UNIVERSIDAD INTERNACIONAL DE LA RIOJA unir

Universidad Internacional de La Rioja
Máster universitario en Ingeniería de Software y Sistemas Informáticos

Desarrollo y análisis del sistema distribuido Melissa basado en Klepsydra y ZeroMQ

Trabajo Fin de Máster

Tipo de trabajo: Desarrollo práctico

Presentado por: Aldazabal Rego, Javier

Director/a: Sicilia Montalvo, Juan Antonio

Resumen

El objetivo de este trabajo ha sido implementar un sistema distribuido con nodos que envíen por red imágenes y coordenadas de navegación. Para conseguir un diseño adecuado en sistemas de estas características, inicialmente se han introducido soluciones a problemas de concurrencia, y posteriormente analizado los frameworks de ZeroMQ, Klepsydra y algunas alternativas. El propósito de esta investigación ha sido poder disponer de los componentes adecuados para crear un sistema embebido autónomo con patrón *publisher-subscriber*.

La finalidad del desarrollo ha consistido en evaluar las ejecuciones del sistema en distintos escenarios para obtener la configuración que mejor rendimiento ofrezca. A este respecto podemos concluir que, aunque una mayor frecuencia de mensajes (esencial para sistemas de respuesta ágil) supone un ligero incremento de procesamiento, ningún evento se pierde y la memoria que consume el sistema permanece constante.

Palabras Clave: Concurrencia, sistema distribuido, disruptor, *event-loop*, sistema autónomo embebido

Abstract

The objective of this work has been to implement a distributed system with nodes that send images and navigation coordinates over the network. To achieve a proper design in this type of systems, initially solutions to concurrency problems have been introduced, and subsequently analyzed the ZeroMQ, Klepsydra and some alternatives frameworks. The purpose of this research has been to have the appropriate components to create an autonomous embedded system with publisher-subscriber pattern.

The purpose of the development has been to evaluate the system executions in different scenarios to obtain the configuration that offers the best performance. In this regard we can conclude that, although a higher frequency of messages (essential for agile response systems) means a slight increase in processing, no event is lost and the memory consumed by the system remains constant.

Keywords: concurrency, distributed system, disruptor, *event-loop,* autonomous embedded system.

Índice de contenidos

1. Introducción 9
1.1 Justificación 9
1.2 Planteamiento del trabajo 12
1.3 Estructura de la memoria 13
2. Contexto y estado del arte 14
2.1. Concurrencia y técnicas lock-free 14
2.2. Klepsydra 21
2.3. ZeroMQ 25
2.4. Herramientas y tecnologías utilizadas 29
2.5. Conclusiones 35
3. Objetivos concretos y metodología de trabajo 36
3.1. Objetivo general 36
3.2. Objetivos específicos 36
3.3. Metodología del trabajo 37
4. Desarrollo específico de la contribución 39
4.1. Organización y transcurso del desarrollo 39
4.2. Descripción del sistema 43
4.1.1. Requisitos del sistema 43
4.1.2. Arquitectura y desarrollo 46
4.3. Evaluación de escenarios de ejecución 53
4.3.1. Escenario E == P 54
4.3.2. Escenario E == 1 58
4.3.3. Escenario 1 < E < P 58
5. Conclusiones y trabajo futuro 59
5.1. Conclusiones 59
5.2. Líneas de trabajo futuro 60

6. Bibliografía 61

Anexos..64

Anexo I. Artículo 64

Introduction 64

State of the art..65

Objetives..67

Software Development..68

Conclusions 69

Bibliography 70

Anexo II. Glosario 71

Índice de tablas

Tabla 1. Aplicaciones potenciales del sistema desarrollado (elaboración propia) 10

Tabla 2. Problemas derivados de locks 16

Tabla 3. Componentes básicos (elaboración propia) 22

Tabla 4. Tipos de datos, serializadores y utilidades (elaboración propia) 23

Tabla 5. Tipos de providers (elaboración propia)..23

Tabla 6. Configuración de escenarios (elaboración propia) 38

Tabla 7. Instrucciones para instalar ZeroMQ (elaboración propia).................................43

Tabla 8. Instrucciones para instalar Klepsydra core y robotics (elaboración propia) 44

Tabla 9. Comportamiento de los servicios RoboBee y QueenBee (elaboración propia) 47

Tabla 10. Indicadores de evaluación de rendimiento (elaboración propia) 53

Tabla 11. Frecuencia suscripción 20hz (elaboración propia) 57

Tabla 12. Frecuencia suscripción 10hz (elaboración propia) 57

Tabla 13. Frecuencia suscripción 5hz (elaboración propia) 57

Índice de figuras

Ilustración 1. Sector vehículos aéreos no tripulados (Goldman Sachs, 2016)....................10
Ilustración 2. UAV robótico (Wyss Institute, 2019)....................11
Ilustración 3. Dining philosophers (Dijkstra, 1971)....................14
Ilustración 4. Algoritmos lock-free y wait-free (Rinaldi, 2019)....................15
Ilustración 5. Programación lock-free (Sutter, 2014)....................16
Ilustración 6. Diagrama de funcionamiento del event-loop (Catarino, 2019)....................17
Ilustración 7. Diagrama event-loop (Ghiglino, 2019)....................18
Ilustración 8. Diagrama de la estructura de datos ring buffer (Fowler, 2011)....................19
Ilustración 9. Esquema de Melissa con disruptors (elaboración propia)....................20
Ilustración 10. Diagrama de interacción de componentes (klepsydra, 2019)....................21
Ilustración 11. Diagrama Klepsydra admin (bitbucket, 2019)....................24
Ilustración 12. Parámetros de creación de ZeroMQ (Hintjens, 2013)....................25
Ilustración 13. Patrón publisher subscriber en ZeroMQ (Hintjens, 2013)....................26
Ilustración 14. Comparativa de soluciones de encolado de mensajes (Hadlow, 2011)....................27
Ilustración 15. Librería de red NNG (Sustrik,2019)....................28
Ilustración 16. Arquitectura MQTT (HiveMQ, 2019)....................28
Ilustración 17. Windows 10 (Microsoft, 2015)....................29
Ilustración 18. Virtualización VMWare Workstation Player (VMWare, 1999)....................29
Ilustración 19. Ubuntu Desktop 18.04 (Ubuntu, 2018)....................30
Ilustración 20. Visual Studio Code (VS Code, 2015)....................30
Ilustración 21. Diagramas de clases con GenMyModel (GenMyModel, 2013)....................31
Ilustración 22. Creación de esquemas con Draw.io (Draw.io, 2005)....................31
Ilustración 23. GNU C++ Compiler (GNU, 1987)....................32
Ilustración 24. Cmake (CMake, 2000)....................32
Ilustración 25. Control de versiones Git (Torvalds, 2005)....................33
Ilustración 26. Plataforma de desarrollo colaborativo Github (Microsoft, 2007)....................33

Ilustración 27. Librería de mensajería ZeroMQ (iMatix, 2013) 34
Ilustración 28. Klepsydra Technologies (Ghiglino, 2019) 34
Ilustración 29. Entrega incremental (Sommerville)....................................37
Ilustración 30. Procedimiento de pull-request (elaboración propia)....................................39
Ilustración 31. Procedimiento de rebase (elaboración propia) 40
Ilustración 32. Análisis de core dumps (elaboración propia) 40
Ilustración 33. Stack backtrace en rbApplication (elaboración propia) 41
Ilustración 34. Patrón composition root (Van Deursen, 2018) 42
Ilustración 35. LLVM project (LLVM dev group, 2003) 42
Ilustración 36. Ejemplo importación de librería en CMake (elaboración propia)....................................44
Ilustración 37. Framework Open MCT (NASA, 2015) 45
Ilustración 38. Melissa Github project (Writ in wáter, 2014) 46
Ilustración 39. Prototipo diagrama de clases (elaboración propia)....................................46
Ilustración 40. Especificación de RoboBee (elaboración propia) 47
Ilustración 41. Constructor RoboBee (elaboración propia)....................................48
Ilustración 42. Método ejecutado cíclicamente (elaboración propia)....................................48
Ilustración 43. Especificación de QueeenBee (elaboración propia) 49
Ilustración 44. Procesamiento imagen (elaboración propia) 49
Ilustración 45. Definición de waypoint (elaboración propia) 50
Ilustración 46. Clase resultante con constructor y métodos básicos (elaboración propia).....50
Ilustración 47. Archivo de configuración YAML (elaboración propia) 51
Ilustración 48. Constructor de ZMQ_middleware_facility (elaboración propia)....................................52
Ilustración 49. Diagrama de comunicación (elaboración propia)....................................52
Ilustración 50. Lanzamiento de aplicaciones en Melissa (elaboración propia) 53
Ilustración 51. Igual número de productores y consumidores (elaboración propia)....................................54
Ilustración 52. Número de eventos descartados (elaboración propia) 55
Ilustración 53. Frecuencia suscripción imágenes (elaboración propia) 55
Ilustración 54. Frecuencia suscripción waypoints (elaboración propia)....................................56

Ilustración 55. Estabilidad en CPU y RAM, respectivamente (elaboración propia) 56

Ilustración 56. Productores a un consumidor (elaboración propia) 58

Ilustración 57. Escenario de asociación mixto (elaboración propia)......................................58

1. Introducción

1.1 Justificación

A la hora de programar utilizando técnicas de ***multithread*** (donde hay más de una unidad de ejecución en el mismo proceso compartiendo los recursos disponibles), los desarrolladores se encuentran con algunos desafíos propios de este modelo de programación.

Los problemas derivados de este modo de programación son más difíciles de detectar, replicar, depurar y solucionar que con la manera convencional, y factores como la falta de un buen diseño pueden provocar caídas de rendimiento. Como introduciremos más adelante, existen soluciones para resolver estos problemas de **sincronización de procesos** tales como introducir bloqueos o emplear técnicas libres de ellos. Sin embargo, para obtener mayor rendimiento es complicado considerar todos los factores para cada escenario.

Debido a las causas expuestas anteriormente, la pérdida de eficiencia es una cuestión fundamental, especialmente en sistemas que requieren gran desempeño como el escenario que se propone en este trabajo. Concretamente, se presentará el desarrollo de un sistema donde una **serie de nodos** (conceptualmente pequeños drones) se comunicarán entre ellos a través de capa de red mediante envío de mensajes con un patrón de publicación y suscripción con relación a **navegación autónoma** y **captura** y **procesamiento de imágenes**.

Consideramos por tanto la necesidad de implementar las aplicaciones propuestas para así evaluar los modelos de comunicación y gestión de mensajes entre ambos tipos de nodos y reflejar cual ofrece **mayor rendimiento** en distintas situaciones, empleando para ello algunas de las más recientes tecnologías disponibles.

La actual solución podría usarse en situaciones reales como los ejemplos mostrados en la tabla 1, tratando de mejorar las prestaciones de soluciones similares, concretamente en transferencia y procesamiento de datos sin pérdida.

Tabla 1. Aplicaciones potenciales del sistema desarrollado (elaboración propia)

Inspección de infraestructuras industriales e identificación de daños
Identificación de recursos naturales y mapeado de terrenos en sectores como minería o agricultura
Tareas de rescate en entornos complejos y condiciones adversas
Estudiar y replicar modelos de comportamiento en especies con inteligencia colectiva
Operaciones de vigilancia y seguridad civil
Sistema aéreo de transporte de paquetes
Monitorización de condiciones climáticas

Los sistemas autónomos embebidos están sufriendo un gran crecimiento de mercado y un ejemplo de ello son las estimaciones planteadas por algunos analistas del sector. Gartner (2017) publica como en 2016 y 2017 los ingresos de venta, únicamente de drones comerciales eran de 6.000 millones de dólares, y la estimación entre 2016 y 2020 era que el mercado moverá 100.000 millones de dólares. (Goldman Sachs, 2016).

Ilustración 1. Sector vehículos aéreos no tripulados (Goldman Sachs, 2016)

En este ámbito y otros como el de la robótica inteligente, el hardware ha mejorado de manera sustancial, proporcionando computadores de a bordo de buenas prestaciones a la vez que han mejorado los dispositivos como sensores ópticos, *LIDAR*, infrarrojos, termográficos, que ofrecen más datos, más rápido y de manera más fiable. Algunas instituciones como la universidad de Harvard, además de diversas empresas innovadoras, ya están consiguiendo progresos funcionales en dispositivos como el insecto robótico que vemos en la ilustración 2..

> "Únicamente debido a los recientes avances de este laboratorio en fabricación, materiales y diseño, se ha podido probar esto. Y simplemente funcionó, espectacularmente bien." (Wyss Institute, 2019).

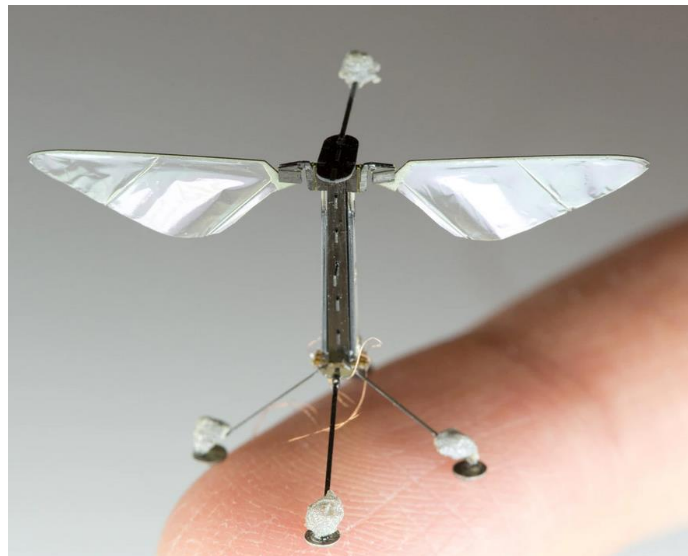


Ilustración 2. UAV robótico (Wyss Institute, 2019)

Esta situación conlleva también unos desafíos importantes, aumentando la complejidad de los algoritmos (filtrado de datos o navegación por visión) unido a tener que procesar grandes cantidades de datos, integración con el resto de los sistemas, etc. Ghiglino (2019) enumera algunos de estos **retos software** como la falta de técnicas modernas o de ingeniería de software de alta calidad, derivando en consecuencias como:

- Aparición de errores
- Funcionalidad limitada
- Retrasos en la entrega de datos
- Infrautilización de recursos

1.2 Planteamiento del trabajo

Este trabajo tratará de servir como guía en la aplicación del software orientado a técnicas de *lock-free* en su implementación de ***event-loop*** para arquitecturas basadas en eventos con patrón ***publisher/subscriber***.

Para este objetivo se propone crear un sistema con aplicaciones y servicios que implementen estos patrones, de cara a conocer una óptima asignación entre **productores** y **consumidores de eventos**.

Por tanto, intentaremos ofrecer unas conclusiones sobre cual es el modelo que ofrece mejor rendimiento, basándonos en los resultados obtenidos tras la ejecución parametrizada de las aplicaciones desarrolladas para el sistema propuesto.

Para poder hacerlo contaremos con una librería de mensajería asíncrona de alto rendimiento orientada a aplicaciones concurrentes distribuidas. También emplearemos un *framework* para implementar la funcionalidad que permitirá tener distintas configuraciones de productores y consumidores.

Para medir el **rendimiento del sistema**, primero definiremos ciertos criterios de evaluación para que arrojen unos resultados válidos y podamos extraer las conclusiones correspondientes. Utilizaremos una herramienta para analizar en tiempo real el uso de recursos, pérdida de datos y eficiencia en ejecución, entre otros.

1.3 Estructura de la memoria

En el **contexto y estado del arte** se comienza describiendo algunas de las técnicas utilizadas como solución a los desafíos que aparecen en la programación *mulithread*, mencionando también ventajas y desventajas. Se profundiza en el patrón *disruptor*, y el *event-loop* para gestión de los eventos en una arquitectura orientada a ellos. Se explica también una librería para programar comunicación de red y un framework adecuado para sistemas autónomos embebidos.

En los **objetivos generales** se describe el própósito del trabajo de manera sintetizada, que es servir de guía para futuros desarrolladores y poder aplicar el sistema en escenarios reales. Por otra parte, en los **específicos** se listan las distintas etapas que lo componen.

En la **metodologia de trabajo** se describen los procedimientos seguidos para iniciar, desarrollar y completar el sistema, así como detallar las herramientas empleadas.

En cuanto al **desarrollo específico**, se compondrá de lo siguiente:

- Incluir diagramas de clases y arquitectura
- Implementación y detalle de los módulos y tests
- Explicación del entorno de pruebas
- Definición de los indicadores evaluadores
- Obtención y presentación de los resultados
- Análisis y extracción de conclusiones

Por último, en **conclusiones** y líneas futuras se sintetizarán las conclusiones globales y se expondrán las líneas de trabajo donde se puede profundizar teniendo en cuenta la contribución de este trabajo.

2. Contexto y estado del arte

2.1. Concurrencia y técnicas lock-free

El rendimiento de un computador inicialmente era concebido, por la rapidez con la que un proceso de hilo único era capaz de ejecutar el programa (Fuller y Millet, 2011). Históricamente, durante el pasado siglo los esfuerzos de los principales fabricantes fueron en la línea de aumentar la frecuencia de reloj de la CPU y el rendimiento de instrucciones.

Sin embargo, debido a los límites impuestos por factores como la temperatura, la industria se centró en evolucionar hacia el ***multithread*** y las arquitecturas *multicore*. (Sutter, 2009). Disponiendo de estas arquitecturas surgió el modelo de programación que permitía lanzar varios hilos en el contexto de un mismo proceso. Estos hilos se ejecutaban de forma independiente, pero compartiendo recursos permitiendo concurrencia en la ejecución (*multithreading*).

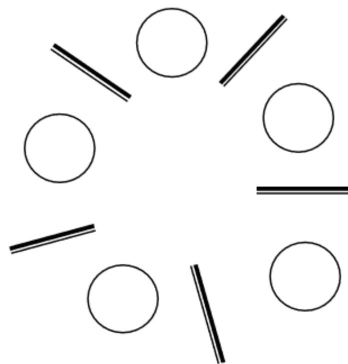


Ilustración 3. Dining philosophers (Dijkstra, 1971)

Un ejemplo clásico que muestra problemas de control de concurrencia es el de los filósofos gastronómicos expuesto por el conocido científico de computación Edsger W. Dijkstra en la ilustración 3.

Debido a esta concurrencia aparecen problemas como la sincronización de procesos con memoria compartida. Dentro de las soluciones para este problema se encuentra la técnica ***mutex*** (exclusión mutua) para ejecutar secciones críticas de código, que permite el acceso ordenado a recursos compartidos para impedir inconsistencias y errores.

El aseguramiento de la sección crítica implica, por las desventajas que conllevan estas soluciones bloqueantes, considerar por software técnicas de sincronización no bloqueantes, habitualmente de menor coste:

- ***Lock-free***: Si varios procesos desean entrar a la sección crítica al menos uno de ellos debe poder hacerlo. En caso de que uno falle o se suspenda, no podrá causar lo mismo a otro *thread* y detener el avance del sistema. (Göetz, 2006).
- ***Wait-free*** (espera limitada): Cualquier proceso debe poder entrar a la sección crítica en un tiempo finito. Esta condición es deseable pero no siempre se puede asegurar, y además con esta técnica se garantiza también el avance de cada uno de los *threads*.

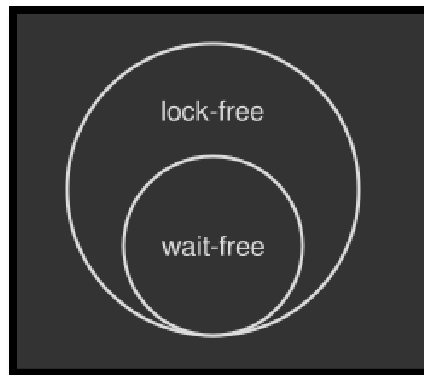


Ilustración 4. Algoritmos lock-free y wait-free (Rinaldi, 2019)

Como se muestra en la ilustración 4, con el uso de los algoritmos *wait-free* los programas se completan en un número finito de pasos, y son un subconjunto de los de *lock-free*.

Algunas ventajas e inconvenientes de *mutex* (por parte de algoritmos bloqueantes) y *lock-free* (por los no bloqueantes) son:

- ✓ En el uso de *mutex* la programación es más sencilla. Sin embargo, en los algoritmos *lock-free* la implementación es compleja al tener que manejar recursos como barreras de memoria u operaciones CAS (*Compare-And-Swap*).
- ✓ En cuanto al determinismo el modelo mutex es inferior a *lock-free*, que ofrece más previsibilidad. Esto se debe a la incertidumbre que ocurre cuando un *thread* se duerme y no se sabe cuándo va a despertar.
- ✓ Además, la exclusión mutua obliga a realizar cambios de contexto en situaciones de bloqueo, lo que supone un coste en materia de recursos.

La ilustración 5 es un reflejo de la comparación de escenarios *single/multi-thread* y los mecanismos típicos para gestión de memoria compartida y sincronización de procesos

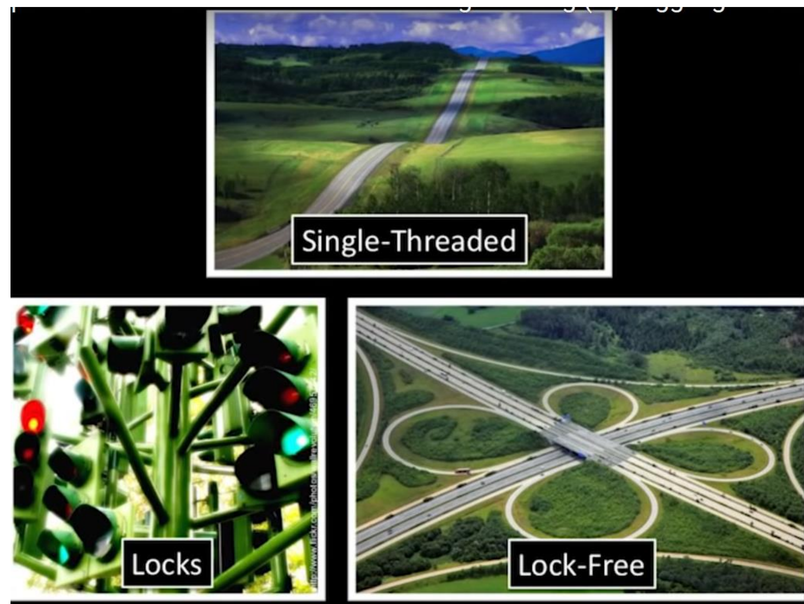


Ilustración 5. Programación lock-free (Sutter, 2014)

En la tabla 2 podemos observar problemas derivados del uso de *locks* a la hora de programar aplicaciones.

Tabla 2. Problemas derivados de locks

	Descripción
Deadlock	Dos hilos se evitan el acceso entre sí a la hora de acceder a un recurso
Livelock	Similar al *deadlock,* pero los procesos cambian su estado
Priority Inversion	Situación que se da cuando una tarea de alta prioridad es precedida por una de baja prioridad
Convoying	Múltiples hilos compiten para acceder al recurso y, aunque progresan, si uno falla pierde su turno y se producen cambios de contexto, que añaden *overhead* al rendimiento.

Extraído de Oneill, 2018

Para evitar bloqueos de aplicación, de *thread* o inversiones de prioridad, es adecuado el uso de técnicas *lock-free*, que posibilitan el intercambio de datos entre múltiples *threads* sin necesidad de usar bloqueos. (Rinaldi, 2019). Algunas se apoyan en la atomicidad de las operaciones susceptibles de generar conflicto, lo cual implica usar micro operaciones no segmentadas y evitar condiciones de carrera y dependencias al utilizar estructuras de datos compartidas.

Namiot (2016) asegura que el mayor desafío para la programación concurrente es la coordinación y **organización del flujo de datos**, más que adentrarse en estructuras de datos de relativamente bajo nivel. A este respecto de control de flujo de datos, un patrón de diseño destacable es el *event-loop.*

Event loop

El concepto de ***event-loop*** se entiende como un *thread* que espera y distribuye mensajes entrantes a una función de control de eventos, y suele utilizar una cola de mensajes para contener los eventos que llegan. Cada mensaje se decodifica y se realiza la acción correspondiente. Se utiliza habitualmente como una herramienta para controlar comunicaciones entre procesos. (Lafreniere, 2017).

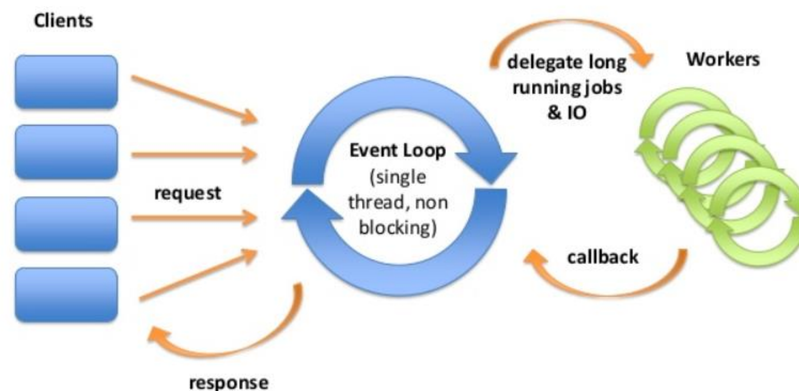


Ilustración 6. Diagrama de funcionamiento del event-loop (Catarino, 2019)

Con la llegada al estándar C++11 del soporte para subprocesos, ya no es necesario invocar funciones específicas del sistema operativo, proporcionando así la portabilidad del código.

El funcionamiento del *event-loop* se basa en varios proveedores de datos y un consumidor al que se le proporcionan datos de forma secuencial tratando de evitar pérdida de mensajes.

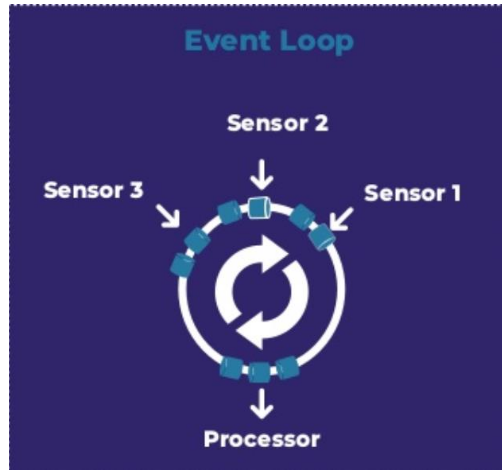


Ilustración 7. Diagrama event-loop (Ghiglino, 2019)

LMAX

Teniendo en cuenta este modelo, en 2011 una plataforma llamada LMAX *exchange* creó, para *trading* financiero minorista, un procesador de lógica de negocios (BLP) que buscaba manejar varios millones de órdenes por segundo con una baja latencia.

Debido a la necesidad de alto rendimiento, el sistema se alimenta de una fuente de eventos, y su BLP está rodeado de un tipo especial de colas (llamadas *disruptors*) para esas tareas de entrada y salida de eventos. Además, se ejecuta completamente en memoria que supone no tener base de datos (lento acceso *i/o*) y también más sencillez al no tener que mapear sus objetos y relaciones. Sobre el problema de la persistencia de datos en caso de fallo, se soluciona restableciendo el estado a partir de *snapshots* y se instancian varios BLPs para dar redundancia y alta disponibilidad.

Para ser procesables, antes y después de aplicar la lógica específica de *trading*, se deben descomponer y componer esos eventos (con funciones ejecutables en cualquier orden), escenario adecuado para **concurrencia** y motivo por el que crearon el *disruptor*.

Disruptor

Fue diseñado por *LMAX Exchange* para proporcionar una cola de trabajo de alto rendimiento y baja latencia en arquitecturas de procesamiento de eventos asíncronos. Este patrón asegura que cualquier dato sea propiedad de un solo hilo para acceso de escritura, reduciendo así la contención en comparación con otras estructuras de datos. (Fowler, 2011)

A primera vista, se puede pensar en un ***disruptor*** como un gráfico de multidifusión de colas donde los productores colocan objetos que se envían a todos los consumidores para consumo paralelo a través de colas separadas. Al observar el interior, se ve que esta red de colas es realmente una estructura de datos única: un búfer en anillo, como muestra la ilustración 8.

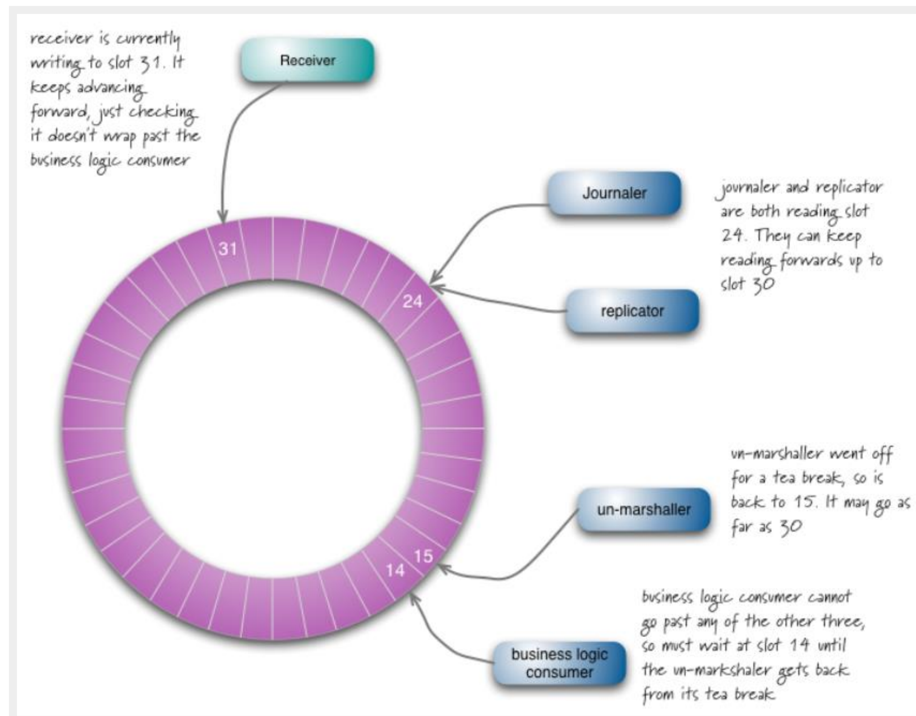


Ilustración 8. Diagrama de la estructura de datos ring buffer (Fowler, 2011)

Cada productor y consumidor tiene un contador de secuencia para indicar en qué ranura del búfer está trabajando actualmente y escribe el suyo propio, pero puede leer los contadores de los demás. De esta forma, el productor puede leer los contadores de los consumidores para asegurarse de que el espacio en el que desea escribir esté disponible sin ningún bloqueo en esos contadores. Del mismo modo, un consumidor puede asegurarse, revisando los contadores, de que solo procesa mensajes una vez que otro consumidor termina con él.

Un enfoque más convencional usaría una cola de productor y una de consumidor, cada una usando bloqueos como mecanismos para manejar la concurrencia. En la práctica, ocurre que las colas están completamente vacías o llenas la mayor parte del tiempo, lo que provoca contención de bloqueo y ciclos desperdiciados. (Thompson, 2011).

El *disruptor* mitiga esto al hacer que todos los productores y consumidores usen la misma estructura de colas, coordinándose entre sí observando los contadores de secuencia en lugar de usar mecanismos de bloqueo.

En resumen, el patrón *disruptor* se concibe como un búfer en anillo lleno de objetos de transferencia preasignados que utiliza barreras de memoria (la manera *lock-free* más rápida para comunicarse) para sincronizar productores y consumidores a través de secuencias. Dicho de otra manera, se trata de un componente de simultaneidad que implementa una red de colas que funcionan sin necesidad de bloqueos (Fowler, 2011).

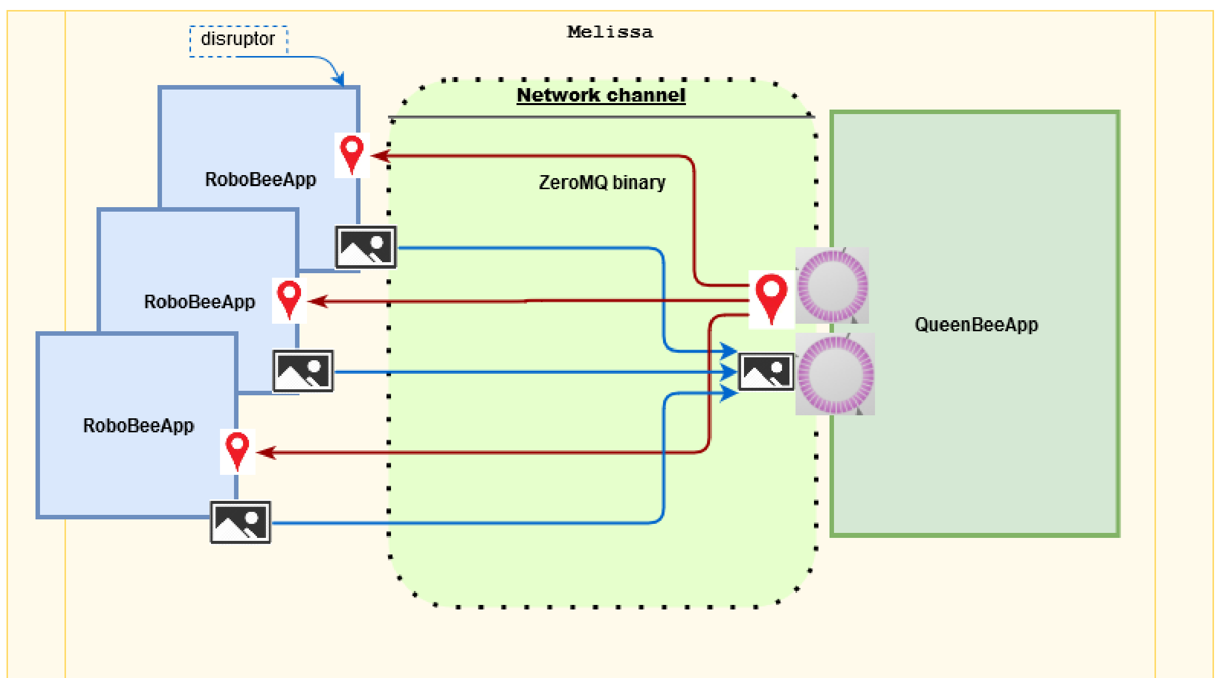


Ilustración 9. Esquema de Melissa con disruptors (elaboración propia)

La importancia del *ringBuffer* en este trabajo reside en que es el mecanismo usado por el *EventLoopMiddlewarerProvider* de Klepsydra, empleado en nuestras aplicaciones tal como observamos en la ilustración 9.

2.2. Klepsydra

Vamos a introducir un reciente software llamado **Klepsydra**, cuyo código ha sido publicado en 2019 y que se apoya en los conceptos de *event-loop* y del *ring-buffer* del *disruptor* descritos anteriormente.

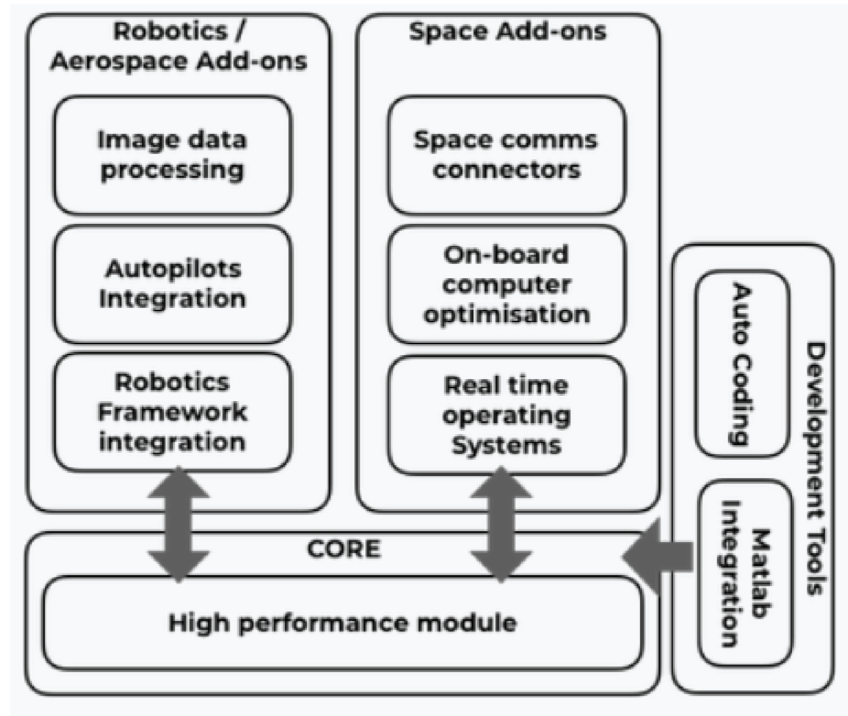


Ilustración 10. Diagrama de interacción de componentes (klepsydra, 2019)

Klepsydra se describe como ingeniería de software de procesamiento de datos de alto rendimiento para sistemas embebidos, aunque sus fundamentos provienen del sector de banca de inversión. Sus principales características son:

- Dispone de componentes para robótica, entornos aeroespacial y espacial, ofreciendo un ritmo rápido de desarrollo.
- Es una plataforma independiente y ligera.
- Está integrado con Matlab y ROS (Robot Operating System)
- Ofrece aumento de rendimiento en procesamiento de datos, especialmente en estabilidad y escalabilidad.
- Aporta facilidades para la resolución de problemas complejos.

Este software va dirigido a sistemas autónomos embebidos cuyos requisitos son, entre otros, uso de algoritmia compleja o procesamiento de gran cantidad de datos con recursos computacionales limitados. Algunos ejemplos de estos sistemas se encuentran en robótica inteligente de enjambre o en vehículos y aeronaves autónomas ya que disponen de un gran número de sensores.

En cuanto a la funcionalidad básica, la composición del software Klepsydra se basa en los conceptos de lo siguientes patrones de diseño:

- Inyección de dependencias
- Patrón *strategy*
- Patrón *composition root*
- Patrón *observer*

Se divide en un API (interfaz de programación de aplicaciones) de **composición** (usado para integrar y conectar las instancias) y en uno de **aplicación** (empleado para conseguir la funcionalidad)

Tabla 3. Componentes básicos (elaboración propia)

Módulo	Descripción
kpsr::Environment	Componente cuyo objetivo es aislar el entorno de ROS (Robot Operating System) de las clases de aplicación para que permanezcan agnósticas al middleware.
Kpsr::YamlEnvironment	Permite cargar ficheros de configuración en lenguaje YAML.
kpsr::Service	Módulo para aplicaciones que necesitan ejecutar tareas regulares, con configuración y administración remota.
kpsr::Publisher	Las aplicaciones que usen esta clase emplean punteros para publicar los eventos, ya sea a otra clase o al middleware.
kpsr::Subscriber	Realiza un seguimiento de los listeners que se registran para eventos que provienen de un middleware

En la tabla 3 se muestran algunas clases básicas que hemos utilizado en el software Melissa como componentes fundamentales.

Tabla 4. Tipos de datos, serializadores y utilidades (elaboración propia)

Módulo	Descripción
Kpsr::vision_ocv::ImageEventData	Un *wrapper* de imagen de OpenCV, que añade número de secuencia y *frameID* a la matriz (cv::Mat)
kpsr::vision_ocv::ImageDataFactory	Utilidad que gestiona el *allocation* de imágenes, la copia o el redimensionamiento
kpsr::vision_ocv::ImageDataZMQMapper	Serializador para transformar los miembros para su envío
kpsr_codegen::generator	Generador automático de código que cosntruye clases a partir de esquemas en lenguaje YAML

En la tabla 4 se muestra un ejemplo de tipo de datos utilizado en Klepsydra, así como alguna utilidad para gestión de imágenes o serialización.

Tabla 5. Tipos de providers (elaboración propia)

Módulo	Descripción
kpsr::EventEmitterMiddlewareProvider	Se usa como ejemplo síncrono para *publisher/subscriber* de evento único.
kpsr::high_performance::EventLoopMiddlewareProvider	Sistema asíncrono de memoria compartida, que crea un *ring buffer* de tamaño conocido y las parejas *publisher/subscriber* asociadas. Es usado generalmente para datos de propósito general, y para mayor rendimiento publica *shared pointers* o usa *SmartPool* (y no hacer reservas de memoria)
Kpsr::zmq_mdlw::toZMQMiidlewareProvider	Módulos que gestionan la (de)serialización de objetos y clases a canales de red, simplificando el ensamblaje.
Kpsr::zmq_mdlw::fromZMQMiidlewareProvider	
kpsr::admin::socket_mdlw::EventLoopSocketAdminContainerProvider	Utilidades para crear *event-loops* basándose en la implementación de *adminContainer* y *sockektServer*. Permiten monitorizar la comunicación y el consumo de recursos del sistema anfitrión.
kpsr::admin::socket_mdlw::EventLoopSocketSystemContainerProvider	

En la tabla 5 se muestran proveedores básicos de memoria compartida y el modo de aplicación que hemos elegido, junto a una descripción básica de su funcionamiento.

El proyecto kpsr-admin, cuyos componentes utiliza el software Melissa, no es *open source* y principalmente se caracteriza por:

- Dos APIs que ofrecen operatividad basada en web y telemetría a los servicios.
- El API REST se inyecta y no necesita ningún cambio en el código (internamente asíncrono)
- El API Node.js también se inyecta en los servicios, y puede requerir algo de desarrollo en Angular.js

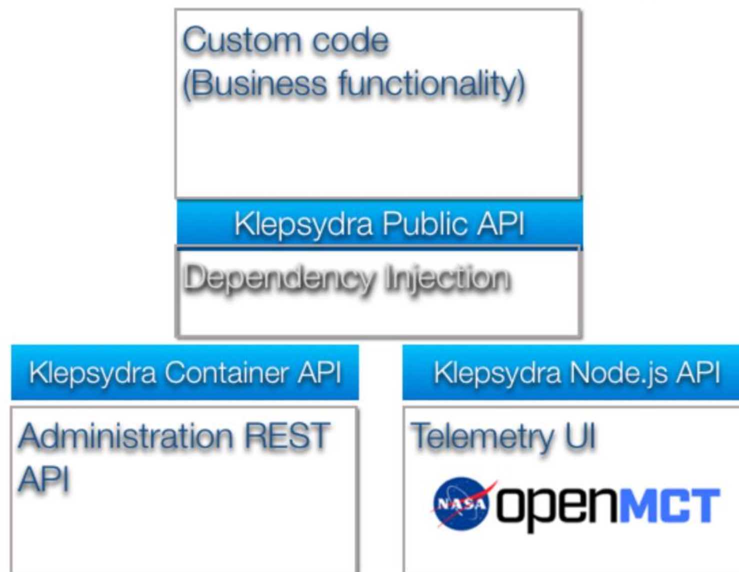


Ilustración 11. Diagrama Klepsydra admin (bitbucket, 2019)

2.3. ZeroMQ

La librería asíncrona de alto rendimiento **ZeroMQ** (también conocida como 0MQ) tiene como funcionalidad clave proporcionar *sockets* de red que unen simplicidad y escalabilidad, lo que resulta en un software limpio y estable. Según los fundadores la creación 'se debe' a los factores de la ilustración 12, y su filosofía se caracteriza por el concepto minimalista de *zero*.

- Cero latencias
- Cero *broker*
- Cero costes
- Cero administración

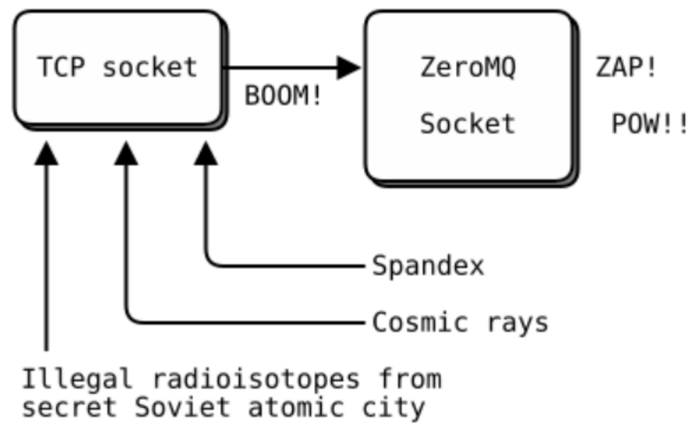


Ilustración 12. Parámetros de creación de ZeroMQ (Hintjens, 2013)

ZeroMQ actúa como un *framework* de concurrencia, aunque parezca una librería integrable de red (Hintjens, 2013). Proporciona *sockets* que llevan mensajes de nivel atómico a través de transportes como *multicast*, TCP, *inter-threads* o *in-thread*. Permite conectar estos *sockets* en relación N a N, con patrones como *publisher-subscriber*, *request-reply, fan-out o task-distribution*.

Además, proporciona **soluciones** para problemas que sufren las aplicaciones en comunicación a través **de red**.

- ✓ Su modelo asíncrono de entrada/salida con *threads* en segundo plano ofrece escalabilidad a las aplicaciones *multicore*, sin usar semáforos o *locks*.
- ✓ Permite acoplar y desacoplar componentes en cualquier orden creando *service-oriented-architectures*.
- ✓ Gestiona lectores lentos o bloqueados de manera segura automáticamente, así como errores de red.
- ✓ No restringe a ningún formato de mensaje específico, y entrega los mensajes tal como fueron enviados.

En cuanto al patrón elegido de *publisher-subscriber* (ilustración 13), el descubrimiento de los distintos nodos se realiza inicialmente a mano o por configuración, y cuando las arquitecturas escalan se hace dinámicamente, con intermediarios o *proxies*.

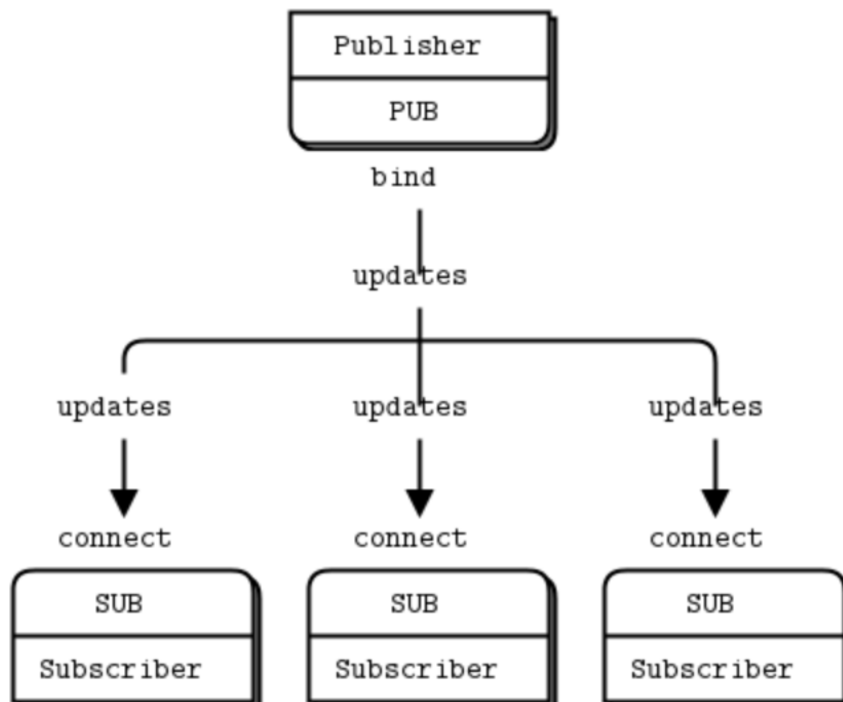
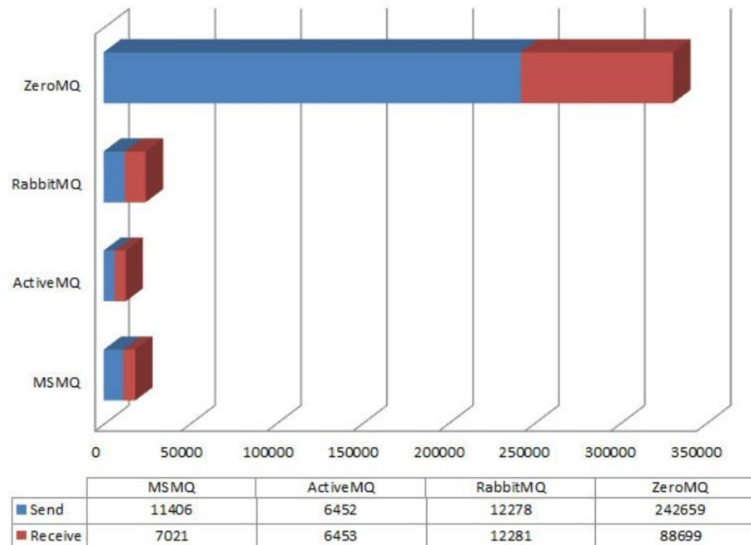


Ilustración 13. Patrón publisher subscriber en ZeroMQ (Hintjens, 2013)

En el software Melissa el patrón utilizado es el *publisher-subscriber*, empleando para ello *wrappers* de Klepsydra. Como reflejo del rendimiento de 0MQ frente a otras soluciones similares como MSMQ, ActiveMQ o RabbitMQ, para enviar mensajes de una aplicación a otra con baja latencia, 0MQ es la mejor opción como según vemos en la ilustración 14 (envío de 1M de mensajes de 1KB), (Hadlow, 2011)



	MSMQ	ActiveMQ	RabbitMQ	ZeroMQ
Send	11406	6452	12278	242659
Receive	7021	6453	12281	88699

Ilustración 14. Comparativa de soluciones de encolado de mensajes (Hadlow, 2011)

Algunas organizaciones cuyos servicios han adoptado ZeroMQ son Microsoft, la NASA, Cisco o Samsung.

Como alternativa similar conocida a ZeroMQ existe **NNG** (nanomsg-next-gen), que es también una librería ligera *brokerless* ofreciendo un API simple que soluciona problemas comunes recurrentes de mensajería, como el *service-discovery*, *publish/subscribe* o *request/reply* con estilo RPC (*remote-procedure-call*). De esta manera el programador puede centrarse en la lógica de sus aplicaciones y no preocuparse por los detalles como la gestión de conexión, reintentos y otro tipo de aspectos.

Ilustración 15. Librería de red NNG (Sustrik,2019)

Nanomsg fue creado por el cofundador de ZeroMQ, siendo un *fork* de este último, debido a que quería añadir integración con el *kernel* Linux, aplicar una licencia MIT y estandarizar el protocolo de red, además de hacerlo cumplimiento POSIX.

En cuanto a alternativas similares que no son *brokerless*, podemos destacar **MQTT** (*Message Queue Telemetry Transport*), que permite envío de mensajes con patrón *publisher-subscriber* de forma sencilla, siendo ligero, abierto y fácil de implementar. Es adecuado para entornos M2M *(machine-to-machine)* o IoT *(Internet of things)*, donde es necesario una huella reducida de código (*footprint*) y el ancho de banda de red es crucial.

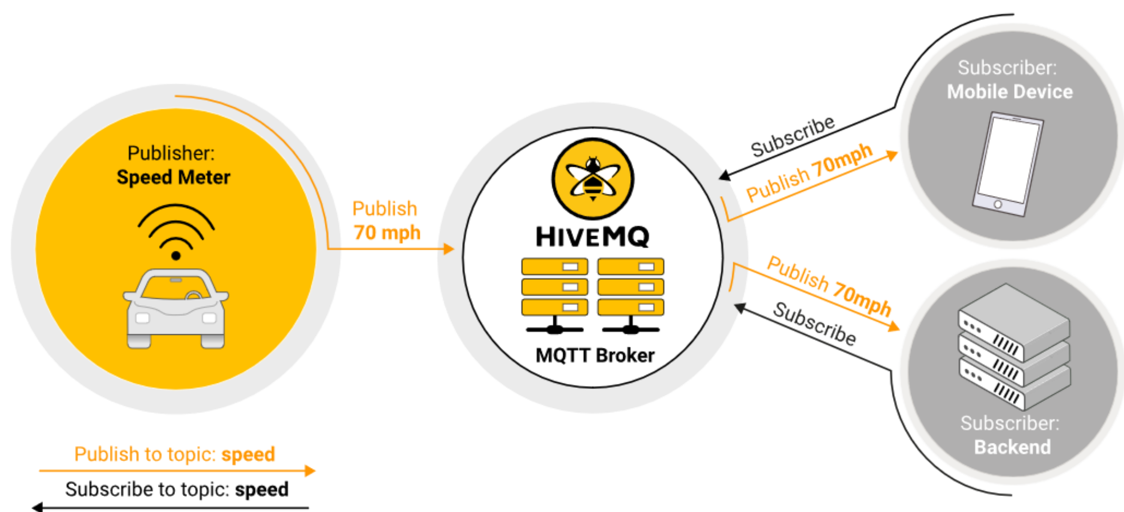


Ilustración 16. Arquitectura MQTT (HiveMQ, 2019)

2.4. Herramientas y tecnologías utilizadas

Entorno de ejecución

Como sistema operativo anfitrión hemos elegido Windows 10 para sistemas de 64 bits (ilustración 17) en su versión Pro, creado por Microsoft en 2015. Este entorno facilita el uso de herramientas de ofimática de cara al desarrollo de la memoria del proyecto, y permite el uso de software instalable solo en esta plataforma.

Ilustración 17. Windows 10 (Microsoft, 2015)

Es en Windows 10 sobre el que hemos instalado la virtualización que ofrece VMware Workstation Player (ilustración 18). Este software permite crear una capa de abstracción de los recursos de la máquina y permite instalar nuevas máquinas virtuales con sus propios sistemas operativos en una máquina física para utilizarlas junto con esta. Sus principales características son:

- Se ejecuta en sistemas operativos x64 de Windows y Linux
- Está disponible de forma gratuita para uso no comercial
- Las máquinas virtuales creadas soportan más de 200 sistemas operativos
- Capacidades de aislamiento del sistema anfitrión, entorno sandbox
- Adecuado para entornos corporativos compatibles con modelos BringYourOwnDevice

Ilustración 18. Virtualización VMWare Workstation Player (VMWare, 1999)

Con VMWare hemos asignado los recursos correspondientes a la máquina virtual y hemos utilizado una imagen de Ubuntu Desktop 18.04 (ilustración 19) para instalar ese sistema operativo. Por razones de compatibilidad con todo el software sobre el que se apoya el proyecto, Ubuntu 18.04 ha sido el entorno elegido para ejecutar el software implementado.

Ilustración 19. Ubuntu Desktop 18.04 (Ubuntu, 2018)

Entorno de desarrollo

Para desarrollar el código se empleó el entorno de desarrollo integrado Visual Studio Code, un editor de código fuente desarrollado por Microsoft (ilustración 20). Ofrece características destacables como son el ser multiplataforma, incluir soporte para depuración, integrar el control de versiones Git, resaltar la sintaxis y autocompletar código, siendo gratuito y de código abierto.

Ilustración 20. Visual Studio Code (VS Code, 2015)

Herramientas de modelado

Para crear algunos diagramas de clases hemos utilizado la herramienta online GenMyModel (ilustración 21). Es gratuita y ofrece un espacio personal para propias creaciones en la nube, e incluso de forma colaborativa. Su uso es sencillo y una característica destacable es que asiste en la creación de tipos de datos que contienen las entidades.

Ilustración 21. Diagramas de clases con GenMyModel (GenMyModel, 2013)

En cuanto la creación de algunos esquemas de comunicación y arquitectura, también hemos utilizado draw.io (ilustración 22), que ofrece gratuitamente bastantes bibliotecas con iconos de todo tipo y permite exportar a múltiples formatos.

Ilustración 22. Creación de esquemas con Draw.io (Draw.io, 2005)

Herramientas de compilación

Para compilar código escrito en lenguaje C++ utilizamos el compilador del proyecto GNU que distribuye la FreeSoftwareFoundation conocido como g++ (Stallman 1987). Con esta herramienta se traducirá el código fuente escrito en las aplicaciones ejecutables que proporcionan la funcionalidad. Durante el proceso de compilación nos avisará de los errores que tenga el código.

Ilustración 23. GNU C++ Compiler (GNU, 1987)

Para automatizar el procedimiento de compilación de los proyectos usamos la herramienta multiplataforma **CMake** creada por Kitware que facilita la configuración y genera makefiles nativos. En los archivos CMakeLists.txt, entre otras cosas, se definen las rutas, se incluyen los encabezados y crean librerías que se enlazan e instalan. Esta herramienta ha ahorrado enormemente el trabajo necesario para la gestión de dependencias del proyecto.

Ilustración 24. Cmake (CMake, 2000)

Herramientas de desarrollo

La herramienta para control de versiones de código utilizada ha sido Git, creada por Linus Torvalds. Es un software cuyo objetivo es llevar registro de los cambios y facilitar un entorno de codificación durante el desarrollo de software.

Ilustración 25. Control de versiones Git (Torvalds, 2005)

La plataforma elegida para alojar los repositorios ha sido Github. Es una plataforma que proporciona un espacio colaborativo y estructura para trabajar y compartir proyectos. La publicación del código generado en los proyectos de este trabajo es uno de los objetivos que se pretenden para contribuir a la comunidad de desarrolladores.

Ilustración 26. Plataforma de desarrollo colaborativo Github (Microsoft, 2007)

Además, se ha integrado la funcionalidad de otros proyectos *open source* alojados en Github:

- Googletest como *framework* de *testing* para C++, que proporciona automatización para las etapas de pruebas unitarias y de integración *(github.com/google/googletest)*
- Cereal como una librería de serialización y deserialización creada por USCilab, como los tipos de datos específicos utilizados en el proyecto Waypoint o ImageData. *(uscilab.github.io/cereal/)*
- Spdlog como una librería de *logging*, que permite establecer niveles de sensibilidad de registro, creada por Gabi Melman. *(github.com/gabime/spdlog)*

Explicada anteriormente, se ha utilizado ZeroMQ como *framework open source* para sistemas de mensajería distribuida como el que nos ocupa, centrándonos en la comunicación TCP con un patrón publisher-subscriber.

Ilustración 27. Librería de mensajería ZeroMQ (iMatix, 2013)

De igual manera, se ha necesitado utilizar algunos componentes de Klepsydra, cuya filosofía es actualizar la ingeniería de software embebido para construir aplicaciones autónomas avanzadas de una forma más eficiente y fiable.

Ilustración 28. Klepsydra Technologies (Ghiglino, 2019)

2.5. Conclusiones

Actualmente, para un máximo aprovechamiento de los recursos computacionales donde las arquitecturas multiprocesador y multinúcleo son dominantes, como vemos existen importantes retos al desarrollar *software* cuyos múltiples threads asíncronos accedan a estructuras de datos compartidas. Es necesario considerar las técnicas actuales no bloqueantes para ofrecer soluciones de mayor rendimiento, aunque a menudo implica mayor complejidad en el diseño.

En cuanto a la creación del sistema, las herramientas elegidas han facilitado en gran medida el ciclo de vida de desarrollo del software, ya sea en control de versiones, automatización de dependencias o los propios componentes software utilizados en el código. Tras investigar estos últimos parece adecuado, para nuestro sistema de productor-consumidor en red, utilizar el patrón *disruptor* como implementación concreta del *event-loop* cuyo uso simplifica Klepsydra. Hemos visto que en este modelo *publisher-subscriber* el uso de *sockets* de tipo ZeroMQ puede solucionar en gran medida los típicos problemas de comunicación en red entre aplicaciones.

3. Objetivos concretos y metodología de trabajo

3.1. Objetivo general

Modelar e **implementar** unas **aplicaciones** y servicios, que al ser ejecutadas permitan **comunicarse** entre sí, a través de protocolos de la capa de **red**. Las ejecuciones parametrizadas de las aplicaciones serán analizadas para reflejar en qué escenarios de número de productores/consumidores y de eventloops se consigue un **óptimo rendimiento sin pérdida** de información en entornos distribuidos.

3.2. Objetivos específicos

El objetivo general puede ser dividido en los siguientes objetivos:

- Recordar **problemas de concurrencia** e introducir el software de Klepsydra y ZeroMQ, mencionando alternativas.
- **Desarrollar servicios** que contengan la funcionalidad de obtener y procesar imágenes y coordenadas de navegación.
- **Implementar aplicaciones** que instancien esos servicios y su comportamiento de comunicación, primero en memoria y después en red. Integrar todos los componentes software necesarios.
- **Evaluar el rendimiento** del sistema en base a unos indicadores y distintos escenarios parametrizados.
- Analizar los resultados y **determinar las conclusiones** correspondientes.

3.3. Metodología del trabajo

La metodología utilizada para llevar a cabo el trabajo ha sido un modelo de **desarrollo iterativo e incremental**. Según indica Sommerville (2011), este modelo sugiere construir partes reducidas de software cuyo tamaño va aumentando para hacer más sencillo detectar problemas importantes antes de que su corrección suponga mayores costes.

En la ilustración 29 se observa el flujo de las etapas en el ciclo de desarrollo de software.

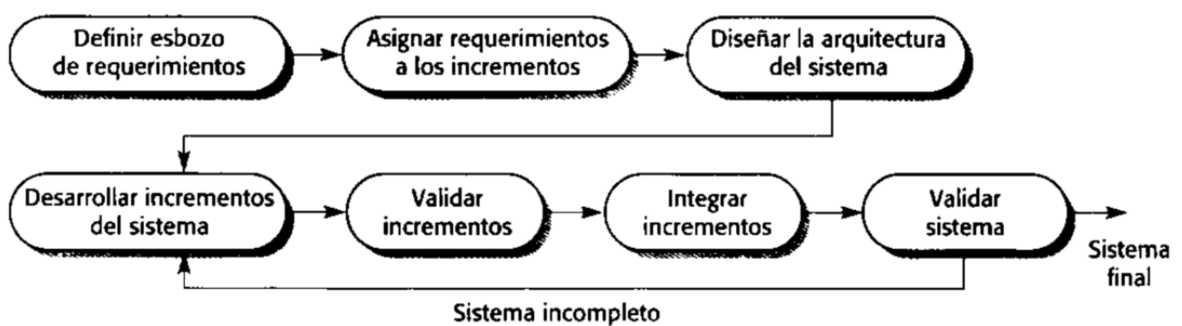


Ilustración 29. Entrega incremental (Sommerville)

Estas etapas reflejan el transcurso del presente trabajo, donde en base al escenario de funcionamiento deseado, se han seguido una serie de procesos:

1. Inicialmente se ha realizado unos prototipos de diagrama de clases que han servido para ayudar a concebir la organización del código.
2. Previo a la codificación se ha escrito un fichero *README* con las instrucciones para instalar el software *thirdparties* requerido por el sistema del proyecto.
3. En cuanto a la implementación del software, se ha creado junto con un prototipo de diagrama, una definición inicial básica de los servicios, estableciendo atributos y métodos con sus parámetros, para codificar después el comportamiento.
4. Después, se han creado los *tests* para los servicios, con el objetivo de probar el buen funcionamiento de sus constructores y procedimientos.
5. La integración de los servicios se ha probado creando aplicaciones que incluyen la interacción entre los distintos módulos, primero mediante mensajes en memoria y después a través de protocolos de red.

6. Tras asegurar la funcionalidad mediante la ejecución de pruebas y aplicaciones, se ha procedido a la etapa de registro de los resultados, utilizando una herramienta de monitorización.
7. Tras una serie de ejecuciones parametrizadas se han analizado, usando ciertos indicadores, los datos obtenidos para hacer posterior validación y elaboración de conclusiones. Para los distintas configuraciones se utilizarán las escenarios propuestos que poseen ciertos requisitos.

Tabla 6. Configuración de escenarios (elaboración propia)

La aplicación tendrá P productores o RoboBees
E consumidores o QueenBees (instancias de *EventLoop*)
E y P son variables dentro del rango 1

En este sentido, evaluaremos esas configuraciones aplicando una metodología de evaluación como es ene ste caso la definición previa de indicadores como los que figuran en la tabla 9 del capítulo 4.3.

4. Desarrollo específico de la contribución

En esta sección vamos primero, en el apartado 4.1, a detallar de qué manera se ha llevado a cabo el proceso de desarrollo del proyecto.

Más adelante, en el apartado 4.2 vamos a explicar el sistema implementado, comenzando por sus dependencias para continuar con su arquitectura y posterior descripción de parte de la codificación.

Una vez definido el funcionamiento, en el apartado 4.3, pasamos a .enumerar los indicadores y evaluar las ejecuciones propuestas en base a estos.

4.1. Organización y transcurso del desarrollo

El desarrollo del código se ha organizado de una forma habitual en el entorno *open* source. Inicialmente hemos creado un **proyecto en Github**, controlando las versiones mediante git, y estructurando ese proyecto en *pull-requests*, que son iteraciones que añaden funcionalidad al software, y permiten una fácil revisión del código en el repositorio rermoto por parte de un colaborador, como se ve en la ilustración 30.

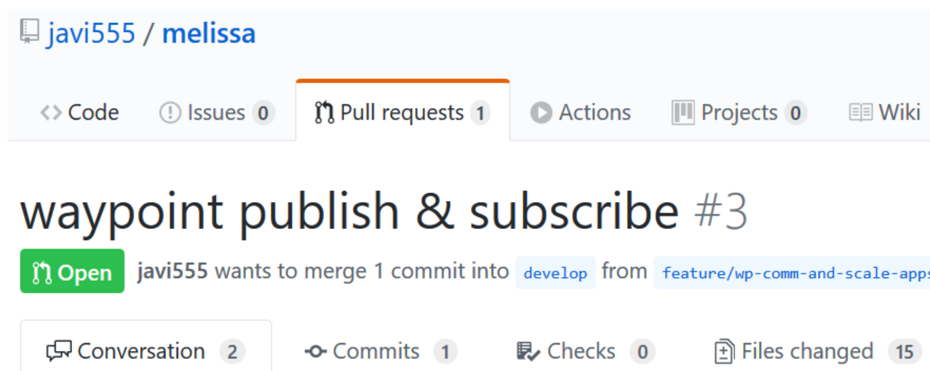


Ilustración 30. Procedimiento de pull-request (elaboración propia)

Los cambios añadidos gradualmente al proyecto se iban subiendo en *commits*, y antes de realizar las fusiones con la rama principal (*merge*), se hacía una compresión de los *commits* conocida como *rebase-squash* como se ve en la ilustración 31, para dar mayor claridad a la historia del proyecto.

```
javi@ubuntu:~/projects/melissa$ git rebase -i 6f7bcd3fa9eabd3f8adb0509d76348cd05d40bc0
[detached HEAD 45cbbe7] waypoint publish & subscribe
 Date: Mon Dec 30 12:35:49 2019 -0800
 11 files changed, 196 insertions(+), 67 deletions(-)
 delete mode 100644 modules/zmq_application/qb_application/src/zmq_middleware_facility.cpp
 create mode 100644 modules/zmq_application/zmq_utils/CMakeLists.txt
 create mode 100644 modules/zmq_application/zmq_utils/include/waypoint_zmq_serializer.h
 rename modules/zmq_application/{qb_application => zmq_utils}/include/zmq_middleware_facility.h (62%)
 create mode 100644 modules/zmq_application/zmq_utils/src/zmq_middleware_facility.cpp
Successfully rebased and updated refs/heads/feature/wp-comm-and-scale-apps.
javi@ubuntu:~/projects/melissa$
javi@ubuntu:~/projects/melissa$
javi@ubuntu:~/projects/melissa$
javi@ubuntu:~/projects/melissa$ git push -f
Username for 'https://github.com': javi555
Password for 'https://javi555@github.com':
Counting objects: 22, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (22/22), 4.97 KiB | 727.00 KiB/s, done.
Total 22 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), completed with 4 local objects.
To https://github.com/javi555/melissa.git
 + 5bfe862...45cbbe7 feature/wp-comm-and-scale-apps -> feature/wp-comm-and-scale-apps (forced update)
javi@ubuntu:~/projects/melissa$
```

Ilustración 31. Procedimiento de rebase (elaboración propia)

Durante el desarrollo ha sido necesario utilizar el d*ebugger* de Visual Studio Code, para poder observar los valores de las variables durante la ejecución de los tests. Asimismo, y como se ve en la ilustración 32, hemos tenido que emplear la herramienta GNU de gdb para analizar los *core dumps* o archivos generados con las últimas instrucciones ejecutadas para poder conocer la causa de fallos de segmentación en memoria, entre otras.

```
32 (gdb) bt
33 #0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:51
34 #1  0x00007fda3a6ec801 in __GI_abort () at abort.c:79
35 #2  0x00007fda3ad41957 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
36 #3  0x00007fda3ad47ab6 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
37 #4  0x00007fda3ad47af1 in std::terminate() () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
38 #5  0x00007fda3ad47d24 in __cxa_throw () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
39 #6  0x00007fda3f432fa1 in zmq::detail::socket_base::bind (this=0x7ffef5ad5a30, addr_=0x7fda3f4a64e8 "tcp://localhost:9002") at /usr/local/include/zmq.hpp:1207
40 #7  0x00007fda3f42dc5b in ZmqMiddlewareFacility::ZmqMiddlewareFacility (this=0x7ffef5ad5df0, context=..., rbUrl="tcp://localhost:9001", imgTopic="image_data", e
41     at /home/javi/projects/melissa/modules/zmq_application/zmq_utils/src/zmq_middleware_facility.cpp:34
42 #8  0x00005623cb698133 in main (argc=2, argv=0x7ffef5adf888) at /home/javi/projects/melissa/modules/zmq_application/qb_application/src/qb_application.cpp:62
43 (gdb) bt full
```

Ilustración 32. Análisis de core dumps (elaboración propia)

En el transcurso del desarrollo se introdujo también un *memory leak*, y para encontrarlo vimos que la herramienta Valgrind nos daba información relevante acerca de la liberación de la memoria reservada por las aplicaciones, que podemos conseguir ejecutando de esta manera:

```
valgrind --tool=memcheck --leak-check=full -v ./app args
```

En la ilustración 33 podemos observar por la información proporcionada (números de línea y *stack trace*) cómo una cantidad de memoria importante se pierde debido a ese problema.

```
==51173== 299,230,165 (122,720 direct, 299,107,445 indirect) bytes in 1,180 blocks are definitely lost
==51173==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-l
==51173==    by 0x39D3F4: kpsr::Publisher<kpsr::vision_ocv::ImageData>*
kpsr::zmq_mdlw::ToZMQMiddlewareProvider::getBinaryToMiddlewareChannel<kpsr::vision_ocv::ImageData>(std:
std::allocator<char> >, int)::{lambda(std::basic_streambuf<char, std::char_traits<char> >*&)#1}::operat
(to_zmq_middleware_provider.h:91)
==51173==    by 0x3C1D33: std::_Function_handler<void (std::basic_streambuf<char, std::char_traits<char
kpsr::zmq_mdlw::ToZMQMiddlewareProvider::getBinaryToMiddlewareChannel<kpsr::vision_ocv::ImageData>(std:
std::allocator<char> >, int)::{lambda(std::basic_streambuf<char, std::char_traits<char> >*&)#1}>::_M_in
std::char_traits<char> >*&) (std_function.h:316)
==51173==    by 0x3BAA10: std::function<void (std::basic_streambuf<char, std::char_traits<char> >*&)>::
(std_function.h:706)
==51173==    by 0x437EA6: kpsr::ObjectPoolPublisher<std::basic_streambuf<char, std::char_traits<char> >
std::char_traits<char> >*&)>) (object_pool_publisher.h:114)
==51173==    by 0x433937: kpsr::ToMiddlewareChannel<kpsr::vision_ocv::ImageData, std::basic_streambuf<c
>*>::internalPublish(kpsr::vision_ocv::ImageData const&) (to_middleware_channel.h:81)
==51173==    by 0x5D32471: kpsr::Publisher<kpsr::vision_ocv::ImageData>::publish(kpsr::vision_ocv::Imag
==51173==    by 0x5D27BD2: mls::RoboBeeSvc::execute() (robo_bee_svc.cpp:51)
==51173==    by 0x38C344: kpsr::Service::runOnce() (service.h:75)
==51173==    by 0x386B02: main (rb_application.cpp:119)
==51173==
```

Ilustración 33. Stack backtrace en rbApplication (elaboración propia)

Junto con la implementación de los servicios hemos creado tests unitarios, y para integrar los servicios con su interacción, hemos creado una aplicación que utiliza la memoria como canal de comunicación llamada *mem_application*, empleando el patrón de diseño *composition-root* e inyección de dependencias.

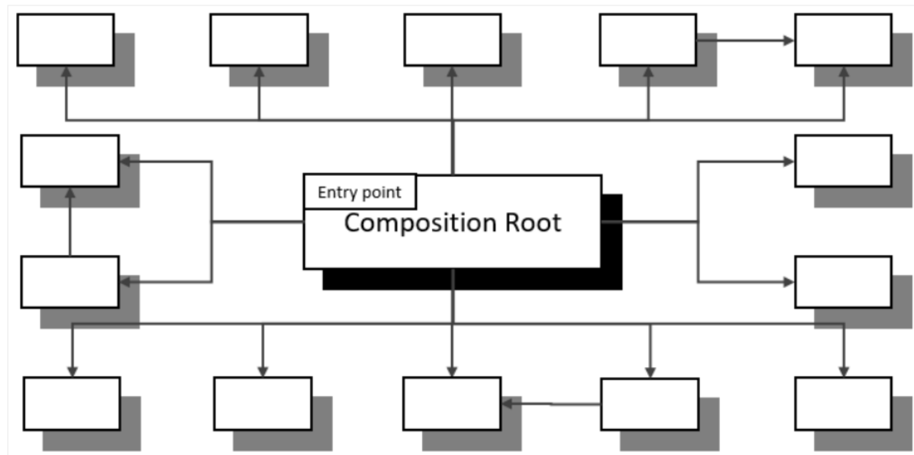


Ilustración 34. Patrón composition root (Van Deursen, 2018)

En cuanto al estilo de codificación, hemos tratado de seguir las directrices de, además de C++11, la convención que proclama CLang, compilador creado por LLVM (Low Level Virtual Machine). LLVM es una infraestructura para desarrollar compiladores y proporcionar herramientas tecnológicas.

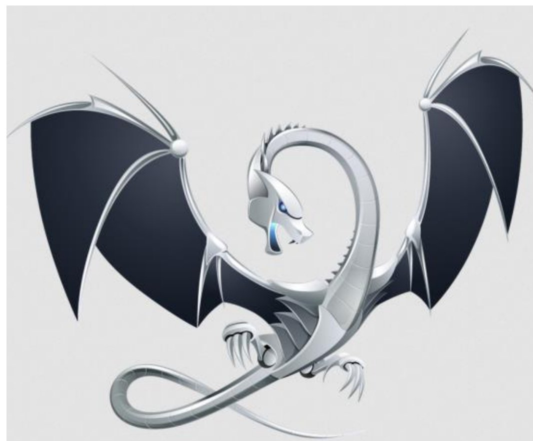


Ilustración 35. LLVM project (LLVM dev group, 2003)

4.2. Descripción del sistema

4.1.1. Requisitos del sistema

En este capítulo se menciona inicialmente algunos requisitos de instalación del software necesario para implementar Melissa.

Instalación de ZeroMQ

Debemos descargar loos proyectos correspondientes, compilar e instalar, como vemos en la siguiente tabla:

Tabla 7. Instrucciones para instalar ZeroMQ (elaboración propia)

```
cd ~/thirdparties/
git clone https://github.com/zeromq/libzmq.git --branch latest_release
cd libzmq/
mkdir build
cd build
cmake ..
make
make test
sudo make install
cd ~/thirdparties/
git clone https://github.com/zeromq/cppzmq.git
cd cppzmq/
git checkout v4.5.0
mkdir build
cd build
cmake ..
make
make test
sudo make install
cd ~/thirdparties/
git clone git://github.com/zeromq/czmq.git
cd czmq
mkdir build
cd build
cmake ..
sudo make install
```

Instalación de Klepsydra

Para este software, ,sobre el que se fundamenta en gran medida las aplicaciones implementadas es necesario instalar, como se ve en la tabla 9, los módulos públicos *kpsr-core* y *kpsr-robotics*, y crear una licencia temporal para el uso de *kpsr-admin* y *kpsr-openMCT*.

Tabla 8. Instrucciones para instalar Klepsydra core y robotics (elaboración propia)

```
rm -rf ~/thirdparties/klepsydra
mkdir -p ~/thirdparties/klepsydra
cd ~/thirdparties/klepsydra
git clone https://github.com/klepsydra-technologies/kpsr-core
~/thirdparties/klepsydra/kpsr-core
cd ~/thirdparties/klepsydra/kpsr-core
git submodule update --init

cd ~/thirdparties/klepsydra/
git clone https://github.com/klepsydra-technologies/kpsr-robotics
~/thirdparties/klepsydra/kpsr-robotics
cd ~/thirdparties/klepsydra/kpsr-robotics
git submodule update --init
```

Para importar estos proyectos en el nuestro usamos las sentencias *find_package(Klepsydra)*, de forma que CMake lo pueda encontrar en el sistema (/usr/local), y tener disponibles sus includes en KLEPSYDRA_INCLUDE_DIRS, así como enlazar sus librerías mediante KLEPSYDRA_CORE_LIBRARIES.

```
85
86   find_package(Klepsydra REQUIRED)
87   include(${KLEPSYDRA_CODE_GENERATOR})
88
```

Ilustración 36. Ejemplo importación de librería en CMake (elaboración propia)

Instalaciones *thirdparties*

Existen otros proyectos que también hemos empleado, como **OpenCV**, **Boost**, **Valgrind** o **nodeJS**. Todos los procedimientos de instalación figuran en el README.md del proyecto Melissa.

Monitorización

Para la visualización de datos se ha elegido el framework *open source* **OpenMCT**, habitualmente empleado para control de misiones, aunque aplicable a otros campos para análisis de datos.

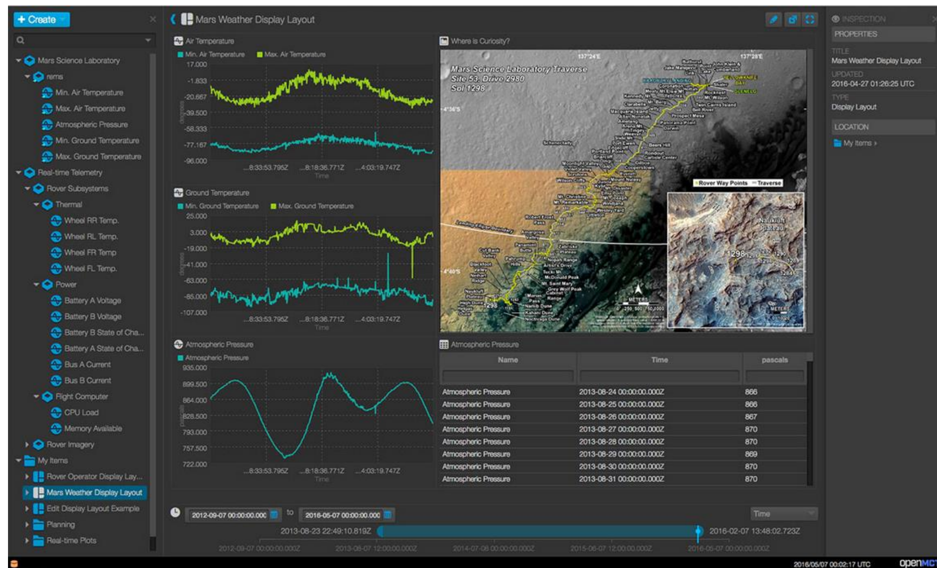


Ilustración 37. Framework Open MCT (NASA, 2015)

Esta herramienta está diseñada para cumplir con la rápida evolución de las necesidades por parte de los sistemas de control de misiones como son operaciones distribuidas, acceso ubicuo a datos y visualización espacial de estos, unido a una fuerte reconfiguración. Está compuesto de ciertas características como son:

- Diseño *responsive* basado en web
- Orientado a objetos y composición parametrizada
- Extensible y con arquitectura flexible

4.1.2. Arquitectura y desarrollo

El sistema propuesto, que denominamos **Melissa**, consistirá en una serie de nodos (RoboBees) con que envían datos a uno o varios nodos centrales (QueenBees) para su procesamiento. El código del proyecto se ha dejado público como contribución *open source*, a la comunidad de desarrolladores.

Ilustración 38. Melissa Github project (Writ in wáter, 2014)

Se han implementado servicios y aplicaciones de ambos, e inicialmente se ha realizado un prototipo de diagrama de clases como guía para empezar la codificación, tal como se observa en la ilustración 39, que esboza un patrón de composición donde la aplicación contiene los servicios que a su vez definen su comportamiento.

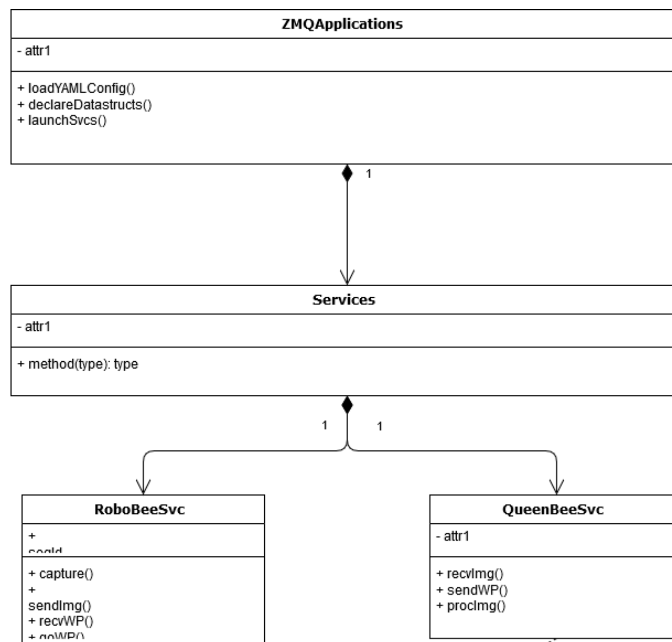


Ilustración 39. Prototipo diagrama de clases (elaboración propia)

Tabla 9. Comportamiento de los servicios RoboBee y QueenBee (elaboración propia)

RoboBee	QueenBee
Cargar configuración e iniciar	Cargar configuración e iniciar
Capturar imágenes	Recibir imágenes de cada RoboBee
Enviar imágenes con identificador	Enviar coordenadas navegación
Recibir coordenadas de navegación de QueenBee	Procesar las imágenes recibidas aplicando detección de contornos circulares

En la tabla 9 se resume la funcionalidad de las entidades implementadas para el sistema creado. Asimismo, en las ilustraciones 27 y 28 podemos ver la especificación de la interfaz de cada una de las entidades, donde hemos indicado los atributos y las operaciones de la clase.

RoboBee

En cuanto a las 'abejas' RoboBee, se componen de un microservicio que ejecuta las funciones mencionadas, y sus miembros pueden ser fácilmente configurables (como la resolución de las imágenes capturadas) mediante ficheros de lenguaje YAML.

```
mls::RoboBeeSvc
_lastImageSeq: int
_lastWpSeq: int
_imgWidth: float
_imgHeight: float
_waypoint: mls::Waypoint
_imageDirName: String
_fileNameList: std::vector<std::string>
_index: unsigned int
_sz: std::vector<int>
_prefix: int
_image: kpsr::vision_ocv::ImageData
*_imageDataPubliisher: kpsr::Publisher<kpsrr::vision_ocv::ImageData>
*_waypointSubscriber: kpsr::Subscriber<mls::Waypoint>
_restartIfNoMoreImages: bool
_fileImagesPath: std::string
RoboBeeSvc(*environment, *imageDataPublisher, *waypointSubscriber)(): mls::RoboBeeSvc
start(): void
stop(): void
execute(): void
create(): void
onWaypointRecv(mls::waypoint &)(): void
hasMoreImages(): bool
getImage(cv::Mat &)(): void
```

Ilustración 40. Especificación de RoboBee (elaboración propia)

Su constructor debe recibir un suscriptor de coordenadas y un publicador de imágenes, además de una serie de parámetros, como vemos en la ilustración 41.

```
RoboBeeSvc::RoboBeeSvc(
    kpsr::Environment *environment,
    kpsr::Publisher<kpsr::vision_ocv::ImageData> *imageDataPublisher,
    kpsr::Subscriber<mls::Waypoint> *waypointSubscriber, int witdh, int height,
    std::string imageDirname, bool restartIfNoMoreImages, int prefix)
    : Service(environment, "robo_bee_service"),
      _imageDataPublisher(imageDataPublisher),
      _waypointSubscriber(waypointSubscriber), _prefix(prefix),
      _imgWidth(witdh), _imgHeigh(height), _imageDirname(imageDirname),
      _lastWpSeq(0), _lastImageSeq(prefix),
      _restartIfNoMoreImages(restartIfNoMoreImages) {}
```

Ilustración 41. Constructor RoboBee (elaboración propia)

Su método start registra un listener de waypoints que asocia una función, y además obtiene una lista de nombres de imágenes, mientras el método stop borra la lista y suprime el listener.

El método execute actualiza el número de secuencia de la imagen capturada, verifica que la imagen cargada cumple con las dimensiones esperadas y la pública, como vemos en la ilustración.

```
void RoboBeeSvc::execute()
{
  _image.frameId = "frame";
  _image.seq = ++_lastImageSeq;
  getImage(_image.img);

  if (_image.img.cols != _imgWidth || _image.img.rows != _imgHeigh)
  {
    spdlog::warn("RoboBeeSvc: Width or heigh of the image should be: {} {}", _imgWidth, _imgHeigh);
    return;
  }
  _imageDataPublisher->publish(_image);
  spdlog::info("RoboBeeSvc: Published Image {}", _image.seq);
}
```

Ilustración 42. Método ejecutado cíclicamente (elaboración propia)

Los métodos getImage y onWaypointReceived leen la matriz con OpenCV y almacenan el waypoint recibido, respectivamente.

QueenBee

Las QueenBees, por otra parte, se comonen de un microservicio que espera a recibir las imágenes enviadas por los RoboBees. Le aplica a cada una un procesamiento de coste computacional (en este caso de detección de figuras via la transformada de Hough), y después publica coordenadas de navegación al Robobee que le envió la imagen. Es también altamente configurable cambiando los archivos YAML.

mls::QueenBeeSvc

*_waypointPublisher: kpsr::Publisher<mls::Waypoint>
*_imageDataSubscriber: kpsr::Subscriber<kpsr::vision_ocv::ImageData>
_circles: std::vector<cv::Vec3f>

QueenBeeSvc(*environment, *imageDataSubscriber, *waypointPublisher)(): mls::QueenBeeSvc
start(): void
stop(): void
execute(): void
create(): void
onImgReceived(kpsr::vision_ocv::ImageData &)(): void
calculateWaypointForImage(const kpsr::vsion_ocv::ImageData &(): mls::Waypoint
processImg((const kpsr::vision_ocv::ImageData &(): void

Ilustración 43. Especificación de QueeenBee (elaboración propia)

Por el contrario, el constructor de este servicio recibe un suscriptor de imágenes y un publicador de coordenadas de navegación, y en su método de start registra el *listener* que vigilará la recepción de imágenes.

El método de procesar la imagen recibida realiza sin necesitar una copia, como se ve en la ilustración 44, las operaciones matemáticas correspondientes para aplicar a la imagen la detección de contornos.

```
void QueenBeeSvc::processImg(const kpsr::vision_ocv::ImageData &img) {
  cv::Mat gray;
  cv::cvtColor(img.img, gray, cv::COLOR_BGR2GRAY);
  medianBlur(gray, gray, 5);

  HoughCircles(gray, _circles, cv::HOUGH_GRADIENT, 1, gray.rows / 16, 100, 30,
               1, 30);
  spdlog::info("number of circles: {}", _circles.size());
  return;
}
```

Ilustración 44. Procesamiento imagen (elaboración propia)

También, al recibir una imagen y procesarla, el servicio QueenBee procede a calcular un Waypoint (coordenada espacial) que después publica al *middleware*..

```yaml
waypoint_descriptor.yaml
modules > codegen > kidl > waypoint_descriptor.yaml
1  class_name: mls::Waypoint
2  middlewares:
3      - type: ZMQ
4        project_name: melissa_codegen
5        already_exists: false
6        class_name: WaypointMessage
7  fields:
8      - name   : seq
9        type   : int32
10     - name   : x
11       type   : float32
12     - name   : y
13       type   : float32
14     - name   : z
15       type   : float32
```

Ilustración 45. Definición de waypoint (elaboración propia)

Para definir la clase *Waypoint* hemos optado por usar un generador automático de código, que recibe archivos de formato YAML, para construir entidades utilizables, como reflejan las ilustraciones 45 y 46.

```cpp
namespace mls {

class Waypoint  {
public:

   Waypoint() {}

   Waypoint(
          int seq,
          float x,
          float y,
          float z)
      : seq(seq)
      , x(x)
      , y(y)
      , z(z)
   {}
```

Ilustración 46. Clase resultante con constructor y métodos básicos (elaboración propia)

A la hora de cargar configuración para las aplicaciones hemos empleado archivos de formato YAML (lenguaje de fácil lectura para serialización de datos). En la ilustración 47 vemos un ejemplo de un archivo de este tipo. En las aplicaciones, utilizamos la ruta del fichero para cargar su contenido mediante un conversor YAML/C++ (conocido como yaml-cpp *parser*), y una vez cargados, accedemos a esos valores con *getters* asignándolos en variables disponiendo también de un *environment* que los contiene.

```
modules > config > yaml > ! rb_config.yaml
1   file_images_path: "/home/javi/projects/melissa/tests/data"
2   img_width: 320
3   img_height: 240
4   restart_if_no_more_images: true
5   robo_bee_prefix: 100000
6   img_cv_type: 16
7   image_topic: "image_data"
8   waypoint_topic: "waypoint_topic"
9   pool_size: 256
10  qb_img_url: "tcp://*:9001"
11  qb_wp_url: "tcp://localhost:9002"
12  container_admin_name: "RBApp1"
13  server_admin_port: 9292
14  server_system_port: 9393
```

Ilustración 47. Archivo de configuración YAML (elaboración propia)

RBApplication y QBApplication

Las aplicaciones en las que se centra el sistema se comportan de la siguiente forma:

1. Cargamos en cada una la configuración correspondiente
2. Creamos los *event-loop* necesarios
   - Sistemas generales de memoria compartida
   - Gestionan datos de cualquier tipo. Sólo hay que especificar el *datatype* cuando se usa *get* [*publisher* | *subscriber*]
   - Requieren el tamaño del *ring buffer* como parámetro *template*
   - Son necesarios su *start* y *stop* para crear el hilo que escuchará y gestionará todos los demás *listeners* asociados a *subscribers*
   - Es importante suprimir los *listeners* antes de ejecutar *stop*.
   - Los *providers* de *system* y *admin* servirán para monitorizar distintos aspectos del sistema (frecuencias de publicación, consumo de recursos…)

3. Establecemos los *sockets* de red de ZeroMQ, e instanciamos las entidades que se encargan de serializar los objetos y enviarlos al *middleware*.
4. Instanciamos entonces el microservicio correspondiente (RoboBeeService, QueenBeeService) y los dejamos ejecutando en bucle continuo. Los robobees por tanto capturan y envían continuamente imágenes, y los queenbees las reciben, procesan y envían el waypoint correspondiente.

Hemos optado por refactorizar parte de QBApplication, creando una clase de utilidad que facilita la lógica y creación del componente encargado de recibir por red las imágenes y serializarlas. Esta entidad, cuyo constructor se muestra en la ilustración 48, permite instanciar varios *event-loops* para crear las relaciones entre productores y consumidores.

```
ZmqMiddlewareFacility(zmq::context_t &context,
                      std::string rbImgUrl, std::string rbWpUrl, std::string imgTopic, std::string wpTopic,
                      kpsr::high_performance::EventLoopMiddlewareProvider<EVENT_LOOP_SIZE> &eventloop,
                      int poolSize, kpsr::vision_ocv::ImageDataFactory &factory,
                      kpsr::Container *container)
```

Ilustración 48. Constructor de ZMQ_middleware_facility (elaboración propia)

En la ilustración 49 podemos observar un diagrama general de comunicación y abstracción del funcionamiento de las aplicaciones descrito anteriormente. Por el momento ejecutamos los nodos en la misma máquina, pero al disponer de comunicación por red, podemos extrapolar el sistema usando redes ethernet o inalámbricas.

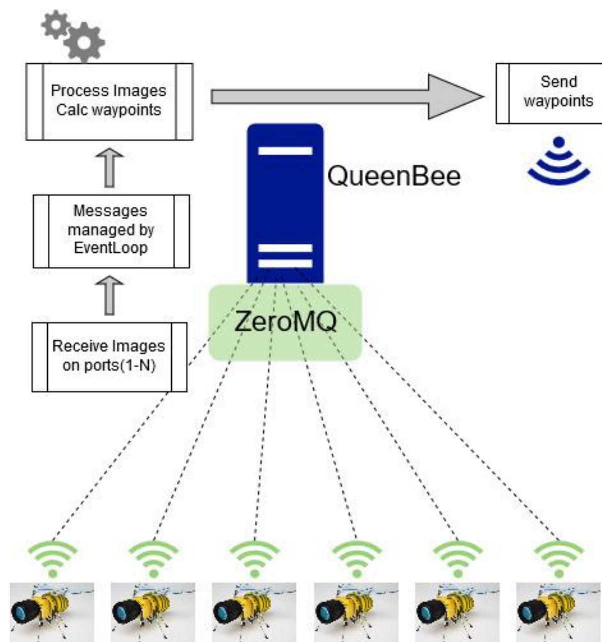


Ilustración 49. Diagrama de comunicación (elaboración propia)

4.3. Evaluación de escenarios de ejecución

Los indicadores que van a evaluar el rendimiento del sistema figuran en la tabla 10 con su descripción. El resto de variables no las tenemos en cuenta (ratio de publicación, eventos reservados, *timestamps*) puesto que de momento no son relevantes para nuestro análisis.

Tabla 10. Indicadores de evaluación de rendimiento (elaboración propia)

Indicadores	Descripción
Img subscription rate	Frecuencia con la que QueenBeeApp recibe imágenes
Wp subscription rate	Frecuencia con la que RoboBeeApp recibe waypoints
Discarded events	Número de eventos publicados no recibidos
Used RAM	Memoria RAM usada por aplicación o total
Used CPU	Uso del procesador por aplicación o total

Para poder realizar el análisis del sistema, dejamos ejecutando el número de aplicaciones correspondientes en casa caso indefinidamente y poder observar así su comportamiento estacionario. Esto se puede apreciar en la ilustración 50, donde dos pares de instancias de aplicaciones QueenBee/RoboBee se envían imágenes y waypoints.

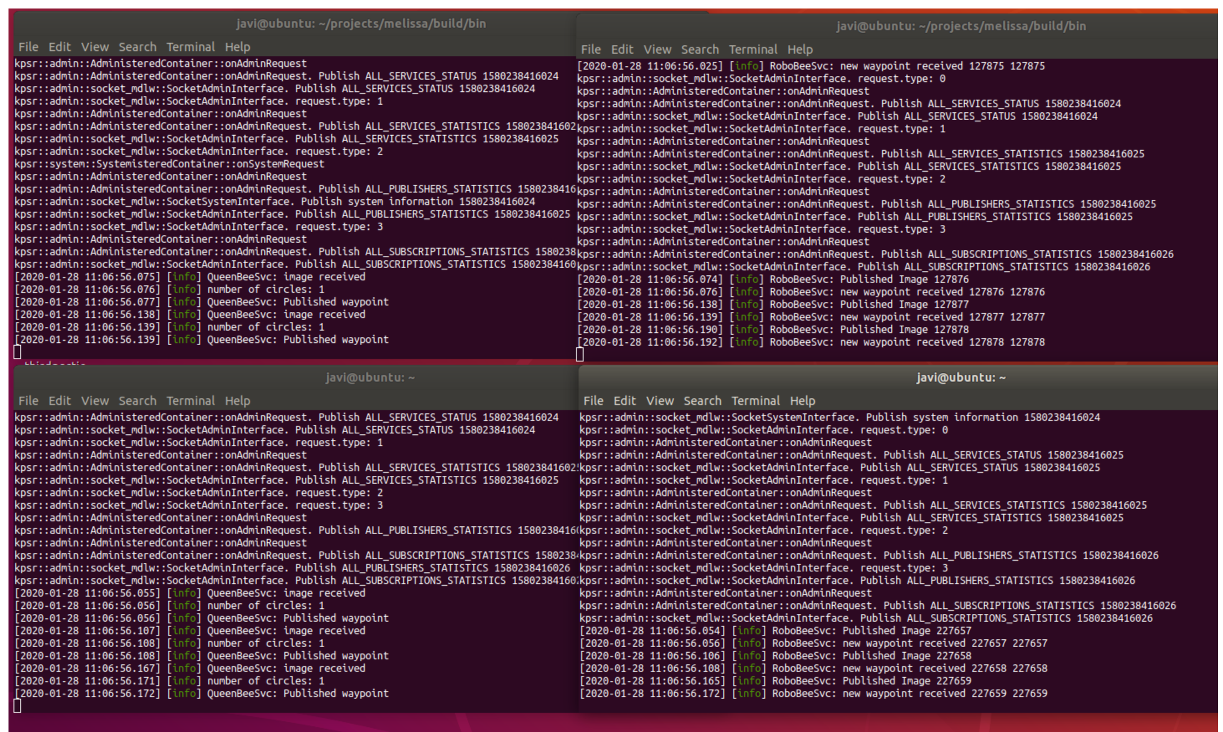


Ilustración 50. Lanzamiento de aplicaciones en Melissa (elaboración propia)

La evaluación consiste en la **comparación** de las distintas **asociaciones de productores a consumidor** que vemos en los siguientes diagramas. Dentro de ellas, se varía entre tres **frecuencias de publicación** y tres **resoluciones** de imágenes capturadas, y se hace fácilmente gracias a diseñar las aplicaciones con alta configuración del entorno.

La estructura que se ha seguido es comprobar el ratio de suscripción de imágenes y de waypoints, mostrar si se han perdido eventos, así como CPU y memoria RAM utilizada por proceso.

Para estas ejecuciones hemos utilizado VMWare virtualizando determinados componentes **hardware**:

- 8 *cores* (2.6Ghz) de un procesador Intel i7-9750H
- 8192 MB de memoria RAM DDR4
- 80 GB SSD m.2 NVMe

4.3.1. Escenario E == P

En este caso, donde el número de productores y consumidores es el mismo, el *event-loop* se comporta como una cola de mensajes. Las instancias de los productores se comunican mediante el mismo canal de red con su consumidor.

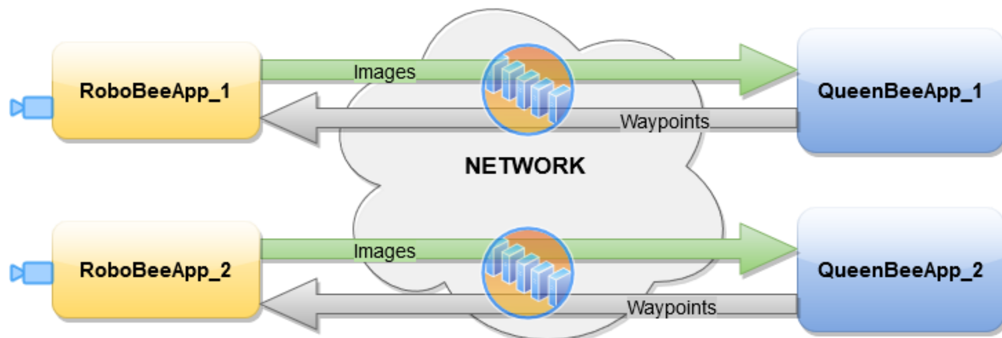


Ilustración 51. Igual número de productores y consumidores (elaboración propia)

En este modelo, vemos en la ilustración 52 que ningún evento ha sido descartado y por tanto no hay pérdida de mensajes para las aplicaciones.

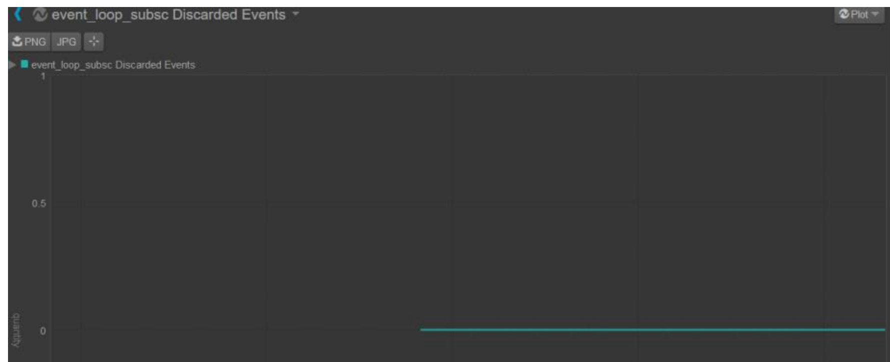


Ilustración 52. Número de eventos descartados (elaboración propia)

En cuanto al ratio de suscripción a imágenes por parte de las aplicaciones QueenBee, se puede ver en la ilustración 53 que mantiene una frecuencia constante en torno a 18hz. De la misma manera, la aplicación RoboBee consigue mantener ese mismo ratio de suscripción (ilustración 54)

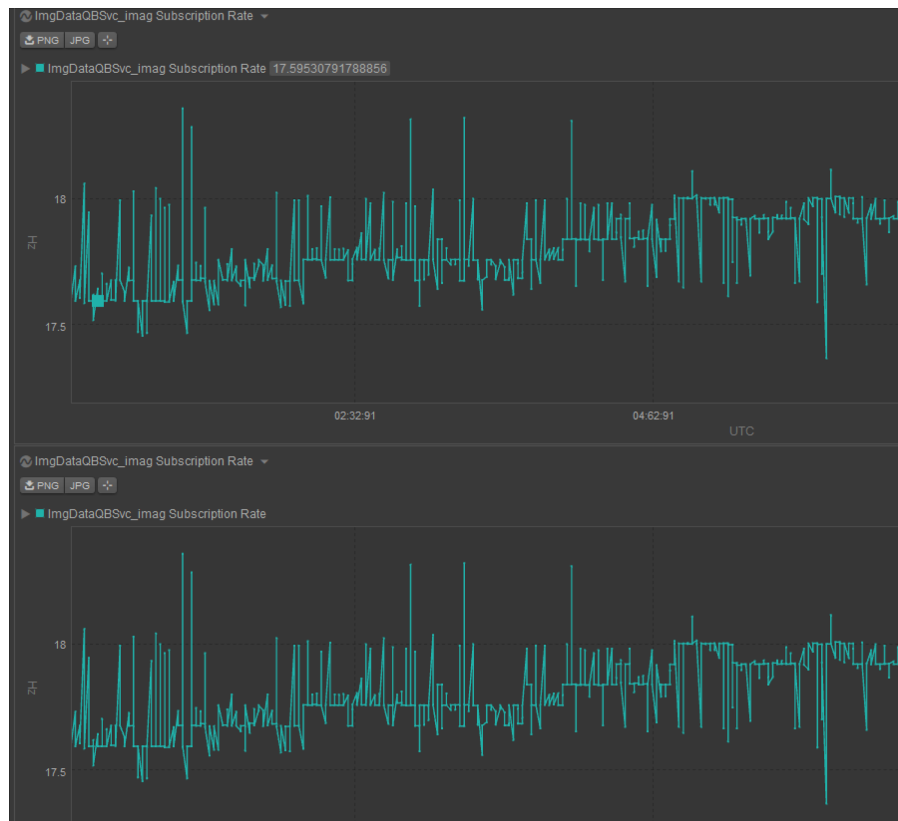


Ilustración 53. Frecuencia suscripción imágenes (elaboración propia)

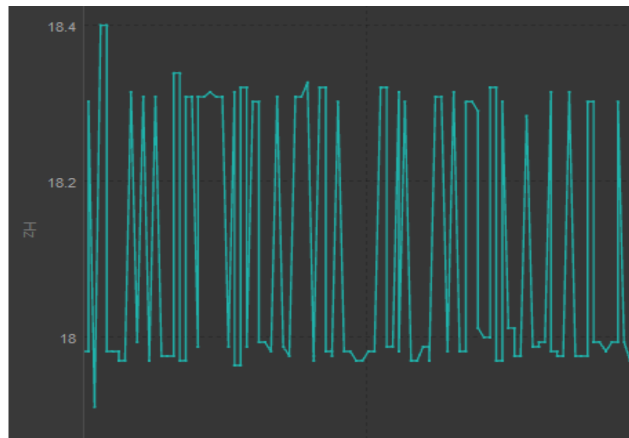


Ilustración 54. Frecuencia suscripción waypoints (elaboración propia)

Además, tanto el consumo de procesamiento (entre 6 y 10%), como el de memoria RAM (en un rango de 20MB) son estables durante la ejecución tal como se observa en la ilustración 55.

Ilustración 55. Estabilidad en CPU y RAM, respectivamente (elaboración propia)

En la tablas 11, 12 y 13 se muestran los datos proporcionados por OpenMCT para este escenario.

Tabla 11. Frecuencia suscripción 20hz (elaboración propia)

Monitorización 20hz (average)	**320x240**		**640x480**		**1280x960**	
	QueenBee	RoboBee	QueenBee	RoboBee	QueenBee	RoboBee
Ratio de suscripción	18.5	18.5	17	17	9.8	10
Eventos descartados	0	0	0	0	0	0
Memoria utilizada (MB)	106	19	365	22	1345	39
CPU utilizada (%)	2.5	0.6	5.7	1.4	15	6.5

Tabla 12. Frecuencia suscripción 10hz (elaboración propia)

Monitorización 10hz (average)	**320x240**		**640x480**		**1280x960**	
	QueenBee	RoboBee	QueenBee	RoboBee	QueenBee	RoboBee
Ratio de suscripción	9.6	9.5	9	9	6.35	6.35
Eventos descartados	0	0	0	0	0	0
Memoria utilizada (MB)	102	18.5	365	20.5	1340	38
CPU utilizada (%)	2	0.5	3.5	1.5	10	4.6

Tabla 13. Frecuencia suscripción 5hz (elaboración propia)

Monitorización 5hz (average)	**320x240**		**640x480**		**1280x960**	
	QueenBee	RoboBee	QueenBee	RoboBee	QueenBee	RoboBee
Ratio de suscripción	4.9	4,9	4.85	4.85	3.8	3.85
Eventos descartados	0	0	0	0	0	0
Memoria utilizada (MB)	103	19	360	22	1320	34
CPU utilizada (%)	1	0	1.5	0.8	5.5	3.2

4.3.2. Escenario E == 1

En este escenario, todos los productores envían a un mismo consumidor, con cuyo *event-loop* itera en todos los suscriptores registrados.

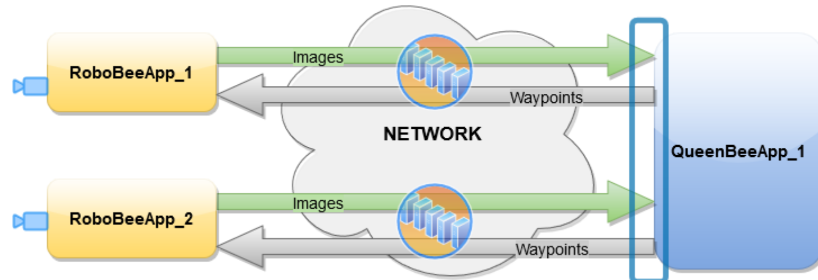


Ilustración 56. Productores a un consumidor (elaboración propia)

4.3.3. Escenario 1 < E < P

En este caso se propone un mapeo de productores a consumidores, siendo configurable el factor de agrupación (R=P/E). Por ejemplo, con 16 robobees y 4 queenbees, cada uno registrará 4 suscripciones.

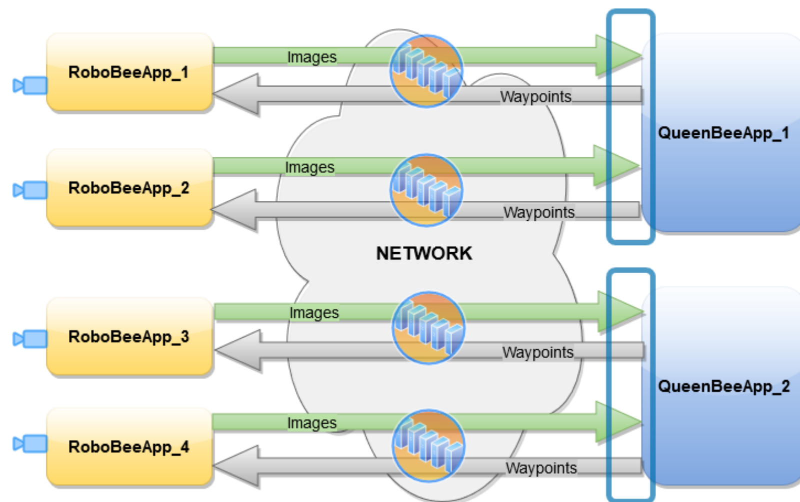


Ilustración 57. Escenario de asociación mixto (elaboración propia)

Para estos dos últimos escenarios es necesario realizar aún implementaciones en el código para adaptar el QueenBee a agrupaciones de múltiples RoboBees con varios *event-loop*.

5. Conclusiones y trabajo futuro

5.1. Conclusiones

En este trabajo pretendíamos conocer la configuración óptima para un sistema distribuido propuesto, con un modelo *publisher-subscriber* en red y utilizando técnicas no bloqueantes tratando de obtener el mejor rendimiento. Para ello hemos implementado el software del proyecto Melissa integrando bucles de eventos especiales, describiendo algunos escenarios parametrizados para validar la solución.

Las contribuciones han seguido la línea de describir y utilizar tanto el framework de red ZeroMQ para la comunicación, como versiones concretas de *event-loop* y *disruptor* del software Klepysdra para gestionar los mensajes enviados entre aplicaciones. Tras completar el desarrollo permitiendo su monitorización, hemos obtenido datos con distintas frecuencias y resoluciones de imagen.

Los resultados concluyen una gran estabilidad y predictibilidad de la memoria utilizada por los nodos, garantizando también la recepción de cada mensaje y con una frecuencia de suscripción muy similar a la de publicación. Con la proliferación de sistemas robóticos de todo tipo, los desarrolladores de su software pueden valerse de las ventajas de modelos como el presentado, ofreciendo mejor desempeño que enfoques de *singlethread* o de *multithread* tradicional. Se considera que los objetivos planteados en términos de implementación y rendimiento del sistema se cumplen en base a los resultados,

5.2. Líneas de trabajo futuro

Respecto a líneas de posible ampliación del alcance del proyecto que no se han incluído, se podrán considerar los siguientes apartados:

- Desarrollo de la funcionalidad para proporcionar una comparativa de otros escenarios de agrupación.
- Implementación de funcionalidad de navegación añadiendo cámara y controladora de vuelo a los RoboBees junto con componentes software como *mavros.*
- Uso de plataformas gráficas como GPGPUs de Nvidia en los Queenbees para procesamiento *real-time* durante vuelo.
- Integración de modelos de dinamica de UAV e inteligencia de enjambre asociados a los Robobees
- Aplicación de técnicas deep learning con modelos de redes neuronales para una detección y clasificación profunda de entidades en base a las imágenes capturadas por los Robobees.
- En base a los resultados tras procesamiento, calcular un algoritmo de trayectorias óptimo para ofrecer máxima eficiencia en la autonomía de vuelo.

6. Bibliografía

Catarino, M. (2019) *Event loop description*. Recuperado (en diciembre 2019) de https://gist.github.com/kassane/f2330ef44b070f4a5fa9d59c770f68e9

Dijkstra, E. (1965) *Solution of a Problem in Concurrent Programming Control.* Recuperado (en diciembre 2019) de https://www.di.ens.fr/~pouzet/cours/systeme/bib/dijkstra.pdf

Dijkstra, E (1971) *Hierarchical ordening of sequential processes* Recuperado (en enero 2020) de http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.pdf

Eugster, P. Guerraoui, R. Felber, P. Kermarrec, A. (2003). *The many faces of publish subscribe*. Recuperado (en enero 2020) de https://www.researchgate.net/publication/220566321_The_Many_Faces_of_PublishSubscribe

Fowler, M. (2011) The LMAX Arquitecture. Recuperrado (en febrero 2020) de https://martinfowler.com/articles/lmax.html

Fuller, S. Miller, L. (2011) *The future of computing performance. Game over or next level?* (pp 155-159) Washington D.C. The National Academies Press

Gartner (2017) *Personal and comercial drones revenue forecast, 2016-2017*. Recuperado (en enero 2020) de https://www.gartner.com/en/newsroom/press-releases/2017-02-09-gartner-says-almost-3-million-personal-and-commercial-drones-will-be-shipped-in-2017

Ghiglino, P (2019). *API for application development in Klepsydra*. Recuperado (en diciembre 2019) de https://github.com/klepsydra-technologies/kpsr-core/blob/master/api-doc/api-kpsr-application.md

Ghiglino, P (2019). Deterministic and high throughput data processing for cubesats. Recuperado (en enero 2020) de https://www.slideshare.net/PabloGhiglino/deterministic-and-high-throughput-data-processing-for-cubesats

Göetz, B. Peierls, T. Bloch, J. Bowbeer, J. Holmes, D. Lea, D. (2006). *Java concurrency in practice* (p.41) Addison-Wesley https://pdfs.semanticscholar.org/3650/4bc31d3b2c5c00e5bfee28ffc5d403cc8edd.pdf

Goldman Sachs (2016) *Technology driving innovation:drones*. Recuperado (en enero 2020) de https://www.goldmansachs.com/insights/technology-driving-innovation/drones/

Hadlow, M. *Message queue shootout!*. Recuperado (en enero 2020) de https://mikehadlow.blogspot.com/2011/04/message-queue-shootout.html

Harper, E. Thijmen de Gooijer (2014) *Performance Impact of Lock-Free Algorithms on Multicore Communication APIs* (p.14) https://arxiv.org/ftp/arxiv/papers/1401/1401.6100.pdf

Hintjens, P. (2013) *ZeroMQ: Messaging for many applications.* O'Reilly.

Miletitch, R. Reina, A. Dorigo, M. Trianni, V. (2019). *Emergent naming of resources in a foraging robot swarm.* Recuperado (en enero 2020) de https://www.researchgate.net/publication/336316898_Emergent_naming_of_resources_in_a_foraging_robot_swarm

Namiot, D. (2016) *On lock-free programming patterns*. Recuperado (en enero 2020) de https://www.researchgate.net/publication/295858544_On_lock-free_programming_patterns

Lafreniere, D. (2017) *C++ std::thread Event loop with message queue and timer.* Recuperado (en enero 2020) de https://www.codeproject.com/Articles/1169105/Cplusplus-std-thread-Event-Loop-with-Message-Queue

Oneill, B. (2018) *Lock-free programming*. Recuperado (en enero 2020) de https://www.robocomtech.com/lock-free-programming/

Oliveira, S. (2018) *Getting Started with CoralQueue*. Recuperado (en enero 2020) de http://www.coralblocks.com/index.php/getting-started-with-coralqueue/

Rinaldi, F. (2019) *Lock-free multithreading with atomic operations*. Recuperado (en diciembre 2019) de https://www.internalpointers.com/post/lock-free-multithreading-atomic-operations

Sommerville, I. (2016). *Ingeniería de software*, Séptima edición, (pp 66-67). Madrid: Pearson.

Stephen N. Freund, Shaz Qadeer: (2004) *Checking ConciseSpecifications For Multithreaded Software.* inJournal of Object Technology, (pp. 81–101) http://www.jot.fm/issues/issue_2004_06/article4.pdf

Sustrik, M. (2019) *NNG, Lightwigth Messaging Library*. Recuperado (en enero 2020) de https://github.com/nanomsg/nng

Tezer, O. (2013) *How to work with the ZeroMQ messaging library*. Recuperado (en enero 2020) de https://www.digitalocean.com/community/tutorials/how-to-work-with-the-zeromq-messaging-library

The HiveMQ Team (2019). *Getting started with MQTT.* Recuperado (en enero 2020) de https://www.hivemq.com/blog/how-to-get-started-with-mqtt/

Thompson, M. Farley, D. Barker, M. Gee, P. Stewart, (2011) A. *High performance alternative to bounded queues for exchanging data between concurrent threads.* Recuperado (en diciembre 2019) de http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf

VanDeursen, S. Seemann, M. (2019) *Dependency injection principles, practices and patterns.* Recuperado (en enero 2020) de https://freecontent.manning.com/dependency-injection-in-net-2nd-edition-understanding-the-composition-root/

Wyss Institute de la Universidad de Harvard (2019) *RoboBees: Autonomous Flying Microrobots.* Recuperado (en enero 2020) de https://wyss.harvard.edu/technology/robobees-autonomous-flying-microrobots/

Anexos

Anexo I. Artículo

Introduction

When developing solutions, in order to obtain best performance, programmers use concurrent software with multiprocessors architectures (Sutter, 2009). However, it is complex to considera ll factors (such as problems arising from access to shared resources) to achieve high efficiency in distributed systems as in the one proposed.

The development of a system is proposed where a series of nodes (conceptually small drones) Will communicate with each other though the network layer by sending messages with a pattern of publication and subscription in relation to autonomous navigation and image capture-processing.

We will try to reflect in which scenarios we achieve the highest performance, varying parameters such as frequencies, resolution or producer-consumer associations. The proposal is born due to the increasing number and improvement of benefits of autonomous distributed systems (Goldman Sachs, 2016), as well as its sensorization and creation of functional prototypes (Wyss Institute, 2019). Given this situation, Ghiglino (2019) highlights the emergence of software challenges sush as delays in data delivery, underutilization of resources or limited functionality and errors.

State of the art

In the world of computing, with the appearance of multi-threaded architectures, the concurrent programming model emerges allowing several threads to be used in the context of the same process. Today, multi-core CPUs have become commodity items. While clock speedsare stable at around 2-3GHz, the number of cores per-chip is doubling every18-24 months. The spread of multi-core CPUs outfrom the data centre will continue so that top-end server CPUs come with 64 cores and this growth will continue, indefinitely. (Hintjens and Sustrik, 2010).

Among the solutions is the mutex technique to execute critical sections of code, which allows orderly access to shared resources to avoid inconsistencies and errors. Using locks is inherently unscalable. A good alternative is to consider non-blocking synchronization techniques that have usually lower cost, such as lock-free or wait-free.

Being mutex easier to implement, it has less determinism and forces to make context changes and cache invalidation in blocking situations, which ruins the performance. On the other hand, lock-free solutions involve a more complex implementation but greater predictability and efficiency.

Namiot (2016) ensures that the greatest challenge for concurrent programming is the coordination and organization of the data flow, rather than entering relatively low-level data structures. In this respect of data flow control, a remarkable design pattern is the event-loop.

Event loop

The concept of event-loop is understood as a thread that waits and distributes incoming messages to an event control function, and usually uses a message queue to contain the incoming events. Each message is decoded and the corresponding action is performed. It is commonly used as a tool to control communications between processes. (Lafreniere, 2017). The operation of the event-loop is based on several data providers and a consumer who is given data sequentially trying to avoid loss of messages.

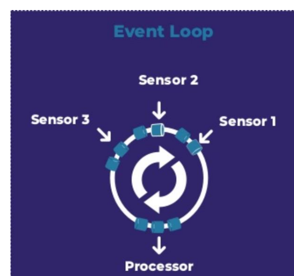


Fig 1. Event-loop scheme (Klepsydra, 2019)

Disruptor

It was designed by LMAX Exchange to provide a high performance and low latency work queue in asynchronous event processing architectures. This pattern ensures that any data is owned by a single thread for write access, thus reducing contention compared to other data structures. (Fowler, 2011)

In summary, the disruptive pattern is conceived as a ring buffer filled with pre-assigned transfer objects that use memory barriers to synchronize producers and consumers through sequences. The importance of the ringBuffer in this work is that it is the mechanism used by the Klepsydra EventLoopMiddlewarerProvider used in our applications.

Klepsydra

Klepsydra is described by Ghihlino as high performance data processing software engineering for embedded systems, although its fundamentals come from the investment banking sector:

- It has components for robotics, aerospace and space environments, offering a rapid pace of development
- It is an independent and lightweight platform, integrated with Matlab and ROS
- Offers increased performance in data processing, especially in stability and scalability
- Provides facilities for solving complex problems

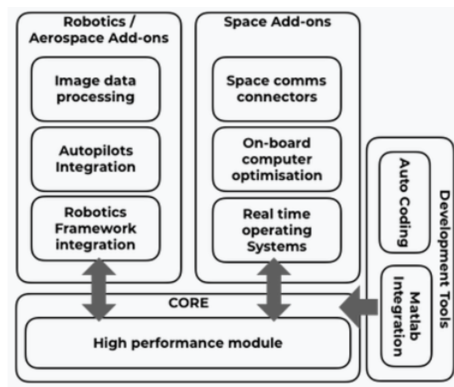


Fig 2. Components interatction diagram (Klepsydra, 2019)

ZeroMQ

ZeroMQ is a concurrency framework similar to an integrated network library that provides sockets with multicast, TCP, inter-threads or in-thread support. It allows you to connect them with common patterns such as publisher-subscriber in our case, all to offer solutions to the typical problems that applications suffer when communicating in a network.

Objetives

Model and implement some applications and services, which when executed allow to communicate with each other, through protocols of the network layer. The parameterized executions of the applications will be analyzed to reflect in which scenarios of number of producers / consumers and eventloops, optimum performance is achieved without loss of information in this distributed environment.

- Remember concurrency problems and introduce the Klepsydra and ZeroMQ software, mentioning alternatives.
- Develop services that contain the functionality of obtaining and processing images and navigation coordinates.
- Implement applications with those services and their communication behavior, first in memory and then in network. Integrate all necessary software components.
- Evaluate system performance based on indicators and different parameterized scenarios.
- Analyze the results and determine the corresponding conclusions.

.

Software Development

The **Melissa** system will consist of a series of 'RoboBees' nodes that communicate with others called 'QueenBees', whose functionality is shown in table 1. For this purpose, it has been necessary to use the ZeroMQ, Klepsydra and OpenMCT frameworks, among other components.

Table 1. Services funtionality

RoboBee	QueenBee
Load configuration and start service	**Load configuration and start service**
Capture images	Receive images of each RoboBee
Send images with identifier	Send navigation coordinates (waypoint)
Receive waypoints from QueenBee	Process the images received by applying circular contour detection

The developed applications are composed of, prior to loading configuration per file, instances of Klepsydra EventLoopMiddlewareProviders as shared memory systems with the size of the ring-buffer (lock-free data queue network) as the only parameter, and when a thread starts it will listen and will manage all listeners associated with subscribers.

The Klepsydra EventLoopSocket[admin | system]ContainerProvider components will be used to monitor certain values such as resource consumption or publication frequencies. A general communication diagram is shown in figure 3, where applications serialize the objects and send over the network sockets in a continuous loop.

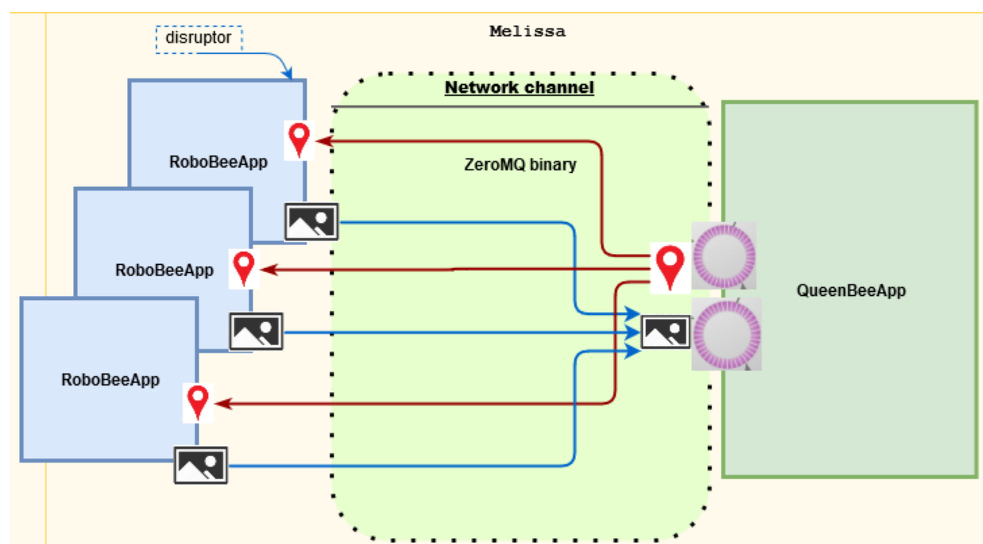


Fig 3. Melissa scheme with disruptors

To evaluate the system, it is based on different producer-consumer association scenarios, and it varies between three publication frequencies and two resolutions of captured images. The indicators used for the analysis are the subscription ratio of images and waypoints, show if events have been lost and the CPU and RAM used, all being measured by OpenMCT framework as we see in figure 4.

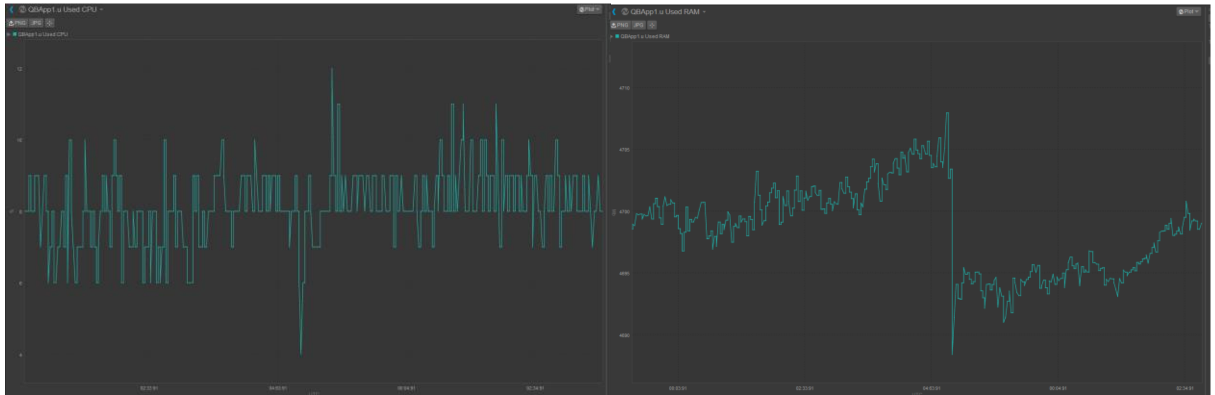


Fig 4. Stable CPU and RAM

Conclusions

In this regard we can conclude that, although a higher frequency of messages (essential for agile response systems) means a slight increase in processing, no event is lost and the memory consumed by the system remains constant.

Bibliography

Fowler, M. (2011) *The LMAX Arquitecture.* Recuperado (en febrero 2020) de https://martinfowler.com/articles/lmax.html

Ghiglino, P (2019). *API for application development in Klepsydra*. Recuperado (en diciembre 2019) de https://github.com/klepsydra-technologies/kpsr-core/blob/master/api-doc/api-kpsr-application.md

Hintjens, P. Sustrik, M, (2010) *Multithreading magic with 0MQ*.

http://zeromq.wdfiles.com/local--files/whitepapers%3Amultithreading-magic/imatix-multithreaded-magic.pdf

Hintjens, P. (2013) *ZeroMQ: Messaging for many applications.* O'Reilly.

Namiot, D. (2016) *On lock-free programming patterns*. Recuperado (en enero 2020) de https://www.researchgate.net/publication/295858544_On_lock-free_programming_patterns

Lafreniere, D. (2017) *C++ std::thread Event loop with message queue and timer.* Recuperado (en enero 2020) de https://www.codeproject.com/Articles/1169105/Cplusplus-std-thread-Event-Loop-with-Message-Queue

Anexo II. Glosario

Commit: Unidad que define, en la historia de un desarrollo, el conjunto de cambios en un repositorio de código.

Framework: Abstracción que provee funcionalidad genérica que el desarrollador convierte en lógica de negocio específica añadiendo código.

GPGPU: Computación de propósito general en unidad de procesamiento gráfico.

Memory leak: Fallo de programación donde se olvida liberar la memoria reservada, haciendo que el consumo crezca y haga fallar la aplicación por falta de esta.

Mutex: mecanismo de sincronización para controlar el acceso a un recurso compartido.

Overhead: consumo excesivo de recursos (tiempo, ancho de banda, memoria, cómputo) para realizar una tarea determinada.

Socket: Herramienta para enviar o recibir datos dentro de un nodo en una red.

Timestamp: marca de tiempo que registra cuando ocurrió un evento.

Waypoint: son coordenadas para ubicar puntos de referencia tridimensionales en la navegación basada en GPS

Wrapper: Abstracción de una clase a la que se le puede simplificar su uso y añadir estado o funcionalidad.