar otro JointPoint determinado finalizamos la aplicación de las garantías de QoS.

En la Figura 5 se modeló el caso de uso correspondiente a este escenario. En principio el usuario o desarrollador establece y/o define requerimientos de QoS para la aplicación mediante la interface *QoS Request*. En tiempo de ejecución la definición de las garantías de QoS necesarias ajustan el rendimiento de la aplicación en forma dinámica. Estos requerimientos de QoS son interceptados por el aspecto *Ensure QoS* y este realiza las acciones tendientes a asegurar las condiciones de QoS. El comportamiento se da sobre el programa en determinados puntos de ejecución establecidos inicialmente al definir los requerimientos de QoS necesarios. Cuando el procesamiento de la aplicación no demanda garantías de QoS el aspecto *QoS release* efectuá acciones para liberar las condiciones de QoS. Otras acciones relacionadas al tratamiento de la solicitud QoS son informar cuando es posible o no atender a la misma generando a su posible planificación y/o rechazo.

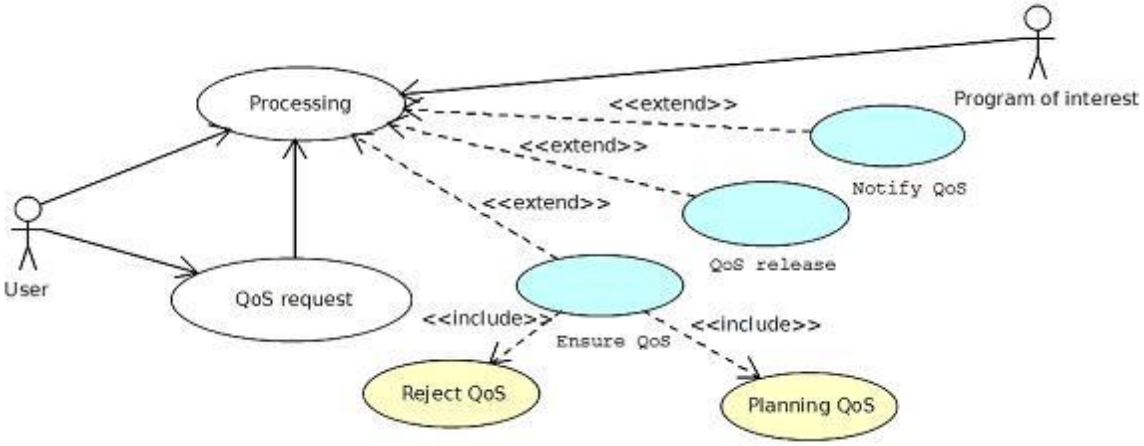

Figura 5: Caso de uso monoproceso simple para QoS en AOSD/UC

Especificación de Caso de uso: Procesamiento <Processing>
Este caso de uso describe como un *Programa de interés* es ejecutado
Flujo básico:
1. El *Usuario* inicia la ejecución de un *Programa de interés*
2. El *Programa de interés* inicia su ejecución normal
3. El *Programa de interés* avanza en su ejecución
4. El *Programa de interés* termina su ejecución
5. El caso de uso termina

Flujo alternativo:
En el paso 2 o 3 el *Usuario* o *desarrollador* realizó una petición de QoS

Especificación de Caso de uso: Petición QoS <QoS request>
Este caso de uso describe como se realiza una petición de QoS mediante la interface *QoS request.*
Flujo básico:
1. El Usuario o desarrollador realiza una petición de QoS mediante la interface *QoS request*, la que incluye parámetros para definir la QoS necesaria y como aplicarla. El parámetro *tiempo de procesamiento* está definido en base a un quantum y prioridad en la política de planificación. Los parámetros como memoria y almacenamiento están definidos en base a una reservación del recurso y tipo de acceso (exclusivo, semiexclusivo o no exclusivo). La aplicación puede ser global y/o en determinados segmentos de código (joinpoint inicial y final).
2. La petición de QoS es gestionada por el aspecto *Ensure QoS* para brindar las garantías de QoS requeridas.

Flujos alternativos
A1. En el paso 2, la gestión de QoS necesita garantizar las condiciones de QoS requeridas*:*
    1. Se invoca al módulo que contiene al aspecto <*Ensure QoS*>.
    2. Se procede con la ejecución del advice apropiado del aspecto.

A2. En el paso 2, la gestión de QoS modifica las condiciones de ejecución para retirar las garantías de QoS:

1. Se invoca al módulo que contiene al aspecto <*QoS Release*>.
2. Se procede con la ejecución del advice apropiado del aspecto.

A3. En el paso 2, la gestión de QoS notifica las acciones realizadas:

1. Se invoca al módulo que contiene al aspecto <*Notify QoS*>.
2. Se procede con la ejecución del advice apropiado del aspecto.

La Tabla 6 detalla los pasos de los Aspectos del caso de uso Monoproceso simple, para asegurar, liberar, notificar QoS para un escenario donde un proceso sin subprocesos hijos es ejecutado implementando gestión de QoS sobre el mismo en base a los parámetros de una solicitud.

Ensure QoS (Asegurar QoS)	
Pasos	
1	El usuario o desarrollador genera una solicitud de garantías de QoS para la ejecución de un determinado proceso, define parámetros necesarios (prioridad, tiempo de ejecución (quantum), política de acceso y reservación de recursos compartidos, acceso a computo de alto rendimiento, etc.) y los correspondientes segmentos de código donde se aplica, definidas por un joinpoint inicial y uno final. Dinámicamente durante la ejecución las condiciones de procesamiento se modifican provocando ajustes en las garantías provistas.
2	Se analizá los requerimientos de las garantías solicitadas, se determina la prioridad actual y la solicitada, la planificación de tiempo de ejecución (empleando por ejemplo: sched_getscheduler, schedsetscheduler, getpriority, setpriority, etc), la política de acceso a recursos (empleando getrlimit, setrlimit, etc) y sus posibles modificaciones. Una parte de estas tareas se realiza mediante llamadas al sistema citados en las Tablas 2 y 4.
3	Se toman acciones para proveer las garantías de QoS, estas incluyen planificar el proceso y acceso a recursos según los requerimientos. Si no fuera posible brindar las garantías de QoS se rechaza la solicitud.
QoS release (Liberar QoS)	
Pasos	
1	El programa solicita fin de las garantías de QoS por alcanzar un jointpoint que señala el final de la aplicación de las mismas en el segmento de código correspondiente.
2	Se analizá solicitud, se determinan los pasos para recuperan recursos comprometidos por las garantías de QoS.
3	Se realizan tareas para terminar las garantías de QoS. Se replanifica proceso y acceso a recursos a estado previo o en su defecto se recuperan los recursos. La relajación de las garantías de QoS altera las condiciones de funcionamiento del sistema.
Notify QoS (Notificar QoS)	
Pasos	
1	El cliente solicita información del programa respecto a las garantías de QoS brindadas. Al alcanzar un jointpoint inicial o final o al modificar planificación o priorización
2	Se vuelca la información a un archivo auxiliar y/o visualiza por pantalla.

Tabla 6: Descripción pasos de los Aspectos para monoproceso simple Asegurar, Liberar y Notificar QoS

4.2 Escenario 2: Threads

Se trata de una aplicación cuyo esquema de ejecución consta de un proceso padre que genera 1..n procesos hijos residentes en un host, conformando un árbol de procesos. Cabe aclarar que sí bien conforman un árbol de procesos no necesariamente cada proceso hijo efectúa la misma tarea que los otros, la generación de threads podría responder a una necesidad de efectuar una serie de tareas en forma concurrente. Así cada uno de los threads podría requerir condiciones de ejecución diferentes implicando la definición de garantías de QoS para cada uno de estos threads. Durante la ejecución del árbol de procesos la carga de trabajo del sistema y las garantías de QoS solicitadas varían. Normalmente los threads colaboran entre sí para cumplir un objetivo común, parte de esta colaboración implica coordinar las condiciones de ejecución del árbol de procesos. Sincronizar de alguna forma las garantías de QoS ofrecidas a los threads que conforman un mismo árbol de procesos permite satisfacer más fácilmente las expectativas.

El parámetro *tiempo de procesamiento* de este escenario esta definido en base a un quantum, prioridades en la política de planificación y en la carga de trabajo generada por el mismo árbol de procesos. Los parámetros asociados a la reservación de recursos y su tipo de acceso requieren estar acorde a las condiciones definidas mediante sincronización en la aplicación para no generar conflictos entre las garantías de QoS. En la Figura 6 se plantea un posible caso de uso correspondiente a este escenario. El usuario o desarrollador dispone de la interface *Request QoS* para establecer y/o definir requerimientos de QoS. En tiempo de ejecución basado en los parámetros que determinan las garantías de QoS necesarias ajusta el rendimiento de la aplicación.

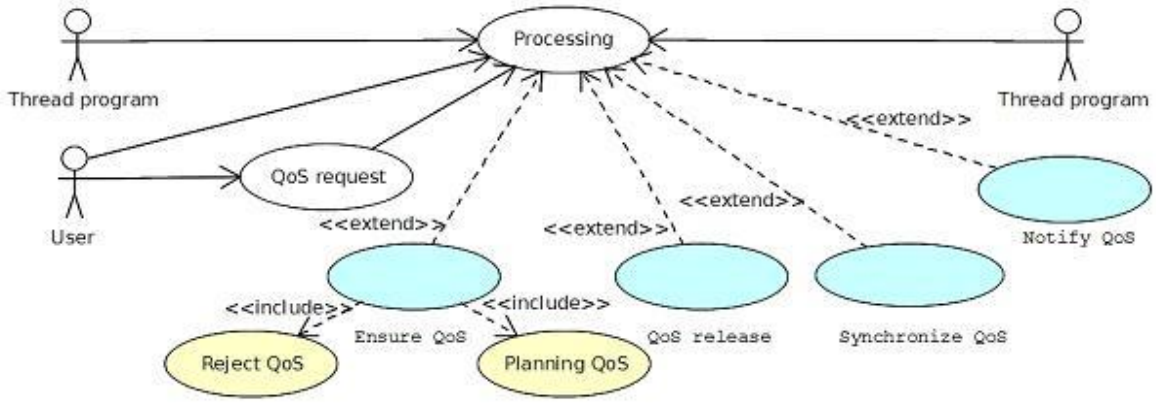

Figura 6: Caso de uso QoS para proceso con threads en AOSD/UC

Especificación de Caso de uso: Procesamiento <Processing>

Este caso de uso describe como un *Programa de interés* es ejecutado, este genera múltiples threads de ejecución.

Flujo básico:

1. El *Usuario* inicia la ejecución de un *Programa de interés*
2. El *Programa de interés* inicia su ejecución normal, generando múltiples threads de ejecución
3. Los *threads del programa* avanzan en su ejecución

4. Los *threads del programa* terminan su ejecución
5. El caso de uso termina

Flujo alternativo:

En el paso 2 o 3 el *Usuario o desarrollador* realizó una solicitud de *QoS*

Especificación de Caso de uso: Petición QoS <QoS request>

Este caso de uso describe como se realiza una petición de QoS mediante la interface *QoS request*.

Flujo básico:

1. El Usuario o desarrollador realiza una solicitud de QoS por medio de la interface *QoS request*, la que incluye parámetros para definir los requerimientos de QoS y donde aplicar los mismos durante la ejecución. La solicitud se define para cada thread del árbol de proceso y puede estar asociada a determinados segmentos de código (joinpoint inicial y final) de un thread. Los parámetros varían de acuerdo a las necesidades propias de cada thread. Para cada thread se define necesidades de reservación de recursos y tipo de acceso.
2. La petición de QoS es gestionada por el módulo del sistema *Ensure QoS* que contiene los aspectos para brindar las garantías de QoS.

Flujos alternativos*:*

A1. En el paso 2, la gestión de QoS requiere establecer las condiciones para brindar las garantías de QoS requeridas:

1. Se invoca al módulo que contiene al aspecto <*Ensure QoS*>.
2. Se procede con la ejecución del advice apropiado del aspecto.
3. Se establecen garantías para cada thread determinado donde es necesario brindar garantías de QoS.
4. Durante la ejecución se sincronizan acciones con otros threads, para evitar conflictos sobre recursos y garantías diferentes de QoS.

A2. En el paso 2, la gestión de QoS debe retirar las garantías de *QoS* que brindo*:*

1. Se invoca al módulo que contiene al aspecto <*QoS Release*>.
2. Se procede con la ejecución del advice apropiado del aspecto.
3. Se retiran garantías establecidas para el thread
4. Se sincroniza la QoS con otros threads del árbol de proceso para mejorar las garantías de los mismos.

A3. En el paso 2, la gestión de QoS notifica las acciones realizadas:

1. Se invoca al módulo que contiene al aspecto <*Notify QoS*>.
2. Se procede con la ejecución del advice apropiado del aspecto.

A4. En el paso 2, la gestión de QoS debe sincronizar garantías de QoS del árbol de proceso:

1. Se invoca al módulo que contiene al aspecto <Synchronize QoS>.
2. Se procede con la ejecución del advice apropiado del aspecto.

3. Se sincronizan acciones para garantizar la QoS entre los threads.

La Tabla 7 detalla los pasos de los Aspectos del caso de uso Threads, para asegurar, liberar, notificar y sincronizar QoS para un escenario con un proceso con subprocesos hijos. La mayor parte de las operaciones de los aspectos Ensure QoS, QoS release y Notify QoS no sufren

mayores modificaciones respecto al escenario 1, salvo las derivadas de implementar la sincronización de la QoS para un grupo de threads y la gestión de QoS en base a los parámetros de la solicitud.

Ensure QoS (Asegurar QoS)	
Pasos	
1	El usuario o desarrollador solicita garantías de QoS para la ejecución de un determinado proceso y sus correspondientes threads. Cada solicitud se define en términos de parámetros (prioridad, tiempo de ejecución, política de acceso a recursos compartidos, acceso a computo de alto rendimiento, etc.) relacionados a la QoS y los correspondientes segmentos de código de cada thread donde la solicitud debe aplicarse, definidos por un joinpoint inicial y uno final. Así la definición de donde aplicar el código contaría con <id_thread, jointpoint inicial, jointpoint final, tipo de activación [after,before,around]>. Dinámicamente las condiciones de ejecución son modificadas alterando las garantías brindadas.
2	Se analiza las necesidades de cada solicitud, se determina priorización, planificación y la política de asignación de recursos actuales y los cambios necesarios para brindar las garantías requeridas a los threads. Parte de estas tareas se realiza con las llamadas al sistema de las Tablas 2 y 4.
3	Se realizan los cambios necesarios para establecer garantías de QoS, se replanifica y prioriza al árbol de proceso y el acceso a recursos para proveer las garantías QoS. Sino es posible proveer las garantías de QoS se rechaza la solicitud.
4 Pasos	Se sincronizan los cambios con otros threads del árbol de proceso para garantizar la QoS de cada uno de ellos.
1	El thread solicita el fin de garantías de QoS por alcanzar un jointpoint que señala el final de la aplicación de las garantías para el thread.
2	Se analiza la solicitud, se determinan los pasos para recuperar recursos comprometidos por las garantías de QoS.
3	Se realizan tareas para retirar del thread las garantías de QoS. Se replanifica al thread y el acceso a recursos a un estado previo y/o en su defecto se recuperan los recursos.
4	Se sincronizan tareas tendientes a brindar mayores garantías de QoS si fuesen necesarias para los otros threads.
Notify QoS (Notificar QoS)	
Pasos	
1	El cliente solicita información del programa respecto a las garantías de QoS al alcanzar un jointpoint inicial o final y al modificar planificación o priorización
2 Pasos	Se vuelca la información a un archivo auxiliar y/o visualiza por pantalla.
1	Se realizaron tareas para brindar o retirar garantías de QoS en un thread
2	Se analizá los cambios realizados. Se determina la nueva disponibilidad o disminución

	de recursos. Se emplean métricas para medir el rendimiento y eficacia de las políticas de QoS (tabla 1).
3	Se realizan medidas de ajuste para sincronizar las garantías QoS de todos los threads.

Tabla 7: Descripción pasos de los Aspectos de caso de uso Threads, Asegurar QoS, Liberar QoS, Sincronizar QoS y Notificar QoS

4.3 Escenario 3: Grupos de Threads

Se plantea un modelo de ejecución que consta de un grupo de procesos cooperantes los que generan procesos hijos dando lugar a una serie de árboles de proceso que residen en un host. Cada uno de los procesos hijos podría requerir condiciones de ejecución diferente. Entonces se requiere definir las garantías de QoS para los mismos.

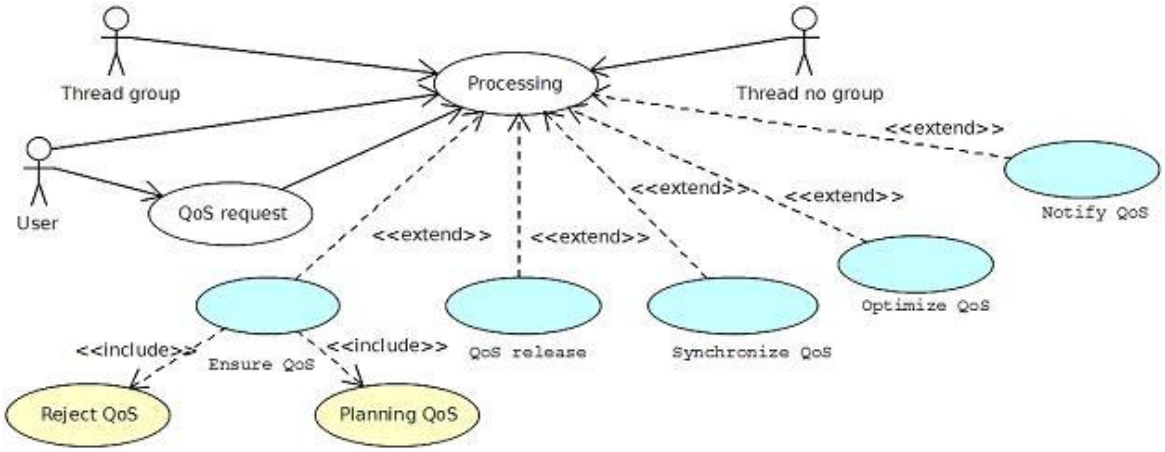

Figura 7: Caso de uso QoS para grupos de procesos con threads en AOSD/UC

Dado que las garantías de QoS son afectadas por la interacción de los diferentes threads y la carga de trabajo del sistema, se origina la necesidad de controlar las condiciones de cada árbol de procesos cooperantes del sistema que utilizan garantías de QoS. Así dentro de un grupo de procesos cooperante se originan esfuerzos tendientes a optimizar la utilización de recursos para los miembros del grupo. Pero no necesariamente para el resto de procesos del sistema. La Figura 7 representa este caso de uso, dónde el módulo que contiene el aspecto <Optimize QoS> trabaja a nivel de arboles de procesos mientras que el módulo que contiene el aspecto <Synchronize QoS> trabaja a nivel de threads en un mismo árbol.

Especificación de Caso de uso: Procesamiento <Processing>

Este caso de uso describe como un *Programa de interés* que consta de varios procesos cooperantes es ejecutado, durante su ejecución generan grupos de múltiples threads asociados a varios árboles de proceso.

Flujo básico:

1. El *Usuario* inicia la ejecución de un *Programa de interés*
2. El *Programa de interés* inicia su ejecución normal, generando múltiples threads.
3. Los *threads de cada proceso* avanzan en su ejecución
4. Los *threads de cada proceso* terminan su ejecución
5. El caso de uso termina

Flujo alternativo:
En el paso 2 o 3 el *Usuario o desarrollador* realizó una solicitud de *QoS*

Especificación de Caso de uso: Petición QoS <QoS request>
Este caso de uso describe como se realiza una petición de QoS mediante la interfaz *QoS request*.
Flujo básico:
1. El Usuario o desarrollador realiza una petición QoS mediante la interface *QoS request*, la que incluye los parámetros para definir la QoS requerida y como aplicarla durante la ejecución. Las garantías de QoS de la aplicación difieren para cada thread perteneciente a los árboles de proceso y los segmentos de código alcanzados por las mismas. Por lo tanto, los parámetros varían de acuerdo a las necesidades propias de cada thread.
2. La petición QoS es gestionada por el módulo del sistema que contiene los aspectos necesarios para brindar las garantías de QoS.
Flujos alternativos:
A1. En el paso 2, la gestión de QoS requiere ejecutar los pasos para asegurar las garantías de QoS:
1. Se invoca módulo que contiene al aspecto <*Ensure QoS*>.
2. Se procede con la ejecución del advice apropiado del aspecto.
3. Se establecen garantías de QoS para el thread individual.
4. Se sincronizan los cambios y sus efectos con otros threads del grupo.
5. Se optimiza la utilización de recursos para los miembros de los grupos.

A2. En el paso 2, la gestión de QoS requiere efectuar cambios en las garantías proporcionadas para liberar las condiciones de QoS:
1. Se invoca módulo que contiene al aspecto <*QoS Release*>.
2. Se procede con la ejecución del advice apropiado del aspecto.
3. Se retiran las garantías al thread.
4. Se sincroniza cambios y sus efectos en las garantías de otros threads del grupo.
5. Se optimizan recursos para los miembros de los grupos de threads.

A3. En el paso 2, la gestión de QoS notifica las acciones realizadas:
1. Se invoca módulo que contiene al aspecto <*Notify QoS*>.
2. Se procede con la ejecución del advice apropiado del aspecto.

A4. La gestión de QoS necesita sincronizar garantías de QoS:
1. Se invoca módulo que contiene al aspecto <Synchronize QoS>.
2. Se procede con la ejecución del advice apropiado del aspecto.
3. Se sincronizan acciones para garantizar la QoS en los threads del mismo árbol de proceso o grupo.

A5. La gestión de QoS debe optimizar la utilización de recursos:
1. Se invoca al módulo que contiene al aspecto <Optimize QoS>
2. Se procede con la ejecución del advice apropiado del aspecto.
3. Se optimiza los recursos para QoS con los grupos de threads.

La Tabla 8 detalla los pasos de los Aspectos para asegurar, liberar, sincronizar, optimizar y notificar garantías de QoS para un escenario que contiene un grupo de procesos los que poseen cada uno procesos hijos los que residen en un host.

Ensure QoS (Asegurar QoS)	
Pasos	
1	El usuario o desarrollador genera una solicitud de garantías de QoS para la ejecución de un conjunto de procesos y sus correspondientes threads. Se definen los parámetros para cada thread (prioridad, tiempo de ejecución, política de acceso a recursos compartidos, acceso a computo de alto rendimiento, etc.) y los segmentos de código donde serían aplicadas las garantías de QoS. La definición de dónde aplicar el código contaría con <id_proceso,id_thread, jointpoint inicial, jointpoint final, tipo de activación[after, before, around]>. Las condiciones de ejecución sufren cambios generando ajustes en las garantías de QoS.
2	Se analizá la solicitud para determinar los recursos requeridos, se determina condiciones actuales y las requeridas de priorización, de planificación y de asignación de recursos para brindar las garantías solicitadas al thread .
3	Se realizan cambios tendientes a brindar garantías de QoS, se replanifica los threads y acceso a recursos para proveer QoS. Sino es posible proveer las garantías de QoS se rechaza la solicitud.
4	Se sincronizan acciones con otros threads del grupo para garantizar la QoS.
5 Pasos	Se optimiza las garantías de QoS y manejo de recursos compartidos con otros grupos de threads.
1	El thread solicita fin de garantías de QoS por alcanzar un jointpoint que señala el final de la aplicación de las garantías.
2	Se analizá solicitud, se determinan acciones para recuperan recursos comprometidos por las garantías de QoS.
3	Se realizan acciones retirar del thread garantías de QoS. Se replanifica al thread y el acceso de los recursos a estado previo o en su defecto se recuperan los recursos.
4	Se sincronizan acciones con otros threads del grupo tendientes a brindar mejores y/o mayores garantías si fuesen necesarias.
5 Pasos	Las garantías de QoS son optimizadas y así como la asignación de recursos compartidos con otros grupos de threads.
1	Se realizaron cambios para brindar o retirar QoS en un thread
2	Se analizá los cambios y se determina la nueva disponibilidad o disminución de recursos. Se emplean métricas para medir el rendimiento y eficacia de las políticas de QoS (Tabla 1).
3 Pasos	Se sincronizan acciones con otros threads del proceso
1	Se realizaron tareas para brindar o retirar garantías de QoS
2	Se analizá tareas realizadas y se determina la nueva disponibilidad de recursos. Se emplean métricas para medir el rendimiento y eficacia de las políticas de QoS (Tabla 1)

3	Se optimizan las garantías mediante ajustes a las mismas en otros grupos de threads.
Notify QoS (Notificar QoS)	
Pasos	
1	El cliente solicita información de la aplicación respecto a las garantías de QoS. Al alcanzar un jointpoint inicial o final y al modificar planificación o priorización
2	Se vuelca la información a un archivo auxiliar y/o visualiza por pantalla.

Tabla 8: Descripción pasos Aspecto de caso de uso Grupos de Threads, Asegurar QoS, Liberar QoS, Sincronizar QoS, Optimizar QoS y Notificar QoS

5. CONCLUSIONES

5.1 Trabajos Relacionados

Otros autores que han estudiado la QoS y AOP:
a) Engvig (2005) describe como se han realizado aplicaciones experimentales de QoS sobre casos de estudio en aplicaciones como Cálculo de PI o transmisiones distribuidas de reproductor de audio. Se hace foco en el desarrollo de componentes mediante arquitectura QuA para aplicaciones sensibles a la QoS. Los experimentos muestran que algunos *Concerns* pueden ser completamente separados como la gestión de recursos y adaptación. A veces la implementación de algunos componentes debe ser modificada para procurar adaptabilidad. En el mencionado trabajo se establece que procurar eficiencia en la QoS brindada por los aspectos a las necesidades de la aplicación recae en el desarrollador.

b) Tesanovic et al. (2005) introduce un paquete de aspectos para permitir retroalimentación de QoS (Re-QoS) proporcionando una serie de políticas de QoS en sistemas de tiempo real. En el mismo trabajo se utilizan los aspectos dentro del paquete QoS para adaptar de manera eficiente la gestión de QoS en un sistema en tiempo real, basándose en los requisitos de la aplicación. Una ventaja de aplicar AOP es que las políticas de QoS se encapsulan en los aspectos y se pueden cambiar y modificar de forma independiente del sistema en tiempo real. Se desarrolla el concepto de aplicar la QoS a través de un paquete para sistemas de tiempo real.

c) Huang (2013) describe la necesidad de QoS a raíz de la introducción de servicios web equivalentes en las plataformas de computación en Cloud. La QoS es usada en el ámbito de servicios web para seleccionar en base a la calidad a los servicios los más adecuados.

5.2 Conclusiones Finales y Trabajo Futuro

Las principales características de este trabajo inicial son definir un enfoque general basado en el procesamiento de la aplicación tradicional y no específicamente en su tipo y/o naturaleza. Esto permitiría reutilizar los mecanismos para proporcionar QoS a un conjunto más amplio de aplicaciones de forma más integral y flexible. Se plantea la utilización de un enfoque aspectual para proporcionar las garantías de QoS a las aplicaciones. El enfoque aspectual evitaría situaciones en las cuales la introducción de las funciones necesarias para brindar las garantías de QoS en la aplicación genera un fuerte entrelazado con otras funciones asociadas a los requerimientos funcionales generando una fuerte pérdida de modularidad. Un enfoque aspectual permite mantener encapsuladas las funciones para la gestión de QoS como el

manejo de las condiciones de procesamiento, manejo recursos como la memoria, control de banda de la red, seguimiento y estimación de recursos, etc.

La invasión del código de la plataforma de ejecución o del middleware para incorporar funciones para garantizar QoS no es deseable debido a la complejidad de tales modificaciones y la dificultad de lograr una integración libre de errores y efectos inesperados sobre el sistema. Siendo la implementación de aspectos para realizar los insertos necesarios en el código de aplicación más eficaz, flexible, mantenible y reutilizable. A raíz de que la encapsulación provista por el enfoque aspectual simplifica el manejo de las modificaciones necesarias al código para dar soporte a requerimientos de QoS no nativos en el mismo. Evitando muchos de los problemas de integración y efectos inesperados por no poder delimitar correctamente los alcances de los cambios.

Este trabajo detalla una serie de casos de uso básicos basados en el procesamiento de la aplicación tradicional de Linux donde sería factible utilizar AOP para implementar garantías de QoS a través del enfoque aspectual. Los casos de uso propuestos se han desarrollado para ser lo más genéricos posibles. Esto último los hace altamente adaptables y flexibles a desarrollos de naturaleza diversa. No obstante el éxito en la adaptación está sujeta por un lado, a las capacidades propias de las funciones de gestión de QoS para satisfacer las expectativas y por otro lado a la definición correcta de PointCuts donde aplicar la QoS en el código de la aplicación. Esto último implica que normalmente para implementar garantías de QoS sobre una aplicación se requiere la intervención del desarrollador.

La implementación de los escenarios propuestos conformaría la base para implementar un framework con capacidad genérica para brindar garantías de QoS a aplicaciones residentes en un mismo host. Este framework utilizaría las capacidades contenidas en el enfoque aspectual para integrarse a distintos desarrollos.

A futuro, los casos de uso que han sido presentados se utilizarán para implementar soluciones mediante aspectos a los distintos escenarios de procesamiento planteados. La incorporación de garantías de QoS a las aplicaciones tradicionales Linux descripta en los escenarios de la Sección 4 plantea la necesidad de lograr desarrollar una serie de módulos que contengan a los aspectos que introducen las funciones necesarias para brindar las garantías QoS.

Parte de la tarea de desarrollar los módulos involucra el diseño y puesta a prueba de los mecanismos propios para brindar las garantías de QoS a través de los aspectos. Estos mecanismos seguramente deberán ser refinados para obtener un impacto positivo en los rendimientos de las aplicaciones, acorde a las garantías de QoS solicitadas. Si bien para algunos pasos en las soluciones planteadas para los casos de uso se toma como ejemplo las llamadas de sistemas operativos Linux, es factible su implementación en otras plataformas o en un escenario distribuido podría explorarse el mismo enfoque para obtener una mejor garantía de QoS para los procesos por parte de los sistemas operativos subyacentes.

REFERENCIAS

ABRAMSON D.(2011). *"Mixing Cloud and Grid Resources for Many Task Computing"*. Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers (MTAGS '11). ACM, New York, NY, USA, 1-2. DOI: http://dx.doi.org/10.1145/2132876.2132878

AHMED M., Suhaimi M., Faisal Q., Haseeb S.(2007).*"Evaluating QoS performance of streaming video on both IPv4 and IPv6 protocols"*. SpringSim '07 Proceedings of the 2007 spring simulaiton multiconference - Volume 1 Pages 109-116. ISBN:1-56555-312-8

BAKKER J., Ut B.T., Clarke S., Tcd A.J. (2005). *"Survey of Aspect-Oriented Analysis."* 1–259. AOSD-Europe-ULANC-9 Deliverable.

BECHLER M., Ritter H.(2001): *"QoS in the Linux Operating System"* Technical Report.Universität Karlsruhe. Institut für Telematik

BLANQUER J., Batchelli A. (2004). *"QoS for internet services: done right"*. Proc. 11th …. DOI: http://dx.doi.org/10.1145/1133572.1133596

BOVET D. y Cesati M.(2006): *"Understanding the Linux Kernel"*, ISBN-10 0-596-00565-2 , O´Really

BRUNO J., Gabber E., Ozden B., Silberschatz A., Ave. M., Hill M.(1998): *"The Eclipse Operating System : Providing Quality of Service via Reservation Domains"*. Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '98). USENIX Association, Berkeley, CA, USA, 20-20.

CANONICO R., D'Arienzo M.(2002): "*On the introduction of quality of service awareness in legacy distributed applications*". Proc. 14th ….

DEMERS A., Keshav S., Shenker S.(1989). *"Design and Analysis of a Fair Queuing Algorithm"*. In *P*roceedings of the ACM SIGCOMM Austin, Texas, September 1989.

ENGVIG T. (2005). *"Separating the QoS Concern in QuA using Aspect Oriented Programming"* , University of Oslo, Department of Informatics.

GALLEGE L.S., Gamage D.U., Hill J.H., Raje R.R. (2013). *"Trust Contract of a Service and its role in Service Selection for Distributed Software Systems".* (CSIIWRW'13) DOI: http://dx.doi.org/10.1145/2459976.2459994

GOYAL P., Guo X., Vin H. M.(1996). *"A Hierarchical CPU Scheduler for Multimedia Operating Systems"*. In Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation Seattle, Washington, October 1996.

HILTUNEN M., Taïani F., Schlichting R. (2006). *"Reflections on Aspects and Configurable Protocols"*. 87–98. DOI: http://dx.doi.org/10.1145/1119655.1119669

HOFFERT J., Schmidt D., Gokhale A. (2010).*"Adapting and Evaluating Distributed Real-time and Embedded Systems in Dynamic Environments"* METRICS AND MACHINE LEARNING. 23–28. DOI: http://dx.doi.org/10.1145/1862821.1862825

HUANG X. (2013).*"Usage QoS: Estimating the QoS of Web Services through Online User Communities"* Charles Sturt University (December 2013) Journal ACM Transactions on the Web (TWEB) Volume 8 Issue 1, Article N°1, ACM New York, NY, USA. http://dx.doi.org/10.1145/2532635

HYDEN E.A. (1994): *"Operating System Support for Quality of Service"*. University of Cambridge.

IEFT RFC 1633, http://tools.ietf.org/html/rfc1633

IEFT RFC 2205, https://tools.ietf.org/html/rfc2205

IEFT RFC 5974, https://tools.ietf.org/html/rfc5974

IETF RFC 4445, https://tools.ietf.org/html/rfc4445

ISO/IEC 14882:2014(E). Lenguaje C++ estándares. https://isocpp.org/std/the-standard

ISO/IEC 9899. Lenguaje C estándares. http://www.open-std.org/jtc1/sc22/wg14/www/standards.html

IYER R., Zhao L., Guo F., Illikkal R., Makineni S., Newell D., Solihin Y., Hsu L., Reinhardt S. (2007). *"QoS policies and architecture for cache/memory in CMP platforms"*. ACM SIGMETRICS Perform. Eval. Rev. 35, 25.

JACKSON A., Nedos A., Hatton N., Amor M. y Fuentes L. (2009).*"A domain analysis of key concerns – known and new candidates."*

JACOBSON I. , Pan-Wei Ng. (2004). "Aspect-Oriented Software Development with Use Cases", Addison-Wesley Object Technology Series. Addison-Wesley Professional.

JONES T. (2007). Anatomy of the Linux kernel. http://www.ibm.com/developerworks/library/l-linux-kernel/

KICZALES G. (1996). *"Aspect-oriented programming"*. *ACM Comput. Surv.* 28, 4es, Article 154 (December 1996). DOI= http://doi.acm.org/10.1145/242224.242420

LINUX MAN pages , https://www.kernel.org/doc/man-pages/

NAHRSTEDT K., Chun H., Narayan S.(1998). *"QoS-aware Resource Management for Distributed Multimedia Applications"*, Journal on High-Speed Networking, Special Issue on Multimedia Networking, IOS Press, Vol. 8, No. 3-4, pp. 227-255

POELLABAUER C., Schwan K. (2005).*"Flexible cross-domain event delivery for quality-managed multimedia applications"*. ACM Trans. Multimed. Comput. Commun. Appl. 1, 248–268.

SCHAUERHUBER A., Schwinger W.,Kapsammer E., Retschitzegger W. y Wimmer M.(2006). *"A Survey on Aspect-Oriented Modeling Approaches"*.

SHANKARAN N., Roy N., Schmidt D., Koutsoukos X., Chen Y., Lu C. (2008).*"Design and Performance Evaluation of an Adaptive Resource Management Framework for Distributed Real-Time and Embedded Systems"*. EURASIP J. Embed. Syst. 2008, 250895.

SPINCZYK O. (2005). *"AspectC++ – Execution Model Overview"*.

SPINCZYK O. y Urban M. (2012). *"Documentation: AspectC++ Language Reference"*. 1–42.

TESANOVIC A., Amirijoo M., Bjork M. y Hansson J. (2005). *"Empowering configurable QoS management in real-time systems"*. AOSD 2005 4th Int. Conf. Asp. Softw. Dev. - Conf. Proc. 39–50.

WOLF L. C., Steinmetz R. (1997). *"Concepts for Resource Reservation in Advance"*, Special Issue of the Journal of Multimedia Tools and Applications on "The State of The Art in Multimedia", Vol. 4. No. 3, May 1997

XIAO X. (2008): Technical, "*Commercial and Regulatory Challenges of QoS: An Internet Service Model Perspective*". ISBN: 9780123736932. Morgan Kaufmann Publishers Inc.