Uso de Frameworks y Patrones de Diseño en Osmius: Software distribuido para la recolección de eventos en Red

La reusabilidad de código, diseños e ideas se ha convertido en el objetivo fundamental de las técnicas y paradigmas de la ingeniería del software por los evidentes beneficios que reporta a la hora de desarrollar un producto software. En este artículo se expone Osmius como producto capaz de recolectar eventos desde diferentes elementos conectados a una red, y cómo implementa la reusabilidad a través de los *frameworks* que proporciona el *middleware ACE,* y a través del uso de Patrones de Diseño. El uso de los Patrones se presenta en su entorno específico dentro la arquitectura software del producto y como solución a unos objetivos concretos en el contexto que proporciona Osmius.

Contenido

Introducción....3
Software de Supervisión....5
Metodologías para incrementar la Reusabilidad....7
  Middleware....7
  Frameworks....9
  Patrones de Diseño....11
ADAPTIVE Communication Environment - ACE....13
  Descripción....13
  Arquitectura....15
  Frameworks....17
    Reactor y Proactor....17
    Configuración de Servicio....19
    Control de Tareas y Procesos....20
    Acceptor y Connector....21
    Streams....22
Osmius....23
  Introducción....23
  Arquitectura....25
    Centro de Supervisión....26
    Agente Maestro....27
    Agente de Supervisión....30
  Resúmen de Características....33
Patrones Utilizados....35
  Wrapper Facade....35
    Descripción....35
    Contexto....35
    Ejemplo de uso....36
  Reactor....37
    Descripción....37
    Contexto de uso....37
    Ejemplo de uso....37
  Acceptor-Connector....41
    Descripción....41
    Contexto de uso....41
    Ejemplo de uso....42
Trabajo Futuro e Investigación....44
Conclusiones....46
Referencias....47
  The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming....48

Introducción

El desarrollo de software preparado para utilizar las capacidades que ofrecen las redes de comunicación cobra cada vez más importancia en un mundo en el que tanto las personas, como las entidades y los sistemas se encuentran en redes con mayor ancho de banda y con más y mejores servicios que ofrecer y utilizar.

Dos características fundamentales de dicho software han de ser la **capacidad de adaptación** y la **eficiencia** en el tratamiento de sus tareas. Necesitamos que el software sea capaz de adaptarse a nuevas formas de comunicación y de tipos de datos a transmitir, adaptarse a nuevos protocolos y sistemas de subscripción, así como a requisitos de calidad y de disponibilidad. La eficiencia es necesaria para poder responder con el adecuado rendimiento a transacciones que así lo demanden como, por ejemplo, en el control de procesos en aviónica o el tratamiento de la transmisión de videos en tiempo real.

Si por los menos pedimos estas dos cualidades al software distribuido en red, el propio proceso de desarrollo de dicho software debe - idealmente – caracterizarse por:

- Uso intensivo de técnicas de reutilización: Nos permite mayor capacidad de adaptación.
- Modularidad: Permite añadir, quitar o modificar partes diferenciadas sin afectar a otras.

El software distribuido en red o software de comunicaciones es complejo por diferentes tipos de razones. Inheremententemente dicha complejidad viene de problemas como la detección y posterior recuperación de fallos de comunicaciones, de servidores o de procesos, tratamiento de diferentes latencias, el diseño distribuido óptimo de procesos para el procesamiento de un modelo concreto, o el diseño de los servicios necesarios dentro de cada proceso. La complejidad derivada del entorno está asociada, por un lado, con problemas derivados de la incompatibilidad entre plataformas y APIs (“Application Program Interface”) y protocolos y, por el otro, con que los middleware de más alto nivel capaces de encapsular los problemas citados no están lo suficientemente optimizados para ser utilizados por aplicaciones que demandan rendimiento.

Para facilitar el paso de cubrir los requisitos demandados en un entorno con una gran complejidad, es fundamental hacer uso de técnicas de reusabilidad de código, diseños e ideas, como son la programación orientada a objeto (OO) y el uso de patrones de diseño y frameworks.

A lo largo de este artículo describiremos primero en qué consiste un sistema de monitorización y supervisión y cuál es el estado actual en este campo respecto a logros, retos y necesidades a corto y medio plazo. A continuación podremos las bases respecto a los conceptos de middleware, framework y patrones de diseño, que necesitaremos para comprender su significado y utilidad dentro del objeto del artículo.

Introduciremos el middleware orientado a software distribuido en red preparado para aplicaciones en tiempo real llamado ACE (ADAPTIVE Communication Environment), y explicaremos su estructura, funcionalidad y sus principales frameworks y componentes. Como ejemplo de software de supervisión concreto utilizaremos el producto Osmius, del que explicaremos los conceptos en los que se basa, su arquitectura y características diferenciadoras.

Como punto central del artículo se presenta cómo, por qué y en qué punto se han utilizado Patrones de Diseño y Frameworks para implementar las funcionalidades y requisitos de calidad de servicio (QoS) de Osmius.

Finalmente se exponen el futuro y las líneas de investigación que se abren en el campo de la arquitectura de software dentro de la supervisión de sistemas, y un planteamiento de expectativas sobre el desarrollo presentado.

Software de Supervisión

Los **objetivos principales** de un sistema de supervisión son seguir el estado de una serie de sistemas e informar de su evolución y sobre los problemas detectados (supervisión reactiva), incluso adelantándose a los mismos (supervisión proactiva).

En el dominio de las aplicaciones de monitorización están aquellas orientadas a la supervisión de sistemas de comunicaciones, servicios y sistemas informáticos, y las más orientados al mundo industrial como pueden ser los sistemas SCADA (Supervisory Control and Data Acquisition – Supervisión de Control y Adquisición de Datos).

Esta aplicaciones se basan normalmente en el paradigma de comunicación basada en eventos o, como también es conocido, publicación/subscripción de notificación por eventos. La aceptación de este paradigma está ampliamente extendida como podemos comprobar en su incorporación a estándares como CORBA (Common Object Request Broker Architecture), SOA (Service Oriented Architecture) y JMS (API de servicios de mensajería de Java) y a sistemas comerciales como TIBCO.

Inicialmente la supervisión y monitorización estuvo orientada a los dispositivos concretos que formaban nuestra red. Este tipo de sistemas estaba orientado a usuarios muy técnicos con una formación muy específica para comprender la información presentada y actuar en consecuencia. En los últimos años se observa un cambio de orientación hacia los Servicios que suministra el sistema monitorizado y que dependen de los dispositivos y de las relaciones entre ellos. La revolución que ha supuesto Internet y los altos niveles de conectividad necesarios para la comunicación constante entre usuarios y empresas, han endurecido las exigencias de rendimiento y disponibilidad de muchos proveedores que se plasman en contratos de Acuerdos de Nivel de Servicio (ANS) con penalizaciones en caso de no cumplimiento. En este entorno es fundamental contar con sistemas de monitorización preparados para el mantenimiento reactivo y proactivo de los sistemas y dispositivos y servicios.

Estos sistemas se basan en gran medida en su interacción con un humano experto, que es quien atiende los eventos y alarmas finales en una consola centralizada global u organizada por dominios. Por dominios entendemos agrupaciones lógicas de eventos, o de los dispositivos que los originan, basadas en criterios definidos por los diferentes perfiles de usuarios, desde más técnicos a más orientados a negocio. En ocasiones se generan gran cantidad de eventos que el administrador debe filtrar para así poder identificar la causa origen del problema cuanto antes, y ejecutar las acciones necesarias para restablecer el servicio o resolver la degradación del mismo.

Desde el punto de vista de la arquitectura software en estos sistemas, podemos hablar de una amplia aplicación del **Paradigma Gestor-Agente** en el que se distribuyen agentes en la infraestructura a monitorizar, y que se encargan de recolectar eventos y enviarlos a un gestor central para su presentación y/o procesamiento.

Es verdad que dicha arquitectura puede presentar problemas de rendimiento en entornos con una gran cantidad de elementos y pueden presentarse cuellos de botella en el servidor central, como también es cierto que un buen diseño de la arquitectura de despliegue y del número y calidad de información de los eventos a recolectar puede paliar dicha problemática.

Actualmente la supervisión de sistemas está cobrando un gran protagonismo ya que los servicios que se prestan ahora desde los Centros de Procesamiento de Datos (CPD), tienden a ser más exigentes desde el punto de vista de la disponibilidad, provocado sobretodo por los nuevos servicios a los usuarios de Internet como pueda ser el Comercio-Electrónico o los servicios multimedia que posibilitan los mayores anchos de banda.

La combinación de elementos en red que es necesario supervisar, más los contratos firmados con los usuarios de los servicios proporcionados por dichos elementos en forma de ANS, junto con la creciente oferta de servicios hace que tome gran importancia contar con sistemas de monitorización robustos, adaptables a nuevos entornos y muy eficaces en el tratamiento de los eventos.

Osmius trata de cubrir esa necesidad haciendo uso de los últimos paradigmas en ingeniería del software y una utilización intensiva de la modularidad y los mecanismos de reutilización.

Instancia:

Origen de un evento. Unidad fundamental sobre la que está construida un sistema que deseamos supervisar.

Servicio:

Conjunto de funcionalidades ofrecidas a un cliente o usuario mediante una funcionalidad pactada. Puede verse como una agrupación de instancias y aplicaciones.

Proceso de Negocio:

Wikipedia: Conjunto de tareas relacionadas lógicamente llevadas a cabo para lograr un resultado de negocio definido. En el contexto de la monitorización supone una agrupación de servicios y disponibiliddades necesarias para dicho proceso .

Acuerdo de Nivel de Servicio (ANS):

Contrato con el cliente o usuario respecto al rendimiento y/o disponibilidad de una aplicación, servicio o proceso de negocio.

Metodologías para incrementar la Reusabilidad

La reusabilidad de código y diseños es fundamental en el proceso de desarrollo de software, porque al reutilizar escribimos menos código que, además de mejorar la productividad, reduce la cantidad de fallos cometidos y los periodos de pruebas y consigue que la calidad y solidez general del producto se vea incrementada.

Middlewares, frameworks y patrones son conceptos diferentes pero interrelacionados que pretenden incrementar la reusabilidad.

Middleware

Es un software que puede incrementar significativamente la reusabilidad mediante soluciones utilizables rápidamente y basadas en estándares aplicables a problemas y tareas comunes en programación.

La idea es que los desarrolladores se puedan concentrar en asuntos propios de la aplicación y olvidarse de problemas comunes - estructurales o no - ya resueltos previamente de forma elegante y satisfactoria.

CORBA (Common Object Request Broker Archytecture) es un middleware estándar que surge la la organización internacional OMG. Otros estándares como JVM, J2EE y .NET han surgido de consorcios industriales y líderes en diferentes áreas del mercado.

Para el desarrollo de los diferentes middlewares son cruciales los patrones de diseño y los frameworks. Es común que los middleware se desarrollen usando frameworks que se basan a su vez en patrones de estrategias de composición y optimización. Entre los tres tipos de abstracción se aprovechan muchas sinergías. El tipo de middleware en el que estamos interesados es el orientado a aplicaciones distribuidas en red.

Normalmente podemos descomponer un middleware en sus capas de igual forma a como se hace con los protocolos de red.

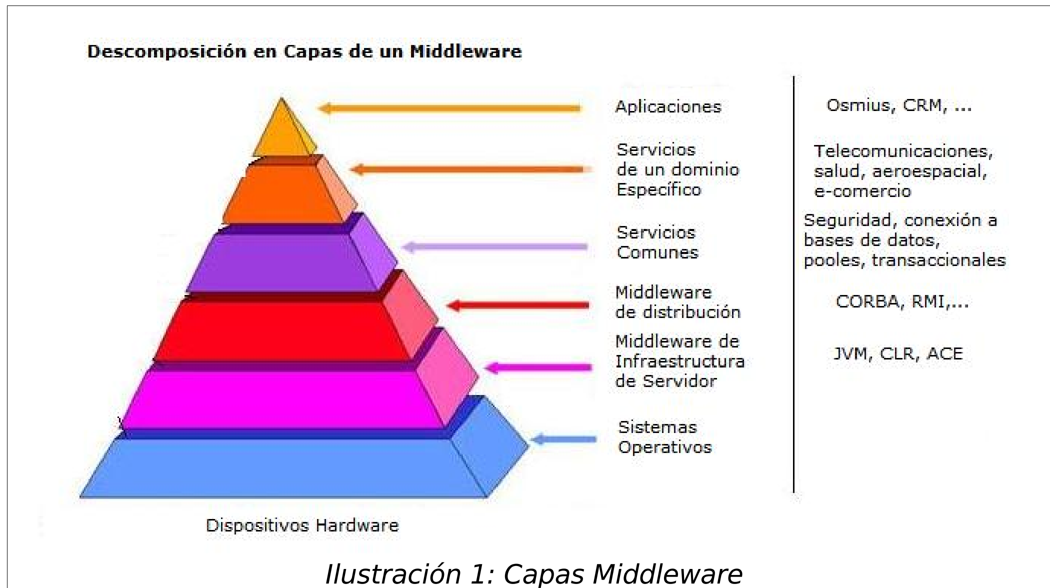


Ilustración 1: Capas Middleware

Empezando de abajo arriba y obviando las capas de dispositivos hardware y la de los sistemas operativos:

- **Middleware de Infraestructura de Servidor**:

  Sirve para abstraerse y mejorar los mecanismos nativos proporcionados por los sistemas operativos subyacentes. Proporciona mecanismos resusables para manejo de la demultiplexación de de eventos, comunicación entre procesos, concurrencia y sincronización por nombrar algunos ejemplos. Abstrayendo estos mecanismos y las peculiaridades de los diferentes SO creamos objetos reusables que ayudan a eliminar los aspectos tediosos y tendentes a errores, además de poco portables, de bregar con los detalles de bajo nivel de cada sistema operativo (SO). Ejemplos de este tipo de middleware son la máquina virtual de Java (JVM), y el Common Language Runtime (CLR) de Microsoft. **ACE** encaja dentro de esta categoría.

- **Middleware de Distribución**:

  Proporcionan un modelo de programación distribuida de más alto nivel cuyos objetos e interfaces automatizan y extienden la funcionalidad de los SOs encapsulada por la capa anterior de infraestructura de servidor. Nos permite programar aplicaciones llamando a operaciones en los objetos destino olvidándonos de dependencias como la localización, el lenguaje de programación, SO, plataforma, protocolos de comunicación y hardware. En esta categoría se encuentran CORBA y Java Remote Invocation (JRI) de Sun. SOAP (Simple Object Access Protocol) es un marco extensible y descentralizado que permite trabajar sobre múltiples pilas de protocolos de redes informáticas. Los procedimientos de llamadas remotas pueden ser modelados en la forma de varios mensajes SOAP interactuando entre sí, y permite el intercambio de información estructurada (XML) en la Web utilizando diferentes protocolos tales como HTTP, SMTP y MIME.

- **Middleware de Servicios Comunes**:

Mejoran el middleware de distribución definiendo servicios reusables de más alto nivel e independientes de dominio, que permite a los desarrolladores de aplicaciones centrarse en la programación y el diseño de la lógica del negocio, evitándose la necesidad de escribir el código que tendrían que escribir para aplicaciones distribuidas si usaran middleware de capas más bajas. Ejemplos de servicios ofrecidos son toda la capa transaccional, seguridad o *pooles* de conexión a base de datos, de manera que el programador no tenga que realizar más esas tareas utilizando un modelo de componentes y lenguajes sencillos por lotes.

- **Middleware de Dominio Específico**:

  Los servicios en este caso están asociados y enfocados a determinados dominios como pueden ser las telecomunicaciones, el comercio electrónico, o el campo de la salud y la medicina.

Frameworks

Las características deseables para un producto software son variadas y muy necesarias. Un software debe ser extensible, modular, robusto y eficiente. Conseguir estas características es complejo, y los frameworks aportan la tecnología que trata de abordar esta complejidad incrementando los niveles de reusabilidad del código.

Un framework es una colección de componentes software que colaboran entre sí para proporcionar una arquitectura reusable para una determinada familia de aplicaciones.

Los componentes de un framework son clases (gestores de mensajes, manejadores de eventos, mapas de conexiones,...), jerarquías de clases, categorías de clases (familias de mecanismos de control) y objetos.

Un componente es una unidad de encapsulación de servicios con uno o más interfaces a través de los cuales se proporciona acceso a los servicios que ofrece.

También podemos definir un framework como un diseño reusable, de una parte o de un todo de un sistema, representado por un conjunto de clases abstractas y la forma en que interaccionan sus instancias.

Se trata de integrar componentes que son independientes de la aplicación con los que son específicos de forma sencilla para que puedan colaborar entre ellos. La configuración de los componentes de un framework puede ser dinámica (en instalación o ejecución) o estática (en tiempo de compilación).

Normalmente los framework se apoyan y hacen un uso intensivo de los patrones de diseño.

Un framework tipo Modelo/Vista/Controlador se puede descomponer en tres patrones de diseño principales y varios secundarios.

- *Observer*: Para asegurar que la presentación de la vista del modelo está

actualizada.

- *Composite*: Para anidar vistas.
- *Strategy*: Para provocar que las vistas deleguen la responsabilidad de tratar los eventos de usuario a sus controladores.

Un framework orientado a aplicaciones distribuidas en red tiene tres características principales:

- Inversión de Control.

  Cambia el modo de programación. No somos nosotros como desarrolladores de un aplicación basada en un framework los que programamos la lógica de control del software. El framework proporciona mecanismos que llamarán a los métodos específicos que programemos para nuestra aplicación.

  La inversión se realiza a través de funciones de devolución de llamada (*callbacks*) a los métodos gancho (*hook methods*) cuando ocurre un evento como la llegada de datos a una conexión de red. Cuando llega el evento el framework llama al método gancho virtual de un componente registrado previamente que realiza el procesamiento específico de la aplicación.

  Los métodos gancho virtuales separan y abstraen el software específico del software reusable que proporciona el framework, y esto nos permite extender y personalizar las aplicaciones de forma más fácil y sencilla.

- Proporciona un conjunto integrado de estructuras y funcionalidad para determinado dominio.

  Soluciones comunes, recurrentes, probadas y elegantes.

- Es una aplicación semi-completa.

  El programador la personaliza para crear la aplicación buscada extendiendo los componentes reusables del framework.

El objetivo perseguido es aumentar la cantidad de código reutilizado frente al nuevo.

El uso de frameworks presenta algunos problemas:

- Difíciles de aprender.

  Son muy potentes pero también son complejos. Requieres de más y mejor documentación que otros sistemas.

- Difíciles de desarrollar:

  Requieren profundos conocimiento de ingeniería y programación, dominio de patrones de diseño y de técnicas Orientadas a Objeto. Mejores programadores que la media.

- Restringido por lenguaje:

  Están implementados en un lenguaje y en general quedan restringidos a productos desarrollados en el mismo lenguaje. CORBA trata de resolver este problema entre otros.

Algunos ejemplos de frameworks son MacApp, X-Windows, Java Swing o MFC.

Patrones de Diseño

Christopher Alexander, arquitecto de edificios reconocido como el originador de la idea de los patrones, explica que “cada patrón en una regla con tres integrantes que expresa la relación entre cierto contexto, un problema y una solución a dicho problema”.

Un desarrollador de midlewares, frameworks o aplicaciones específicas se tiene que enfrentar a retos relacionados con el diseño y la programación de asuntos tales como:

La persistencia de los datos, la organicación de los datos, la gestión de las conexiones, inicialización de servicio y objetos, el manejo de los parámetros de configuración, la comunicación entre estructuras, la concurrencia, el control de acceso a los recursos, control del flujo de programa, la gestión de errores y de información de *log*, bucles de gestión de eventos, etc.

La resolución de estos retos se presenta una y otra vez, e inicialmente el conocimiento necesario para resolverlos estaba en las mentes de algunos desarrolladores o, de forma implícita y poco aprehendible, en el código. Los patrones de diseño tratan de solventar este problema.

Los patrones permiten la reutilización de los conocimientos y experiencia de diseño implementando los aspectos estáticos y dinámicos de soluciones satisfactorias a problemas que surgen al desarrollar software en un contexto determinado. De esta forma ayudan a incrementar la reusabilidad capturando y reutilizando las estructuras y las formas de colaboración de los elementos clave de undiseño software.

Se ha realizado un gran esfuerzo para documentar patrones y en el desarrollo de frameworks basados en ellos que permitan a su vez el desarrollo y la reutilización de middlewares.

- Patrones de Diseño:

  Proporcionan un esquema con los elementos de un sistema software y las relaciones entre ellos y describen una estructura dfe elementos en comunicación que soluciona un problema de diseño general en un contexto particular.

- Patrones de Arquitectura:

  Expresan la estructura fundamental de organización de un software y proporcionan un conjunto de subsistemas con sus responsabilidades, y las líneas para organizar la relación entre dichos subsistemas.

- Lenguajes de Patrones:

  Agrupan una trama de patrones para definir un vocabulario para hablar de problemas de desarrollo y ofrecer un proceso para la resolución de aquéllos.

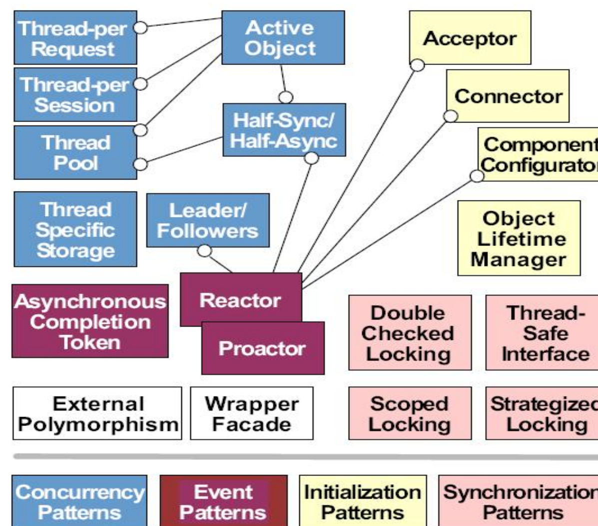


Ilustración 2: Lenguaje de Patrones para Programación Distribuida en Red en ACE

Podemos clasificar los patrones de diseño como:

- **Patrones de Creación:**
  Orientados a la inicialización y configuración de clases y eventos.
  - *Abstract Factory*
    Para construir objetos.
  - *Singleton*
    Factory para construir una sola instancia.

- **Patrones Estructurales:**
  Orientados a la separación entre el interfaz y la implementación de clases y objetos.
  - *Bridge*
    Abstracción para seleccionar una de entre varias implementaciones.
  - *Composite*
    Estructura para construir agregaciones recursivas.
  - *Wrapper Facade*
    Simplifica el interfaz para un subsistema.

- **Patrones de Comportamiento:**
  Orientados a las relaciones dinámicas entre conjuntos de clases y objetos.
  - *Iterator*
    Agregar elementos que son acceddos de forma secuencial.
  - *Observer*
    Actualizar dependencias automáticamente cuando cambia alguien.

Podíamos decir que los patrones son descripciones más abstractas que los frameworks ya que éstos están implementados en un lenguaje en particular. En general los frameworks albergan e implementan docenas de patrones de

diseño, y a su vez los patrones se utilizan a menudo para documentar los frameworks.

ADAPTIVE Communication Environment - ACE

Descripción

ACE se corresponde con las siglas de ADAPTIVE Communication Environment, y es un middleware que proporciona varios marcos de trabajo – frameworks – orientado a objeto, de código abierto y disponible gratuitamente, que implementa muchos de los patrones centrales para el desarrollo de software de comunicación.

ACE está orientado a desarrolladores de aplicaciones y servicios de alto rendimiento en red y tiempo real. Simplifica el desarrollo de este tipo de aplicaciones haciendo uso de la programación orientada a objetos, servicios y componentes.

Hoy en día desarrollar software debe tener como uno de sus principales objetivos ser independiente de plataformas tanto hardware como software y sistemas operativos, y no verse atado a ningún fabricante o proveedor en concreto. Las soluciones que hay en el mercado son:

- **Sun Java Virtual Machine (JVM)**, que proporciona una máquina virtual responsable de interpretar el código Java y traducirlo al lenguaje de la máquina real en la que reside. De esta manera las aplicaciones ven de forma transparente el sistema operativo subyacente.
- **Microsoft Common Lenguaje Runtime (CLR)**, que es la infraestructura sobre la que están construidos los servicios Web de Microsoft .NET. Es similar a JVM
- **ADAPTIVE Communication Environment (ACE)**, disponible de forma gratuita, con acceso a su código y altamente portable entre plataformas. ACE es un entorno y un conjunto de herramientas escritas en C++, que separa el sistemas operativo de la aplicación a través de la capa de Adaptación al Sistema Operativo.

Las principales diferencias entre ACE, JVM y CRL .NET son:

- ACE siempre genera un código compilado para la plataforma destino en lugar de un código intermedio que debe interpretarse después, eliminando un nivel de direccionamiento y optimizando el rendimiento en ejecución.
- ACE es Código Abierto, por lo que es posible adaptarlo o usar sólo un subconjunto en el que estemos interesados.
- ACE es capaz de ejecutarse en MÁS sistemas operativos y hardware que JVM y CLR.

ACE corre actualmente en **multitud de plataformas** como son todos los sistemas Windows de 32 y 64 bits, WinCE, Linux RedHat, Debian y Suse, Solaris, AIX, HP-UX, FreeBSD, NetBSD. Como ejemplos de sistemas en tiempo real están VxWorks, LynxOS y OS/9 y de grandes sistemas OpenVMS y Tandem.

Los beneficios resumidos de usar ACE son:

Aumento de la Portabilidad:

ACE está preparado para generar aplicaciones en la gran mayoría de Sistemas Operativos. No tenemos que preocuparnos por quedarnos atrapados con un solo sistema operativo.

Aumento en la Calidad del Software:

Los componentes de ACE están diseñados utilizando muchos patrones de diseño. Todo esto mejora cualidades como la flexibilidad, la modularidad, la reusabilidad y la solidez frente a errores del software.

Mejora de la Eficiencia:

ACE ha sido cuidadosamente diseñado para soportar un amplio espectro de requerimientos de aplicaciones en cuanto a su Calidad de Servicio (QoS). Esto incluye requisitos de bajas latencias para aplicaciones muy sensibles a retrasos en las comunicaciones en red, de alto rendimiento para aplicaciones que demandan un fuerte uso del ancho de banda, y las necesidades de predicción en aplicaciones de tiempo real.

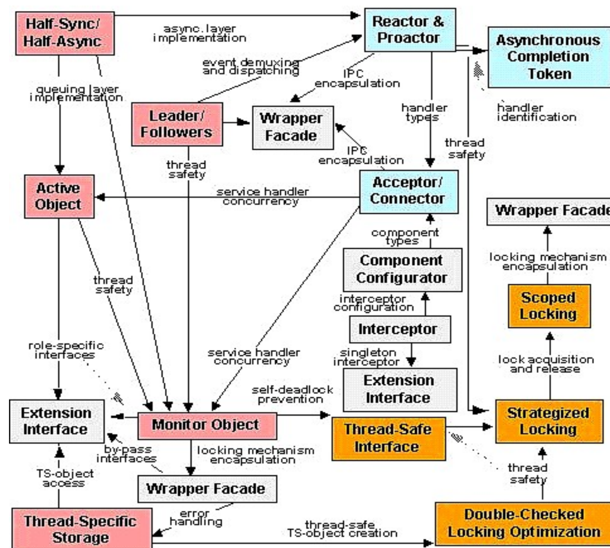


Ilustración 3: Patrones en ACE

ACE consta a día de hoy de más de 300.000 líneas de código, con más de 3.500 clases y mantenidas por más de 40 desarrolladores habituales.

Arquitectura

ACE está diseñado usando una arquitectura en capas para separar niveles de abstracción y reducir su complejidad facilitando su comprensión.

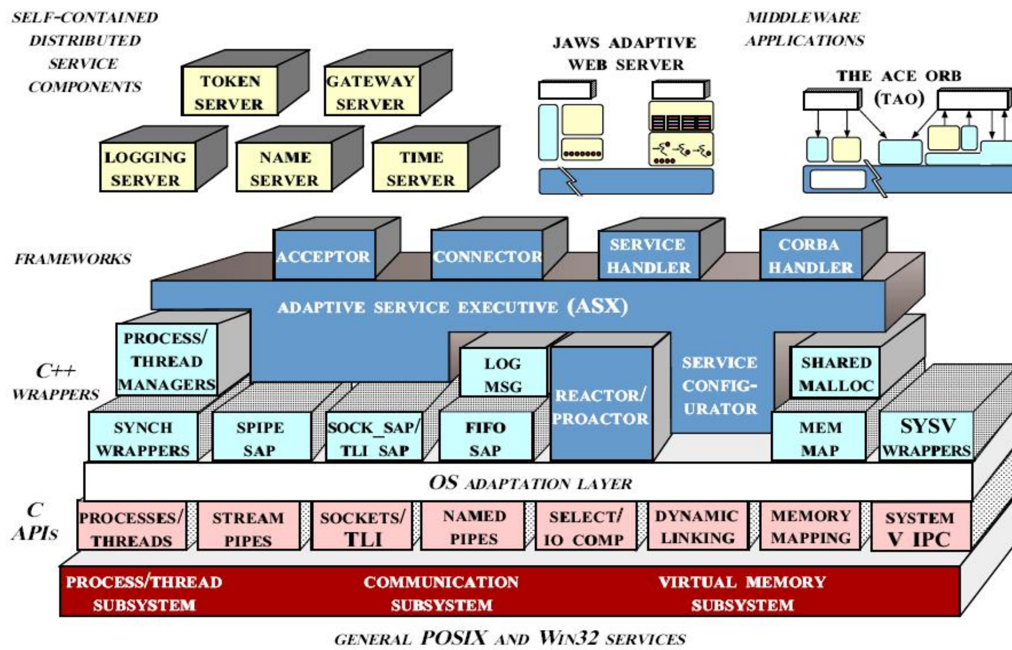


Ilustración 4: Estructura en Capas de ACE

Las capas más bajas contienen las partes de adaptación al sistema operativo y sus envoltorios en C++ orientado a objeto, y son las que encapulan de forma portable las servicios de comunicación, gestión de procesos y concurrencia de los sistemas operativos. Las capas altas proporcionan frameworks reusables, componentes orientados a servicios distribuidos y componentes de más alto nivel de computación distribuida. Juntas, todas las capas simplifican la creación, composición y configuración de sistemas de comunicación.

Capa de Adaptación al Sistema Operativo

Esta capa está directamente encima de las APIs nativas de los SO escritas en C. Separa a las otras capas de ACE de las dependencias específicas de cada SO en las siguientes áreas:

- Concurrencia y Sincronización

  Mecanismos multihilo, multiproceso y sincronización.

- Comunicación entre procesos (IPC) y memoria compartida.
- Demultiplexación de eventos.

  Síncrona o asíncrona, eventos basados en E/S, en contadores de tiempo, y en señales.

- Enlace Dinámico.

  DLL y *shared objects*.

- Sistemas de ficheros.

  APIs de los SO para manipulación de ficheros y directorios.

Capa de envoltorio C++ (Wrapper)

Se pueden programar aplicaciones altamente portables utilizando la capa anterior, pero esta capa simplifica el desarrollo de aplicaciones encapsulando y mejorando los servicios del SO con interfaces robustos usando los tipos del lenguaje C++.

Como los envoltorios C++ están fuertemente tipados, los compiladores pueden detectar violaciones de tipo en tiempo de compilación en lugar de en ejecución como suele ocurrir con el uso de APIs C. Las aplicaciones pueden combinar y componer las clases que proporciona esta capa heredando de, agregando y/o instanciando los siguientes componentes:

- Componentes de concurrencia y sincronización.
- Componentes IPC y de sistemas de ficheros.
- Componentes de manejo de memoria.

Capa de Frameworks

Los frameworks proporcionados por esta capa integran y mejoran los envoltorios C++, y sus componentes permiten desarollar aplicaciones y servicios de comunicaciones concurrentes y fácilmente configurables.

Los componentes de esta esta capa son:

- Componentes de demultiplexación de eventos.
- Componentes de inicialización de servicios.
- Componentes de configuración de servicios.
- Componentes en capas jerárquicas de streams.
- Componentes de Adaptación a ORB.

Veremos los diferentes frameworks que ofrece ACE con mayor profundidad en el presente documento.

Componentes distribuidos autocontenidos.

Además de las clases, servicios, componentes y frameworks comentados, ACE proporciona un conjunto de servicios que están distribuidos como paquetes con funcionalidad completa y plenamente utilizables y ejecutables.

Son por ejemplo un servidor de nombres o un servidor de tiempos plenamente operativos, que sirven además de para probar todas las capas subyacentes de ACE como ejemplos de utilización práctica de todo el entorno.

También se distribuyen con ACE un servidor Web de alto rendimiento y además ACE sirve como base a las mayores capacidades de abstracción proporcionadas por TAO (*The ACE ORB*), un ORB para aplicaciones en tiempo real que escapa del contexto del presente artículo.

Frameworks

En general los frameworks de ACE facilitan el desarrollo de software de comunicación que puede ser actualizado y extendido en funcionalidad sin tener que modificar, recompilar, reenlazar e incluso sin volver a lanzar los procesos.

Este alto grado de flexibilidad se consigue en ACE combinando las posibilidades que nos ofrece el lenguaje C++ como la herencia, la sobrecarga de métodos, el uso de plantillas y el enlace dinámico con la aplicación de patrones de diseño.

Reactor y Proactor

Muchas aplicaciones de comunicaciones se desarrollan como programas orientados a evento.

Los tipos de evento más comunes incluyen la actividad en los mecanismos IPC para operaciones de Entada/Salida, señales POSIX, gestión de señales en Windows y la expiración de contadores de tiempo.

Reactor está orientado al tratamiento síncrono de eventos, mientras que Proactor lo está para el asíncrono. Nos centraremos sólo en el framework Reactor que es uno de los utilizados en el software de Osmius.

El framework nos permite separar los conceptos de detección, demultiplexación y despacho de eventos , de su tratamiento. Implementa el patrón *Reactor*.

Se automatizan las siguientes tareas:

- Detección de eventos desde varios orígenes.
- Demultiplexación de eventos hacia los manejadores previamente registrados de esos eventos.
- Despacho a los métodos gancho (*hook methods*) definidos por los manejadores para procesar los eventos como definen los requisitos de la aplicación concreta que se está implementando.
- Portabilidad

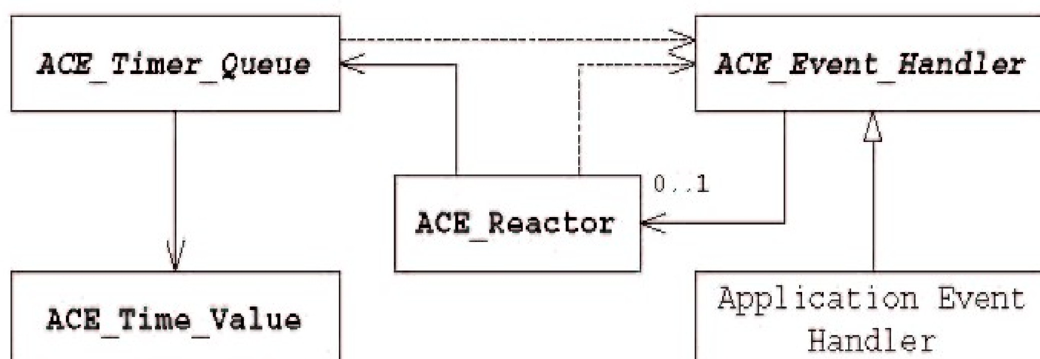


Ilustración 5: Clases Principales del framework ACE Reactor

Los beneficios de uso del patron *Reactor* son:

- Separación de Problemas.
- Modularidad, reusabilidad y configurabilidad.
- Control de Concurrencia.

La clase principal de este framework es ACE_Reactor e implementa el patrón *Facade* para definir el interfaz para las capacidades del framework:

- Centraliza el bucle de tratamiento de eventos en aplicaciones reactivas.
- Detecta los eventos a través del demultiplexor proporcionado por el SO y usado en la implememtación concreta del *reactor*.
- Despacha a los métodos gancho de los manejadores de eventos para que procesen según el tratamiento definido por la aplicación.
- Garantiza que cualquier hilo pueda cambiar el conjunto de eventos o encole una llamada a un manejador, y que pueda esperar una respuesta a su petición de forma rápida.

Este framework ha sido diseñado para que sea posible extenderlo y se ha separado el interfaz de la implementación utilizando el patrón *Bridge*. Hay más de una decena de implementaciones diferentes del *Reactor* en ACE siendo las más comunes éstas:

- **ACE_Select_Reactor**

  Utiliza la función síncrona de demultiplexación de eventos *select()* para detectar eventos de E/S y expiración de contadores de tiempo. Incopora también señales POSIX.

- **ACE_TP_Reactor**

  Usa el patrón *Leader/Follower* para extender el manejo de eventos de la implementación anterior usando un Grupo de Hilos (*Thread Pool Reactor*).

- **ACE_WFMO_Reactor**

  Mediante la función de Windows WaitForMultipleObjects() para la demultiplexación de eventos detecta eventos de E/S, *timeouts*, y eventos de sincronización de Windows.

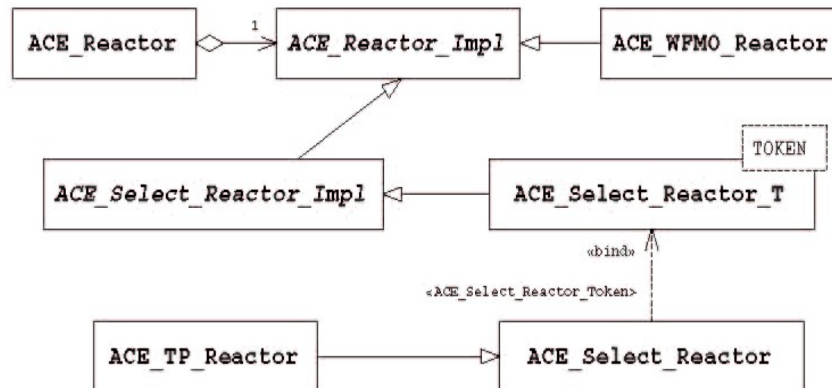


Ilustración 6: Implementaciones del ACE_Reactor

Es casi directo el realizar una nueva implementación del Reactor en nuestros programas reutilizando elementos e integrarla en este entorno.

Configuración de Servicio

Este framework (ACE Service Configurator) implementa el patrón *Configurator* y permite a las aplicaciones retrasar las decisiones de implementación y configuración de los servicios que ofrecen hasta más tarde dentro del ciclo de diseño de software, incluso en tiempo de instalación o de ejecución.

Tiene la habilidad de activar servicio de forma selectiva en tiempo de ejecución estén los servicios enlazados estáticamente o dinámicamente.

Gracias a que ACE tiene integrados los diferentes framework posibilita que los servicios diseñados usando este entorno puedan ser despachados por el framework ACE Reactor.

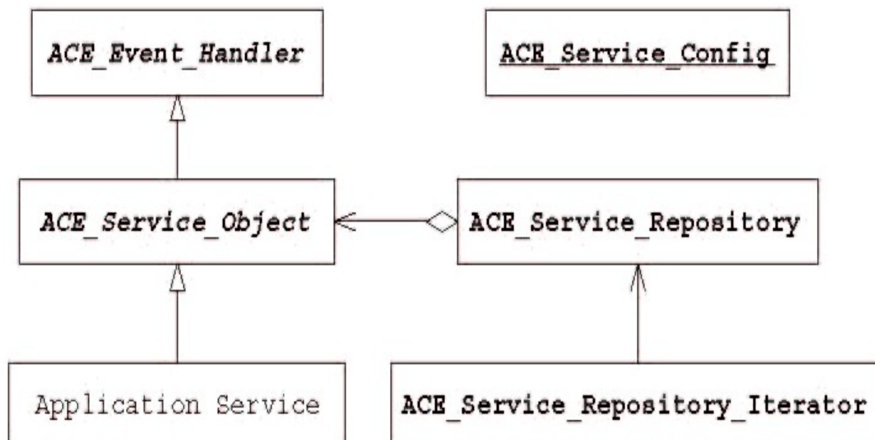


Ilustración 7: Clases Principales de ACE Service Configurator

Los beneficios de uso de ACE Service Configurator son:

- **Uniformidad**

  Mismo interfaz de configuración y control de los servicios.

- **Administración Centralizada**

  Agrupamos componentes y funcionalidad en una unidad administrativa lo que simplifica el desarrollo.

- **Modularidad y Reusabilidad**

  Separamos la implementación de componentes de la forma en que dichos componentes se configuran en procesos.

- **Dinamismo en la Configuración y el Control**

  Permitimos que los servicios puedan configurarse dinámicamente sin necesidad de recompilar o codificar.

Control de Tareas y Procesos

El framework ACE Task proporciona capacidades muy potentes en el ámbito de la concurrencia de procesos orientada a objetos, y nos permite lanzar varios hilos simultáneamente en el contexto de un objeto (*Active Object*).

También nos permite el paso de mensajes directo o través de colas entre objetos ejecutándose en hilos diferentes.

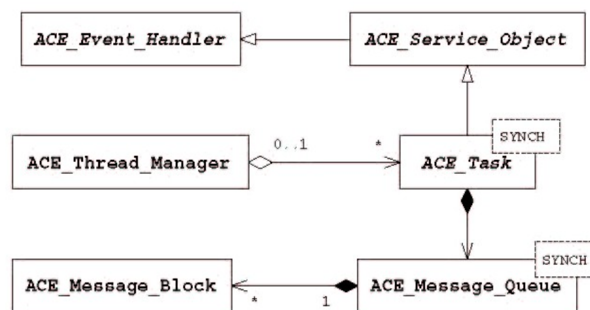


Ilustración 8: Clase Principales de ACE_Task

La clase ACE_Task contiene una cola de mensajes (ACE_Message_Queue) que se puede utilizar para separar el flujo de información de su procesamiento y para enlazar hilos que ejecutan servicios productores y consumidores de mensajes concurrentemente (Patrón *Producer/Consumer*).

Los beneficios de usar ACE_Task son:

- Mejora de la seguridad en los tipos de datos
- Mejora de la concurrencia problemas de sincronización simplificados.

  Se permite que los hilos ejecuten métodos asíncronos de forma simultánea. La sincronización se simplifica utilizando un planificador que evalúa las restricciones de sincronización.

- Aprovechamiento transparente del paralelismo disponible.

  Varios métodos de objetos activos pueden ejecutarse en paralelo si lo permite el SO y el hardware subyacente.

- El Orden de ejecución puede diferir del orden de invocación.

  Métodos llamados de forma síncrona pueden ejecutarse de forma síncrona de

acuerdo a las restricciones de la aplicación. Podemos agrupar llamadas y enviarlas o procesarlas todas juntas para mejorar el rendimiento.

Acceptor y Connector

Este framework implementa el patrón *Acceptor/Connector*, que mejora la reusabilidad del software y su facilidad para extender su funcionalidad separando las tareas necesarias para la conexión e inicialización de pares servicios de red, del procesamiento que han de realizar una vez conectados en inicializados.

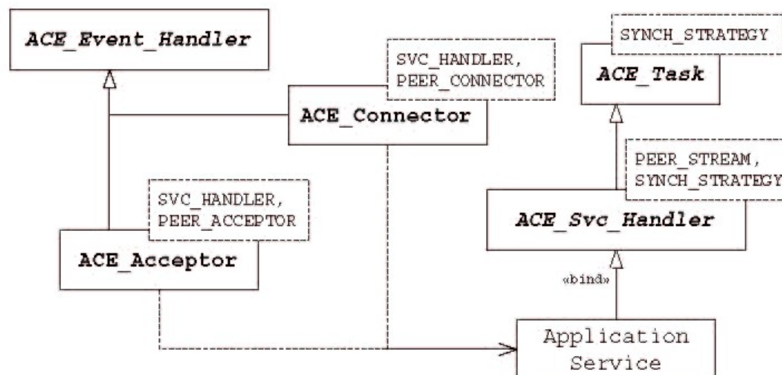


Ilustración 9: Principales Clases en Acceptor/Connector

Las clases principales son:

- **ACE_Svc_Handler**

  Representar uno de los extremos de conexión de un servicio y contiene un mecanismo IPC que es utilizado para comunicarse con un compañero de conexión.

- **ACE_Acceptor**

  Implementa un *factory* (creador de objetos) que espera de forma pasiva aceptando conexiones y cuando se produce una inicializa un *ACE_Svc_Handler* como respuesta a la petición activa de conexión.

- **ACE_Connector**

  Este *factory* se conecta de forma activa a un compañero de conexión tipo *Acceptor* e inicializa un *ACE_Svc_Handler* para comunicarse.

La clase ACE Acceptor nos ayuda a separar la conexión y la inicialización de un servicio, del tratamiento posterior de forma reusable para que los desarrolladores no tenga que reescribir este código repetidas veces.

Streams

Este framework se basa en el patrón *Pipes & Filters*, que simplifica el desarrollo de aplicaciones modulares en capas que pueden comunicarse de manera bidireccional entre los módulo de procesamiento.

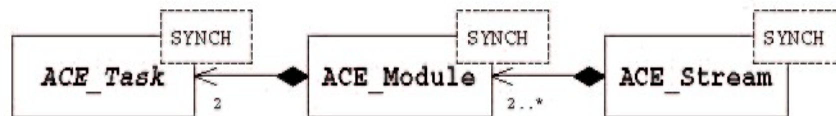


Ilustración 10: Principales Clases en ACE Streams

Define una arquitectura para procesar una corriente de datos en la que cada paso de procesamiento está encapsulado en algún tipo de filtro o componente.

Los datos se pasan entre filtro adyacentes a través del mecanismo de comunicación definido , que puede ser desde canales IPC conectando procesos locales o remotos hasta simples punteros que referencian a objetos dentro del mismo proceso. Cada filtro puede añadir, modificar o eliminar los datos antes de pasarlos al siguiente filtro. En ocasiones los filtros carecen de información de estado, en cuyo caso el paso de los datos se hace entre los filtros sin ser almacenados.

Con este framework ACE nos permite implementar esta estructura de forma sencilla y dinámica. Podemos añadir o quitar módulos de la cadena de tratamiento de los datos incluso en tiempo de ejecución basándonos en algunas capacidades de los frameworks anteriores. Podriamos añadir un módulo de depuración entre otros dos durante la ejecución del programa, en un hilo diferente, para ver los datos que se pasan entre ellos, y una vez depurarlo eliminarlo de la cadena.

El uso de los frameworks presentados se puede combinar dentro de la misma aplicación o servicios, y de hecho es lo que ocurre en la estructura de software de Osmius, como veremos posteriomente con más detalle. En concreto se utilizan los frameworks: Reactor, Task, Acceptor y Connector.

Osmius

Introducción

Osmius es un sistema de supervisión de elementos heterogéneos en red basado en una estructura de Servidor Central y Agentes, multiplatafoma y de código abierto. Osmius utiliza platafomas abiertas basadas en frameworks y patrones de diseño. En concreto utiliza el entomo de trabajo, ya presentado, ADAPTIVE Communication Environment (ACE) que permite la reutilización de código para múltiples platafomas con un rendimiento excelente incluso en tiempo real.

Tipos de Dispositivos o Instancias
Elemento de Red: Router, FireWall, Switch,...
Servidor: Linux, Windows, Solaris, ...
Base de Datos: Oracle, MySql, Sybase,...
Servicio Internet: http, ftp, DNS, ...
Industria: Sensor Temperatura, Vávula P
Domótica: Interruptores, Electrodomésticos,.

Tabla 1: Instancias

Para Osmius cualquier posible origen de un evento recibe el nombre de **Instancia**, sea ésta un dispositivo hardware final, un servidor, una base de datos, un servicio ftp o un sensor de presión o temperatura. Cada tipo de instancia tiene asociado un tipo de agente, que es el que sabe cómo preguntar por eventos a sus instancias y se integra perfectamente en el resto de la arquitectura Osmius para enviar los eventos y recibir órdenes e indicaciones sobre los eventos.

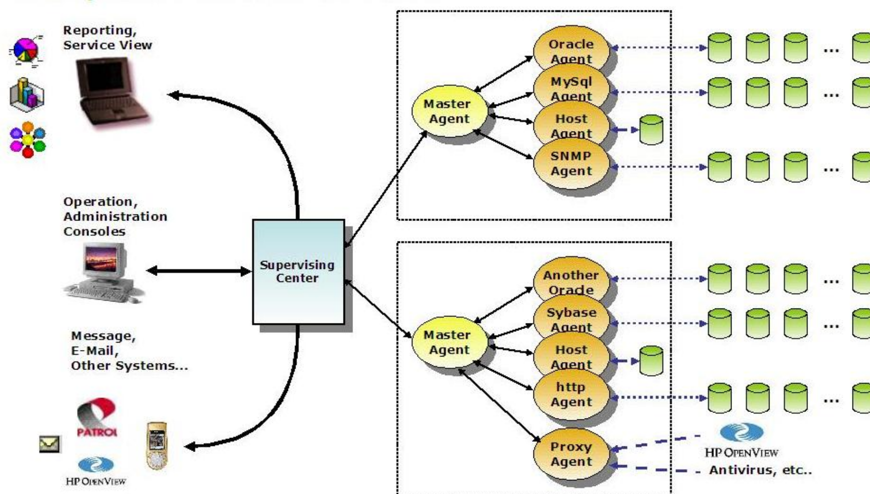


Ilustración 11: Arquitectura Osmius

Osmius está subvencionado por el Ministerio de Industria en el año 2006.

Los agentes están integrados en la arquitectura a través de **Agentes Maestros** que además se encargan de recoger y enviar el **Servidor Central** los eventos y de recibir comandos y pasarlos a cada uno de los agentes.

En Osmius toda la organización lógica reside en el **modelo de datos** del Servidor Central lo que nos permite que la orientación más técnica – Dispositivo o Instancia – o más orientada a Usuario – Servicio o Proceso de Negocio – sea una cuestión de presentación según el perfil del usuario conectado.

De esta manera se consigue **independizar la arquitectura del sistema** de monitorización de las instancias a monitorizar. Los agentes son poco intrusivos y al tener un interfaz muy definido es muy fácil desarrollar e incorporar nuevos tipos de instancias a monitorizar.

La potencia de Osmius es - además de que el motor está basado en programación OO, frameworks y patrones de diseño – su modularidad y la capacidad de creación rápida, sencilla y robusta de nuevos agentes capaces de supervisar variables de nuevos tipos de instancia.

Mediante estas técnicas Osmius consigue:

- Reutilización Efectiva de Código.
- Reutilización Efectiva de Algoritmos.
- Código Multiplataforma.
- Adaptabilidad a nuevas necesidades y/o plataformas tanto hardware como software.

Osmius se ha diseñado desde un primer momento para poder **incorporar nuevos agentes e instancias de forma muy fácil** y sin tener que aprender todo el bagaje de patrones de diseño, frameworks, clases y arquitectura que subyace a todo el conjunto. Su arquitectura modular permite además su conexión con sistemas de mensajes - como SMS y otros – y otros sistemas de supervisón existentes en el mercado, además de poder incorporar motores de correlación en el servidor central.

En este sentido **Osmius proporciona un entorno o framework** para gestionar eventos y alarmas e implementar la monitorización de un conjunto de sistemas, un sistema de control industrial en el mercado energético o de comunicaciones, sistemas de domótica y videovigilancia y en general sistemas distribuidos en red orientados a eventos.

Funciones:

- Recolectar datos y señales de tipos diversos de sistemas o instancias.
- Actuar y ejecutar acciones sobre dichos sistemas.
- Concentrar toda la información recibida y enviada y presentarla al usuario en tiempo real y mediante informes.
- Conectarse con otros sistemas del mercado.

Al ser una iniciativa de software libre basada en licencia GPL, este código y su documentación y análisis estarán al alcance de cualquiera que quiera aportar o adaptar el software a sus propios sistemas. Esto permite que se orqueste una

comunidad de investigadores, desarrolladores y productores de software y hardware a su alrededor, mejorando la calidad final del producto y su potencial de uso.

Arquitectura

Osmius opta por una arquitectura Gestor-Agente con agentes ligeros, y por agrupaciones lógicas por perfil para cumplir con las espectativas de orientación técnica para los usuarios del sistema y de orientación a negocio para los clientes o dirigentes de negocio.

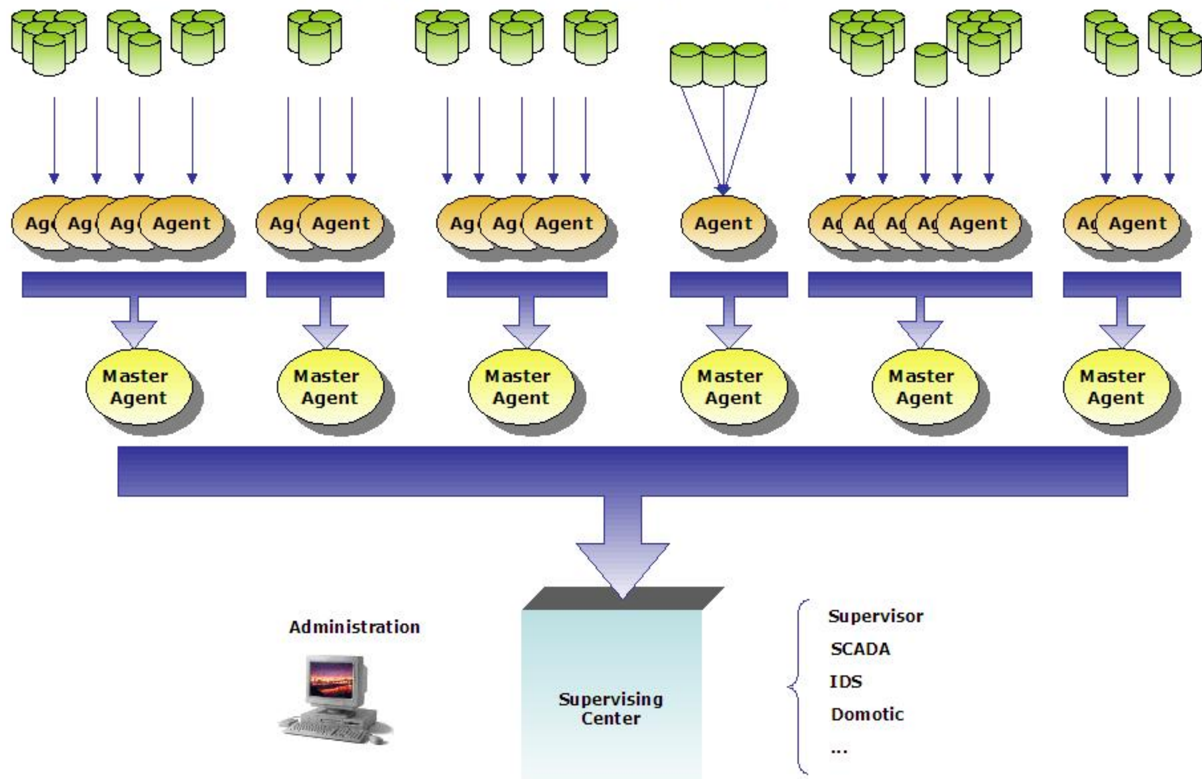


Ilustración 12: Arquitectura Osmius

Los tres elementos fundamentales de las arquitectura de Osmius son los que aparecen como principales en la Ilustración 12. En el análisis realizado para el desarrollo del producto se ha puesto especial hincapié en diseñar cada proceso atendiendo a tres aspectos:

- Requisitos Funcionales:

  Funcionalidad que debe cumplir el elemento de cara a la monitorización de procesos y a los usuarios finales por un lado, y por el otro centrándonos en un usuario más técnico que utilice el entorno de Osmius para desarrollar nuevos agentes o nuevas funcionalidades, en definitiva para desarrolladores usuarios que utilizarán Osmius como un franework en sí mismo.

- Requisitos de Calidad de Servicio:

  Se han detallado elementos como las necesidades de velocidad de

procesamiento, el aprovechamiento o no del posible paralelismo ofrecido por el hardware y software huésped y elementos que debían ser parametrizables en contraposición a los fijos.

- Servicios de Cada Elemento:

  Cada componente se ha dividido, antes de empezar con el diseño detallado y con la codificación, en grupos de funcionalidades lo más independientes posible entre ellos, para así dividir los esfuerzos de desarrollo y elevar los índices de reusabilidad a la vez que reducimos la complejidad para comprender y utilizar el entorno.

En los siguientes apartados pasamos a describir cada uno de estos elementos y los servicios de que se componen para más adelante presentar los ejemplos de uso de los Patrones de Diseño.

Centro de Supervisión

En el Centro de Supervisión se concentran todos los mensajes recogidos por los agentes maestros de los agentes que, a su vez, controlan. Además desde él se gestiona toda la infraestructura que nos permite desplegar los agentes, realizar instalaciones y actualizaciones remotas y cambiar o recoger la configuración de todo el sistema de recolección de eventos. Toda la parte de envíos de configuración, control de órdenes para la parada o rearranque de agentes es controlada desde el Gestor de Comandos.

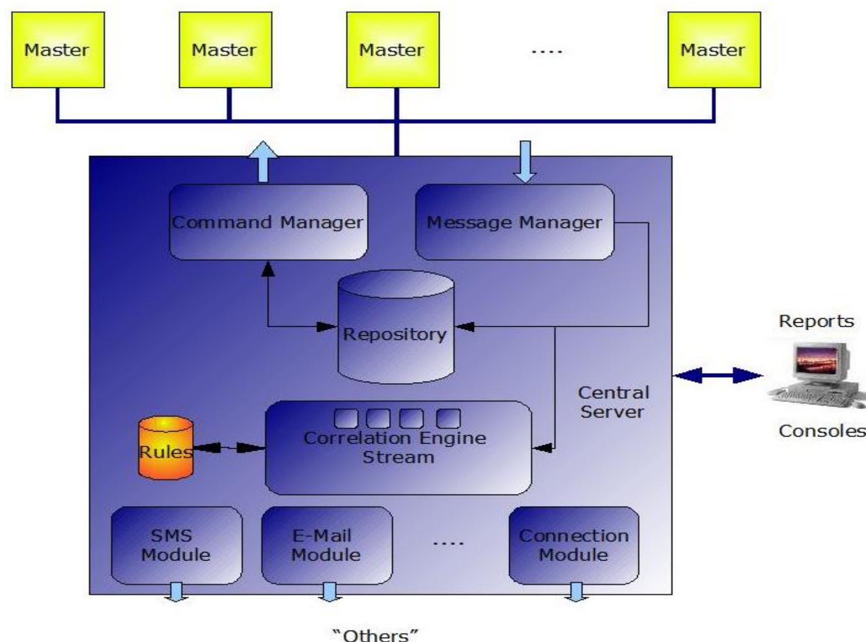


Ilustración 13: Arquitectura de Centro de Supervisión de Osmius

Consta de un repositorio central soportado con una base de datos en la que están los eventos activos, el histórico de eventos y los datos de infraestructura y configuración. A este repositorio se accede a través de las consolas de operación, administración y control de calidad o cuadro de mando. Estas consolas están implementadas en Java en una arquitectura de tres capas.

Por otro lado y para poder enviar mensajes y alertas a otros sistemas existen módulos especializados para cada uno de ellos a conectar, con la idea de que sea sencillo hacer un nuevo módulo, aplicando además reusabilidad para el código común.

De los diferentes componentes del Centro de Supervisión nos centraremos en este artículo en el Gestor de mensajes o “Message Manager”.

Gestor de Mensajes

- Requisitos Funcionales:

  Es el encargado de recoger los mensajes de todos los agentes maestros, que a su vez los han recogido previamente de los agentes de instancia que controlan.

  Otros procesos después de él se encargarán de procesar esos mensajes y transformarlos en eventos de alarma, aviso o informativos y de aplicarles las trasformadas de correlación. El gestor de mensajes debe tener poca carga lógica para poder responder con el máximo rendimiento a las peticiones de envío de los agentes maestros y garantizar que los datos se almacenan correctamente en la base de datos del repositorio.

- Requisitos No-Funcionales:

  Para poder optimizar el rendimiento debe ser capaz de aprovecharse de las capacidades de concurrencia del SO y Harware en que se ejecute. Uso de hilos y control de la sincronización.

  Como es previsible que los agentes maestros estén enviado constantemente mensajes, es útil que las conexiones sean permanentes y con capacidad de reconexión.

  Realizar el procesamiento muy rápido y con poca complejidad para dejar a los agentes hacer su trabajo sin preocuparse más del mensaje enviado.

  Debe ser escalable en previsión de tener que proporcionar sus servicios a un número muy elevado de agentes maestro. Mantener las operaciones con complejidades reducidas y nunca exponenciales en función del número de agentes maestros o del número de eventos recibidos.

Agente Maestro

Se encuentra en el centro de la arquitectura entre el centro de supervisión y los agentes especializados en monitorizar variables de un tipo o tipos de instancia determinados. Debe ser multiplataforma y muy robusto, de manera que permita el crecimiento del producto Osmius añadiendo nuevos módulos al centro de supervisión por un lado, y por el otro creando y añadiendo a la infraestructura nuevos agentes para nuevos tipos de instancia. Al agente maestro le es transparente el tipo de agente que controla. Sabe cómo arancarlo, controlar su estado, enviarle comandos (sin comprenderlos) y recibir sus mensajes cualesquiera que sean.

Además es el encargado de recibir las órdenes del centro de supervisión,

ejecutarlas e informar de su resultado. Las órdenes típicas la instalación de un nuevo tipo de agentes, cambio en la configuración propia o de un agente controlado por él, informar del estado de los agentes y pararlos o arrancarlos.

Como el resto de procesos cuenta con un servicio de configuración encargado de leer, comprobar y mantener disponible para los otros servicios la información de configuración.

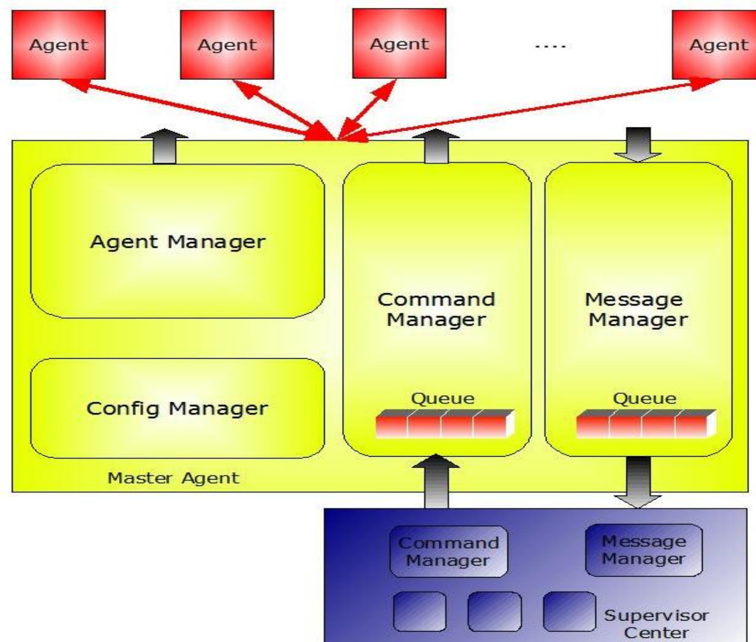


Ilustración 14: Arquitectura de Agente Maestro Osmius

Gestor de Mensajes

- Requisitos Funcionales:

  Es el encargado de recoger los mensajes de todos los agentes que dependen del agente maestro. En ningún caso los interpreta, sólo añade la información de fecha y hora de recepción para enviarlos al Centro de Supervisión.

  Mantiene disponibles las conexión y espera la llegada de datos de forma pasiva.

- Requisitos No-Funcionales:

  Los agentes deben dedicarse a la recolección de datos de sus instancias asociadas y, por lo tanto, no esperar más que el mínimo necesario para poder enviar sus mensajes. Esta exigencia ya indica la utilidad de realizar el tratamiento de los mensajes de forma asíncrona y aplicar el patrón de diseño Half-Sync/Half-Asynch para separar la recepción de mensajes de su tratamiento o envío final.

  Como es previsible que los agentes maestros estén enviado constantemente mensajes, hacer que las conexiones sean permanentes pareciera de utilidad.

Realizar el procesamiento muy rápido y con poca complejidad para dejar a los agentes hacer su trabajo sin preocuparse más del mensaje enviado.

Debe ser escalable en previsión de tener que proporcionar sus servicios a un número muy elevado de agentes. Mantener las operaciones con complejidades reducidas y nunca exponenciales en función del número de agentes maestros o del número de eventos recibidos.

Gestor de Comandos u Órdenes

➔ Requisitos Funcionales:

Acepta comandos desde el centro de supervisión bien para procesarlos él mismo directamente o enviarlos a los agentes y sean ellos los encargados de hacerlo. Podrá recibir órdenes de parada, arranque , estado, reconfiguración de sí mismo y de los agentes controlados.

Permite que a través de él se puedan desplegar nuevos agentes o sus actualizaciones y así independizar y facilitar tanto el desarrollo de nuevos agentes como la integración en infraestructuras existentes.

➔ Requisitos No-Funcionales:

No se preven altos índices de ejecución de comandos - al revés de lo que ocurre con los mensajes. Además en este caso la concurrencia en la ejecución de los mismos es probable que no sea una ventaja; la ejecución en serie de cada uno de los comandos recibidos tendrá su sentido e incluso puede haber dependencias entre ellos.

Sí es útil que el procesamiento de las órdenes pueda realizarse en su propio hilo para no perturbar a los otros servicios del agente maestro y, eventualmente, aprovecharse del paralelismo ofrecido por el SO y la plataforma Hardware de las capas inferiores.

Gestor de Agentes

➔ Requisitos Funcionales:

Es el encargado de arrancar los agentes que dependen del agente maestro con los adecuados parámetros de inicio que se leerán de los ficheros de configuración.

También les pasará la información de los puertos locales de comunicación para la recepción de órdenes.

Monitoriza el estado de los agentes y mantiene referencias a ellos de forma que ante una caída abrupta de uno de ellos informará a través del gestor de mensajes.

➔ Requisitos No-Funcionales:

Aprovechará las ventajas de las APIs de cada SO para el lanzamiento y control de procesos. No debería consumir mucho tiempo de proceso en comparación con los otros procesos.

Gestor de Configuración

- Requisitos Funcionales:

  Lee, parsea, comprueba y almacena tanto los parámetros recibidos como los que se encuentran en los ficheros de configuración.

  Permite que el resto de servicios tenga un acceso fácil y rápido a los valores de los diferentes parámetros.

- Requisitos No-Funcionales:

  Encapsula las diferencias en el tratamiento y lectura de ficheros de configuración de las diferentes plataformas y SO.

  Proporciona una capa de acceso de los valores portable y utilizable entre plataformas.

Agente de Supervisión

Su principal cometido es ejecutar determinadas acciones cada una con un periodo determinado contra cada una de las instancias que debe supervisar. Estas acciones ejecutadas contra una instancia (sea ésta una base de datos, un servidor http, o un sensor de temperatura) devuelven un valor o un error de ejecución que se añade a una estructura tipo mensaje que es enviada al agente maestro local del cual dependemos.

El agente es capaz de responder a acciones u órdenes del agente maestro que le controla como son arrancar, parar o releer los ficheros de configuración.

La **principal idea de Omius** es que desarrollar un nuevo agente ser reduzca a crear un conjunto de eventos específicos del tipo de instancia y a reutilizar todo el resto de código común que le permite integrarse en la arquitectura software. Incluso alguien sin profundos conocimientos de programación orientada a objeto ni de los patrones ni frameworks utilizados pueda, en un periodo corto de tiempo, desarrollar, probar e integrar nuevas instancias en la infraestructura que Osmius proporciona.

Los servicios comunes de cualquier agente se encargan de procesar las órdenes recibidas, de enviar los mensajes hacia el agente maestro que los controla y leer, tratar, recargar y almacenar la información de configuración.

El servicio que varía en cada agente es el que se encarga de crear los objetos instancia adecuados y sus eventos asociados desde la configuración es el Gestor de Instancias. En este servicio se ha separado la implementación del interfaz, usando patrones como *Bridge,* para conseguir el objetivo de aumentar el código reusado frente al nuevo.

Un desarrollador interesado en recibir mensajes o eventos de una nueva instancia consistente en una vávula de presión conectada a un conducto de fluidos, debe derivar de dos clases e implementar una serie de métodos virtuales sin más.

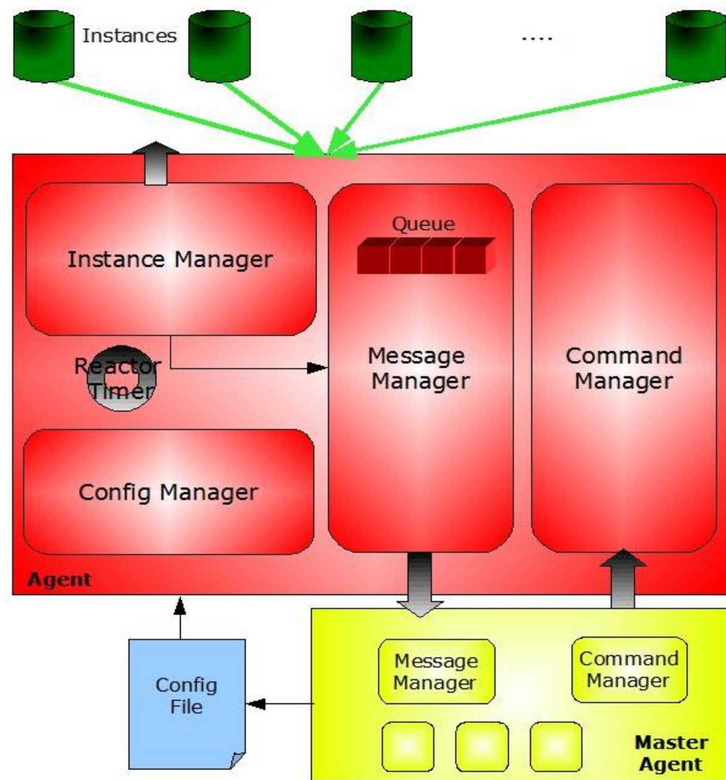


Ilustración 15: Arquitectura Servicios Agente Osmius

Gestor de Instancias

➔ Requisitos Funcionales:

Es el servicio principal de cada agente y el que está preparado para poder ser reutilizado en el desarrollo de nuevos agentes.

Es el encargado de crear los objetos adecuados de Instancia y Eventos (y acciones asociadas) desde los ficheros de configuración. Una vez creados los registra con el Reactor que proporciona ACE, para que sea él el que despache de vuelta a los métodos encargados de recuperar el valor de cada evento contra la instancia correcta. Una vez recogidos los valores, cada evento se encarga de pasarlos al servicio común de Gestión de Mensajes para que se encole y se envíe primero al Agente Maestro y finalmente al Centro de Supervisión.

Es capaz de cambiar toda la estructura de objetos instancia y evento creada, por cambios en la configuración para volver a registrar los eventos en los periodos correctos.

Básicamente ejecuta cada T segundos cada uno de los Eventos E, contra la instancia adecuada I, crea un mensaje M y lo encola a través del servicio adecuado.

➔ Requisitos No-Funcionales:

Cada evento no debería consumir mucho tiempo de proceso y si fuera así

deberá contarse con un timeout para abortar acciones excesivamente lentas informando adecuadamente de la situación.

Este servicio está estrechamente relacionado con el bucle de eventos que debe tratar los eventos de expiración de tiempos. El bucle de eventos es interesante que pueda ejecutarse en varios hilos a la vez para prevenir que un evento impida la ejecución de los demás. El patrón *Leader/Follower* con un *Thread Pool* (grupo de hilos) parece encajar en esta situación.

Gestor de Mensajes

- ➔ Requisitos Funcionales:

  Es el encargado de proporcionar un interfaz para recoger los mensajes de todos los eventos según expiran y se ejecutan en el servicio anterior.

  Mantiene disponibles la conexión con el agente maestro y espera la llegada de datos de forma pasiva.

- ➔ Requisitos No-Funcionales:

  No hacer esperar a los eventos para poder enviar sus mensajes. Contorlar la longitud de la cola de mensajes.

  Hacer que las conexión sean permanente con el agente maestro con capacidades de reconexión cada 2^n segundos.

  Mantener las operaciones con complejidades reducidas y nunca exponenciales en función del número de agentes maestros o del número de eventos recibidos.

  El envío de bloques de mensajes en lugar de uno a uno puede estar muy indicado para mejorar el rendimiento.

Gestor de Comandos u Órdenes

- ➔ Requisitos Funcionales:

  Acepta comandos desde el agente maestro para procesarlos e informar del resultado y en algún caso indicar al resto de servicios, por ejemplo de que ha cambiado la configuración. Las órdenes pueden ser de parada, arranque , estado, y de reconfiguración.

- ➔ Requisitos No-Funcionales:

  No se preven altos índices de ejecución de comandos - al revés de lo que ocurre con los mensajes. Ejecución en serie de cada uno de los comandos recibidos.

  Sí es útil que el procesamiento de las órdenes pueda realizarse en su propio hilo para no perturbar a los otros servicios del agente y, eventualmente, aprovecharse del paralelismo ofrecido por el SO y la platafoma Hardware de las capas inferiores.

Gestor de Configuración

- ➔ Requisitos Funcionales:

  Lee, parsea, comprueba y almacena tanto los parámetros recibidos como los que se encuentran en los ficheros de configuración.

  Permite que el resto de servicios tenga un acceso fácil y rápido a los valores de los diferentes parámetros.

- ➔ Requisitos No-Funcionales:

  Encapsula las diferencias en el tratamiento y lectura de ficheros de configuración de las diferentes plataformas y SO.

  Proporciona una capa de acceso de los valores portable y utilizable entre plataformas.

Resúmen de Características

Además de cubrir una necesidad dentro de los sistemas de recolección de eventos desde elementos distribuidos en red, proporciona un entorno donde desarrollar y probar resultados y teorías de investigación en el campo de las ciencias de la computación.

Característica	Descripción
Software Libre	Permite contrastar la calidad del producto, favorece su modularidad y que una comunidad de desarrollo aúne esfuerzos para mejorar y crear nuevas áreas. Apoya la difusión del producto y democratiza su uso.
Orientado a Objeto y al uso de Patrones de Diseño	Se centra en el código específico de la aplicación reusando código ya escrito y probado. Reutiliza diseños probados y aporta experiencia. Permite investigar a la vez que se desarrolla y fomentar la relación universidad con la empresa.
Uso de ACE	Reutilización de código probado. Recibe ideas y mejoras de la comunidad de ACE, y aporta realimentando beneficios mutuos. Independiente de Plataformas y Sistemas Operativos.
Modularización basada en Objetos	Permite el desarrollo de agentes para nuevos tipos de software y/o hardware de forma muy sencilla. Puede avanzarse en varios frentes sin conflictos.
Arquitectura Abierta	Permite añadir nuevos módulos a empresas cliente, empresas de desarrollo y a comunidades en Internet
Estándares Abiertos	El proceso de análisis y diseño se hace en comunidades abiertas, y sus resultados están disponibles para su revisión. Se asegura la calidad mediante el control de cambios y sus procesos de pruebas.
Orientado a Arquitectura vs Producto	El motor de Osmius permite implementar un sistema de Supervisión o un SCADA, o cualquier software que comparta estas arquitecturas, reutilizando el diseño y el análisis y por ende el código. Más robusto y fiable a medio y largo plazo.

Tabla 2: Características Distintivas de Osmius

Patrones Utilizados

Hasta este punto dentro del presente documento se han introducido los conceptos necesarios para comprender el funcionamiento general y los objetivos de:

- El dominio de **aplicaciones para la supervisión** y monitorización de sistemas.
- **ACE y los frameworks** que ofrece para el desarrollo de aplicaciones distribuidas en red .
- **Osmius** como aplicación concreta para recolección de eventos basada en ACE.

A continuación se expone el uso de los patrones de diseño en el contexto de Osmius y ACE, explicando las necesidades que han llevado a su elección y uso, y las ventajas concretas.

Wrapper Facade

Descripción

Es un patrón de diseño para encapsular las funciones y datos proporcionados por APIs existentes no orientadas a objeto, dentro de interfaces de clases orientadas a objeto aportando los beneficios de ser más concisas, robustas, mantenibles y portables.

Contexto

Dos de los tres principales componentes de Osmius como son el Agente Maestro y los Agentes para la monitorización de variables en las diferentes instancias tienen como requisito claro el poder ejecutarse en diferentes plataformas hardware y de sistema operativo. Necesitamos poder distribuir nuestro código y compilarlo (estamos tratando con C++) en por ejemplo, Linux, Windows y Solaris sin modificaciones o con muy poco trabajo extra.

Al ser Osmius una aplicación distribuida en red hace uso de las llamadas del SO para comunicaciones en red a través de sockets y otros mecanismos, de llamadas para el tratamiento y lectura de ficheros, de mecanismos para el lanzamiento, control y terminación de procesos e hilos, y de funciones para demultiplexar la ocurrencia de eventos en conjuntos de uno o varios recursos.

Estas funcionalidades suelen estar disponibles a través de las APIs – normalmente en C no orientado a objeto – de los diferentes SO. Programar usando estas APIs es difícil de aprender, tedioso y poco portable.

ACE proporciona unos interfaces comunes en C para los diferentes SO, que luego encapsula mediante clases C++ aplicando el patrón *Wrapper Facade* que nos aporta los siguientes beneficios en el caso de Osmius:

- Sólo aprendemos una vez el interfaz en lugar de con cada SO.

- Portabilidad garantizada.
- Detección de posibles problemas con los tipos en tiempo de compilación.
- Podemos dejar para el tiempo de instalación e incluso de ejecución parámetros de comportamiento de los diferentes mecanismos.

Con este patrón nos evitamos el uso directo de las APIs de bajo nivel de los diferentes sistemas operativos.

Ejemplo de uso

Osmius utiliza las clases envoltorio que proporciona ACE para asegurarse que sus componentes corran en diferentes plataformas.

Por ejemplo al utilizar Osmius la ejecución de procesos en diferentes hilos necesita de mecanismos portables y con el mismo interfaz para garantizar la exclusión mutua en determinadas partes del código y en el acceso a recursos.

La clase que utilizamos es ACE_Thread_Mutex cuyos principales métodos tienen la siguiente interfaz:

```
int     remove (void)
int     acquire (void)
        Acquire lock ownership (wait on queue if necessary).
int     acquire (ACE_Time_Value &tv)
int     acquire (ACE_Time_Value *tv)
int     tryacquire (void)
int     release (void)
```

La implementación del método acquire() para Solaris es:

```
int acquire(void) {
        return mutex_lock(mutex_);
}
```

,y existen implementaciones para otras APIS como VxWorks, LynxOS, Windows o hilos POSIX.

El uso dentro del código es:

```
ACE_Thread_Mutex osmius_mutex;
.....
osmius_mutex.acquire();
.....
funcion_protegida_y_sincronizada();
......
osmius_mutex.release();
```

Otras clase Wrapper Facade de ACE que usamos en Osmius encapsulan las llamadas para Sockets, gestión de procesos e hilos, operaciones de tiempo, etc.

El uso de este patrón está muy extendido en Osmius y las propias clases de ACE se basan en él para proporcionar servicios de más alto nivel, permitiendo un uso implícito sólo por usar algunos de los framworks que ACE proporciona.

Reactor

Descripción

Es un patrón de arquitectura que nos permite a aplicaciones orientadas a eventos demultiplexar y despachar las peticiones de servicio que llegan desde uno o más clientes.

Contexto de uso

Tanto en el Centro de Supervisión como en el Agente Maestro se repite la situación que nos permite aprovechamos de este patrón. Ambos procesos tiene un servicio para la recepción de mensajes a través de varios sockets. En el caso del centro de supervisión recibiremos conexiones y datos de múltiples agentes maestros y, en el caso de éstos últimos, recibirán peticiones de conexión y de envío de mensajes de los agentes gestionados.

Los agentes también manejan varios orígenes de eventos como son los sockets para la recepción de órdenes y un nuevo tipo de origen constituido por las expiraciones de los contadores de tiempo asociados a los periodos de monitorización de las diferentes variables de las instancias monitorizadas.

Necesitamos quedarnos a la espera de la aparición de eventos en determinados recursos, detectar su aparición, enviar o despachar su tratamiento a los métodos adecuados y realizar el tratamiento adecuado en cada caso.

Además de nuevo nos encontramos con el requerimiento multiplataforma y la deseada reusabilidad, con lo que programar los mecanismos de demultiplexación de forma fija para determinada plataforma nos impedirá alcanzar los objetivos además de perder tiempo y recursos.

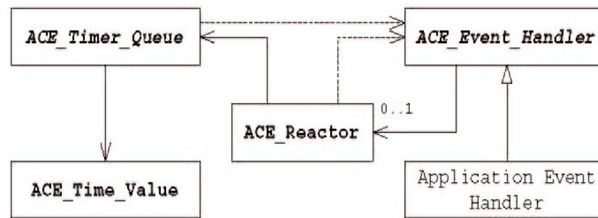
Beneficios en el caso de Osmius:

- Reutilización del código no específico.
- Claridad de separación de demultiplexación y despacho de eventos, de su tratamiento. Nos centramos en codificar éste último.
- Portabilidad.

Ejemplo de uso

En este patrón hay que cambiar la mentalidad típica de programación de aplicaciones en las que es la aplicación la que programa el bucle de captura de eventos y se encarga de llamar de manera activa a las funciones para el tratamiento de cada evento.

Este patrón y el framework en que se encuadra dentro de Osmius implementan **inversión de control**. Por inversión de control entendemos que es el framework el que llama a determinados métodos gancho de objetos que habremos previamente registrado con el Reactor, al ocurrir los eventos.



En el caso de los eventos Osmius que deben ejecutarse al vencer determinado periodo de tiempo lo primero que hacemos en crear nuestra clase haciendo que sea derivada de ACE_Event_Handler utilizando la herencia.

```
class OSM_Event : public ACE_Event_Handler
{
public:
  // Constructor al que le pasamos un reactor por si queremos uno distinto del de
  // por defecto.
  OSM_Event (ACE_Reactor *r= ACE_Reactor::instance());

  // Método gancho que se llama siempre así que será llamado cuando expiren los contadores
  // de tiempo registrador.
  // ACE_Time_Value es otra clase de la Wrapper Facades para encapsular el tratamiento de
  // tiempos y fechas de forma portable.
  virtual int handle_timeout (const ACE_Time_Value &tv,
                              const void *);
  // Tratamiento específico de inicialización para Osmius.
  int open(const ACE_TCHAR* name, const OSM_Instance_Base* instance,
           const int  interval , const int i_delay);

  // Cierre del objeto.
  int close(void);

  // Método gancho llamado cuando este objeto se saca del framework del reactor y es
  // destruido.
  virtual int handle_close   (ACE_HANDLE = ACE_INVALID_HANDLE,
                              ACE_Reactor_Mask = 0);

private:
   ..........
   ..........
   ..........
};
```

En la implementación de handle_timeout() ponemos el tratamiento específico de nuestra aplicación, en nuetro caso ejecutamos la acción contra la instancia que nos devuelve un valor que metemos en un mensaje que le pasamos al manejador de mensajes MsgManager a través de su método put().

```
int
OSM_Event::handle_timeout(const ACE_Time_Value &tv, const void *)
{
  ......
  ......
  ACE_Time_Value      time_before = ACE_OS::gettimeofday();
  // Creamos el mensaje y le ponemos los campos iniciales
  OSM_Message*        osmius_message=0;
  ACE_NEW (osmius_message, OSM_Message);
  osmius_message.typ_message    ("N", 1);
  ......
  ......
  ACE_Time_Value time_after;
  // Ejecutamos la acción que sea contra la instancia y recuperamos el valor en "value".
  this->action_.execute(this->cmd_line_,ACE_OS::strlen(this->cmd_line_),
                           this->timeout_,
                           val_text      ,
                           &val_len      ,
```

```
                              &value )

  // Ponemos el OSM_Message dentro de un objeto ACE_Message_Block para su envío.
  ACE_Message_Block* mblk=0;
  mblk = (ACE_Message_Block*) &osmius_message;
  // Prepare the message to be sent.
  osmius_message->encode();

  // Enviamos el mensaje al agente maestro o donde lo envíe ins_manager con put().
  // The pointer to the instance manager is in our osm_instance owner.
  this->osm_instance_->ins_manager()->put(mblk,0)

  return 0;
};
```

Por último tenemos que registrarnos con el reactor para que nos llame cada vez que venza el periodo de tiempo (*interval*) asignado al evento.

```
int
OSM_Event::open(const ACE_TCHAR* name, const OSM_Instance_Base* instance,
                const int    interval, const int i_delay);
{
       ACE_OS::strncpy(this->name_, name,OSM_Message::CODLEN);
       this->cod_message_[OSM_Message::CODLEN]=0;
       this->osm_instance_ = instance;

       if (interval > 0)
              this->interval_ = interval;
       else
              this->interval_ = 5;

       if (initial_delay > 0)
              this->initial_delay_ = initial_delay;
       else
              this->initial_delay_ = 5;

       // Nos registramos con el reactor a través de la llamada para contadores de tiempo
       // que se llama schedule_timer().
       ACE_Time_Value tv_interval(this->interval_);
       ACE_Time_Value tv_delay(this->initial_delay_);
       this->reactor()->schedule_timer(this, 0, tv_delay,tv_interval)
       ......
       ......

       return 0;
}
```

Si queremos aplicar este patrón orientado a cuando se reciban datos de red en determinado socket como es el caso en el servicio que recoje los mensajes en el Master Agent para a su vez enviarlos al Centro de Supervisión:

Declaramos la clase.

```
class ACE_Svc_Export OSM_MA_MsgReceiver
  : public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
// Derivamos de ACE_Svc_Handler que es a su vez derivada de ACE_Event_Handler utilizando
// templates para indicar que utilizaremos sockets para la comunicación y que no queremos
// sincornización ya que no vamos a utilizar en este caso multihilo.
{
public:
  // Constructor.
  OSM_MA_MsgReceiver (OSM_MA_MsgSender *handler = 0)
    : msg_sender_ (handler) {}

  // Método gancho de inicialización que será llamado cuando un agente haga una nueva
  // conexión.
  virtual int open (void *);

  // Método gancho para cierre y para desregistrarse con el reactor.
```

```
  virtual int close (u_long = 0);



protected:
  // Método gancho que se llamará cuando un cliente (agente) tenga un mensaje que
  // enviar (con send()).

  virtual int handle_input (ACE_HANDLE handle);

  // Método gancho que se llama cuando un cliente termina.
  // Cerramos el socket y quitamos el cliente de "connected_clients_".
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);

  // Puntero al servicio MessageSender.
  OSM_MA_MsgSender *msg_sender_;

  // Lleva la cuenta de los descriptores de los clientes conectados.
  ACE_Handle_Set connected_clients_;
};
```

Registro con el Reactor para cuando hay datos en el socket en el método open():

```
int OSM_MA_MsgReceiver::open (void *)
{
  ACE_HANDLE handle = peer ().get_handle ();
  if (reactor ()->register_handler
        (handle, this, ACE_Event_Handler::READ_MASK) == -1)
    return -1;
  connected_clients_.set_bit (handle);
  return 0;
}
```

Tratamiento en handle_input():

```
int OSM_MA_MsgReceiver::handle_input (ACE_HANDLE client_handle)
{
  ACE_Message_Block   *mblk = 0;
  OSM_MsgHandler                    message_handler(client_handle);

  // Creamos un bloque para poner el mensaje y enviarlo luego.
  if (-1 == message_handler.recv_message (mblk))
  {
       return -1;
  }

  // Enviamos el mensaje.
  if (-1 == msg_sender_->put(mblk))
  {
       // Could not put the message.
    mblk->release ();
    return -1;
  }

       mblk->release ();
  return 0;

}
```

El patrón Reactor dentro del framework que proporciona ACE se utiliza extensamente en Osmius para envío y recepción de datos a través de sockets, capturar expiraciones de contadores del tiempo y para recoger llamadas desde señales como la pulsación de Control-C por un usuario.

Acceptor-Connector

Descripción

Es un patrón de diseño que separa el proceso de conexión e inicialización de servicios entre aplicaciones cooperantes en un entorno de red del proceso posterior al establecimiento de la dicha conexión e inicialización.

Contexto de uso

Este patrón implementa soluciones para dos tipos de situaciones: aquélla en la que eres el componente activo de la conexión el que la inicia, y la situación en la que eres la parte activa que está esperando a que se produzcan conexiones desde componentes activos.

En el caso de Osmius la principal aplicación de este patrón es en la parte pasiva o Patrón *Acceptor*. Como para el inicio de conexión y envío de datos es secillo programar un código portable con las *Wrapper Facade* ACE tiene menos aplicación el patrón activo *Connector*.

El Centro de Supervisión de Osmius abre un puerto de red mediante sockets, en el que espera que se conecten los agentes maestros para enviarle los mensajes recogidos de sus colas de mensajes desde los agentes de monitorización de instancias.

El Agente Maestro abre también un puerto en el que se queda escuchando paticiones de conexión y de envío de mensajes desde los Agentes de Monitorización. También y al igual que los agentes, esperan conexiones para el envío de órdenes.

Este patrón y el código que lo implementa - basado en ACE – se utiliza en varios lugares del motor de Osmius. Nos permite centramos en el tratamiento de los datos una vez se ha establecido la conexión y han sido inicializados los servicios.

Este patrón se basa en la clase ACE_Svc_Handler, de la que heredaremos nuestra clase para implementar el servicio que trata los datos recibidos.

Al crear el Acceptor en nuestro programa éste abre la conexión en modo espera y se registra con el *Reactor* para recepción de conexiones en el socket indicado.

Cuando llega una conexión el Reactor llama al método *handle_input()* en el que el *Acceptor* - que es un Factory o clase capaz de crear otros objetos - crea una nueva instancia de la clase que presta el servicio, le pasa la conexión, lo inicializa y lo registra con el reactor para eventos de llegada de datos para que sea él el que los trate.

Resumiendo Acceptor crea y conecta a un recurso una instancia del tipo que le indiquemos.

Acceptor está preparado para que el canal de conexión sea cualquier mecanismo IPC soportado por el SO y para utilizar nuevos tipos de canales como ficheros o incluso creados por nosotros. En el caso de Osmius se utilizan sockets remotos y locales.

Beneficios en el caso de Osmius:

- Reusabilidad de código de conexión e inicialización.
- Reusabilidad de estructura entre procesos.
- Capacidad para cambiar canales de comunicación con poco impacto.
- Portabilidad garantizada.
- Integración con *Reactor* para el tratamiento de eventos.

Ejemplo de uso

Veamos el caso del agente maestro que ha de recibir mensajes de sus agentes y enviarlos al servidor central.

La clase que inplementa nuestro servicio para el tratamiento de los mensajes de entrada:

```
class ACE_Svc_Export OSM_MA_MsgReceiver
  : public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
// Heredamos de la clase ACE_Svc_Handler y le indicamos que las conexiones
// van a ser en este caso por sockets sin hilos ni sincronización.
{
public:
  // Constructor. Le pasamos un puntero al servicio encargado de enviar los
  // mensajes al Centro de Supervisión.
  OSM_MA_MsgReceiver (OSM_MA_MsgSender *handler = 0)
    : msg_sender_ (handler) {}

  // Open() será llamado por el Acceptor después de crearme al llegar una conexión.
  virtual int open (void *);

  // Método gancho llamado antes de destruir la instancia.
  virtual int close (u_long = 0);

protected:
  // Llamado cuanodo agente tenga datos que enviar. Lo leeremos y se lo pasamos al
  // servicio de envío envuelto en un ACE_Message_Block.
  virtual int handle_input (ACE_HANDLE handle);

  // Llamado cuando un cliente termina. Quitamos al agente de "connected_clients_".
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);

  // Puntero al servicio MessageSender.
  OSM_MA_MsgSender *msg_sender_;

  // Control de descriptores de clientes o agentes conectados.
  ACE_Handle_Set connected_clients_;
};
```

La implementación:

```
int OSM_MA_MsgReceiver::open (void *)
{
  ACE_HANDLE handle = peer ().get_handle ();
  // Nos registramos para que nos llame el reactor cuando lleguen datos.
  // Llamará a nuestro método handle_input() cuando esto ocurra.
  if (reactor ()->register_handler
        (handle, this, ACE_Event_Handler::READ_MASK) == -1)
    return -1;
  connected_clients_.set_bit (handle);
  return 0;
}

*---------------------------------------------------------------------------*/
```

```
int OSM_MA_MsgReceiver::handle_input (ACE_HANDLE client_handle)
{
  ACE_Message_Block   *mblk = 0;
  OSM_MsgHandler      message_handler(client_handle);

  // Recibimos el mensaje.
  if (-1 == message_handler.recv_message (mblk))
  {
       return -1;
  }

  // Se lo pasamos al encargado de enviarlo
  if (-1 == msg_sender_->put(mblk))
  {
    mblk->release ();
    return -1;
  }
  mblk->release ();
  return 0;
}

/*---------------------------------------------------------------------------*/
int OSM_MA_MsgReceiver::handle_close (ACE_HANDLE handle, ACE_Reactor_Mask)
{
  connected_clients_.clr_bit (handle); // Un agente menos.
  return ACE_OS::closesocket (handle); // Cerramos el socket.
}
```

La definición y la implementación del Acceptor encargado de crear al anterior, inicializarlo y pasar la conexión:

```
class ACE_Svc_Export OSM_MA_MsgAcceptor
  : public ACE_Acceptor<OSM_MA_MsgReceiver, ACE_SOCK_ACCEPTOR>
// Creamos nuestro Acceptor OSM_MA_MsgAcceptor, diciéndole que el tipo de conexión es de
// tipo SOCKET y que debe crear instancias de servicio del tipo que hemos visto antes.
{
public:
  // Constructor.
  OSM_MA_MsgAcceptor (OSM_MA_MsgSender *h = 0)
    : msg_sender_(h), msg_receiver_(h) {}

protected:
  typedef ACE_Acceptor<OSM_MA_MsgReceiver, ACE_SOCK_ACCEPTOR>
          PARENT;

  // Este es el método factory para crear los objetos servicio. Es un método Gancho.
  virtual int make_svc_handler (OSM_MA_MsgReceiver *&sh)
  {
       sh = &msg_receiver_;
       return 0;
  }

  // Cierre.
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0)
  {
    PARENT::handle_close ();
    msg_receiver_.close ();
    return 0;
  }

  // Message Sender - to central server
  OSM_MA_MsgSender *msg_sender_;

  // Message receiver - from local agents.
  OSM_MA_MsgReceiver msg_receiver_;
};
```

No hay más. La estructura del Acceptor ya está preparada por defecto para registrarse con el Reactor también por defecto, para nuevas conexiones en el

socket y para crear objetos de nuestro tipo OSM_MA_MsgReceiver cuando sea el momento.

Para poner todo a funcionar basta con:

```
// Puerto de escucha.
u_short   cld_port = 1970;

// Wrapper Facade para direcciones INET, portable y reutilizable.
ACE_INET_Addr cld_addr (cld_port);

// Nuestro Acceptor.
OSM_MA_MsgAcceptor osmius_message_acceptor;

// Listo para funcionar y registrado con el Reactor.
// Cuando este empiece a detectar eventos se pondrá toda la maquinaria a funcionar.
osmius_message_acceptor.open(cld_port);
```

Trabajo Futuro e Investigación

Osmius implementa muchos de los paradigmas investigados actualmente en las ciencias de la computación, y siendo su código de libre acceso y modificación permite su uso intensivo en la investigación dentro de la Inteligencia Artificial como en otros campos relativos a la Tecnología y la Información.

El campo de los aplicaciones de supervisión de sistemas en red tiene un interés manifiesto dentro del creciente marco actual de proveedor de servicio – cliente y ANS. La pronta identificación de problemas raíces cobra una importancia capital y, la reusabilidad para la consecución de los requisitos funcionales, de calidad de servicio y de tiempos y costes del desarrollo se convierten en un objetivo fundamental.

Osmius proporciona un entorno que además de encajar en objetivos de mercado y tendencias en el área de control y supervisión de servicios, permite a investigadores de las ciencias de la computación el aplicar de forma práctica el resultado de sus investigaciones.

En este sentido se pretende crear un entorno extremadamente reusable y aplicar conceptos MDA y la creación de un lenguaje de patrones específico para la generación de nuevos agentes y módulos, desde modelos definidos en metalenguajes independientes de la implementación.

Desde unos requisitos funcionales de un nuevo agente para supervisar un nuevo tipo de elemento (supongamos un electrodoméstico), junto con las restricciones de tiempo útil y de rendimiento, podremos generar el código completo y compilarlo gracias a la base portable en la que se sustentan Osmius y ACE para la platafoma o plataformas destino sin pérdida de tiempo en sincronización de compatibilidades.

Otro campo que se abre para Osmius es el de la Inteligencia Artificial aplicada a resolver los problemas de correlación y simplificación de eventos para mejorar el tiempo de toma de decisiones y acciones de correción ante fallos o determinados eventos.

Parece además éste un campo que promete resultados para la Inteligencia Artificial, ya que el problema tantas veces mencionado de la falta de datos, en este caso, no es tal. Es relativamente fácil hacerse con una buena base de datos de eventos de un sistema de monitorización ya implantado y aplicar algoritmos de aprendizaje o técnicas de minería de datos.

Osmius manejará en el Centro de Supervisión un histórico de eventos en el que se encuentra el punto de información en el que se basará toda la problemática de análisis y minería de datos con el objetivo de identificar patrones de ocurrencia y generación de nuevas reglas para los diferentes dominios.

Es la minería de datos aplicada a la generación de reglas por dominio, junto con el uso de patrones de diseño de la arquitectura software, la que dotan al modelo de una gran adaptabilidad y flexibilidad para alcanzar el objetivo de mejorar la identificación de problemas y la respuesta para solucionarlos.

Según vaya madurando la implantación de sistemas mixtos y con resultados patentes como se propone en este artículo, se abordarán nuevos retos algunos de los cuales se trata de avanzar es este apartado.

En este área cobrarán mayor protagonismo en interés los **Algoritmos Predictivos**. La idea es aplicar este tipo de algoritmos para detectar problemas en un servicio, y actuar corrigiéndolos antes incluso de que afecten a los contratos firmados con los clientes o usuarios del los sistemas.

Estos algoritmos deberán ser capaces de reponder de manera proactiva no sólo ante problemas si no a previsiones de capacidad de rendimiento, o de cualquier tipo de recurso.

Preguntas como:

- ¿Cuál será el uso de CPU o disco de este servidor durante el próximo mes?
- ¿Y durante el próximo minuto?
- ¿Puedo predecir si este router va a fallar en la próxima media hora?

Y en general del tipo:

- ¿Cuál será la evolución de la variable X en el recurso Y?

, serán objetivo muy interesante de los sistemas de supervisión en los que tendrá aplicación la investigación sobre aprendizaje automático y minería de datos en sistemas orientados a evento.

Los nuevos sistemas y algoritmos podrán utilizar los datos de cada uno de los tipos de evento (variables) recibidos de las diferentes instancias para realizar las predicciones, y utilizar las posibles correlaciones con otras variables para variar las estimaciones.

Deberemos utilizar algoritmos de Análisis de Series Temporales o Data-Mining, dependiendo de si queremos predicciones a corto o largo plazo.

No parece descabellado que según se vayan programando y usando algoritmos para todo el entorno que supone Osmius, aparezcan **patrones de diseño** específicos para aplicaciones de Supervisión o en el aprendizaje automático orientado a eventos. También será apasionante la identificación de **patrones de**

metodología de trabajo y su posterior documentación que surgirán de la experiencia del desarrollo del software y sus posibles módulos y del análisis de las bases de datos de eventos.

Conclusiones

Hemos visto que los sistemas que ayudan a supervisar y monitorizar otros sistemas en red toman una gran importancia para ayudar a estos últimos a cumplir con una calidad de servicio cada vez más exigente y que se plasma en contratos con los usuarios y/o clientes a través de los Acuerdos de Nivel de Servicio.

Mejorar la productividad de los sistemas y de los grupos de operación encargados de supervisarlos es importante, y que las aplicaciones encargadas sean robustas y fácilmente adaptables cobra en este entorno toda su importancia.

Las herramientas comerciales para la supervisión y monitorización de elementos o instancias en red presentan ciertos problemas como son:

- La **elevada complejidad** de configuración, puesta en marcha y mantenimiento.
- Suelen ser **dependientes de plataforma**. Los motores están desarrollados para sólo ser usados con el formato propietario de la herramienta de supervisión. Además, los usuarios sólo disponen de los binarios compilados para un número reducido de plataformas.
- **Caras**.

Una buena herramienta de Código Abierto capaz de generar nuevos agentes de forma muy rápida y robusta y capaz de funcionar en entornos productivos exigentes,cubre un hueco evidente en el mercado actual.

Se han presentado y explican en el artículo los mecanismos de reutilización más comunes de forma genérica, para pasar a ejemplos concretos de plataforma (ACE) y aplicación (Osmius).

En el contexto de la aplicación Osmius se han expuesto lo Patrones de Diseño principales con el problema concreto a resolver junto con ejemplos de código comentados para ver su aplicación y su potencial de la forma más clara posible.

En el código conjunto de Osmius y su uso de ACE se utilizan muchos patrones la mayor parte de las veces de forma implícita porque así lo hacen las clases y estructuras de ACE.

El aprendizaje de un Middleware y los frameworks que ofrece es un proceso con una curva de aprendizaje larga y con cierta pendiente, pero los beneficios obtenidos una vez alcanzado el nivel de comprensión que permite el uso ágil de la plataforma, en el caso de Osmius, ha mejorado con creces la productividad y la reusabilidad final del proyecto. En cada caso habrá de analizarse según los tiempos y requisitos del proyecto si es interesante realizar este tipo de esfuerzo.

En general y si los objetivos de desarrollo son a medio o largo plazo parece que sí merece la pena.

No cabe duda de que el interés en sistemas capaces de supervisar aplicaciones o procesos de negocio de manera que se mejore el servicio prestado seguirá en auge en los próximos años y Osmius trata de aunar este interés con la investigación para ofrecer un entorno práctico para teorías provinientes de la investigación.

Referencias

The ADAPTIVE Communication Environment An Object-Oriented Network Programming Toolkit for Developing Communication Software.

Douglas C. Schmidt - 1993
11th and 12th Sun user group conferences. California

An architectural overview of the ACE framework.A Case Study of Successful Cross-Platform Systems Software Reuse

Douglas C. Schmidt - 1999

USENIX

C++ Network Programming: Mastering Complexity Using ACE and Patterns
Libro escrito por: Douglas C. Schmidt

Stephen D. Huston

SBN 0-201-60464-7 (2002)

C++ Network Programming: Systematic Reuse with ACE and Frameworks
Libro escrito por: Douglas C. Schmidt

Stephen D. Huston

ISBN 0-201-79525-6 (2003)

The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming

Libro escrito por: Stephen D. Huston

James CE Johnson

Umar Syyid

ISBN 0-201-69971-0 (2003)