Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de la Matanza
ISSN: 2525-1333
Vol.: 7 - Nro. 2 (DICIEMBRE 2022)

Comunicación en Congreso

Informe técnico, telemetría satelital de tiempo real sobre websockets y framework Django

Technical report, realtime satellite telemetry over websocket and Django framework

Pablo Soligo, Jorge Salvador Ierache, Pablo Witold Martínez

Universidad Nacional de La Matanza,
Buenos Aires, Argentina

Resumen:

Se presenta los resultados preliminares obtenidos, en el uso de websockets sobre el framework para desarrollo web Django como solución a la visualización de telemetría de tiempo real en un prototipo experimental de software de segmento terreno multiplataforma-multimisión en el contexto del desarrollo de la estación terrena UNLaM. Se realizan pruebas de estrés para evaluar la factibilidad y los límites de aplicación para la tecnología actualmente utilizada como segmento terreno experimental en la UNLaM y su vinculación con el NASA OPENMCT [2].

Abstract:

ISSN: 2525-1333
Vol.: 7 - Nro. 2 (DICIEMBRE 2022)

Preliminary results are presented on the use of websockets over Django web framework as a solution for real-time telemetry visualization in an experimental prototype of multiplatform-multimission ground segment software in the context of the development of the UNLaM ground station. Stress tests are performed to evaluate the feasibility and application limits for the technology currently used as experimental ground segment at UNLaM and its integration with NASA OPENMCT [2].

Palabras Clave: Satélites, Segmento Terreno, Diseño de Software, websockets

Key Words: Satellites, Ground segment, Software Design, websockets

Colaboradores: Germán Merke, Pablo Ferreira, Santiago Mansfeld

I. CONTEXTO

La UNLaM (Universidad Nacional de La Matanza) posee un prototipo de segmento terreno experimental denominado UGS (UNLaM Ground Segment) [1]. Actualmente se puede visualizar, en la versión pública (https://ugs.unlam.edu.ar) telemetría satelital histórica de satélites en órbita procesada desde datos crudos provenientes principalmente de la red SatNogs [3] y otras fuentes alternativas. El Grupo de Investigación y Desarrollo Aeroespacial de la UNLaM (GIDSA) [4] se encuentra desarrollando su propia estación terrena para enlazar de manera directa con los satélites en órbita y como consecuencia de esto es necesario implementar mecanismos que permitan la visualización eficiente de telemetría en tiempo real. El UGS ha sido desarrollado como una plataforma de investigación, experimentación y capacitación, y tiene entre sus objetivos lograr que las soluciones en el área aeroespacial sean costo-efectivas y vinculadas con el estado del arte de la ingeniería de software, minimizando tanto como sea posible alternativas que impliquen desarrollos ad-hoc, herramientas de escasa penetración en la industria de software de propósito general o de costosa implementación. Desde su primera versión, el prototipo ha tenido como objetivo ser multimisión, pudiendo trabajar con satélites de organizaciones y fabricantes independientes.

II. INTRODUCCIÓN

Un sistema moderno de operación multimisión debe proveer de herramientas de explotación de datos que incluyan, entre otras, las siguientes características [5] y [6]:

- Habilidad de mantener de forma eficiente, segura y transparente los datos de la misión completa.
- Integración para cambiar entre datos históricos y de tiempo real.
- Una performance que permita obtener datos de años en unos pocos segundos.
- Manejar grandes volúmenes de datos y niveles de variabilidad y granularidad.
- Garantía de fiabilidad.
- Punto de acceso único disponible desde interfaces de programación de aplicaciones (API del inglés Application Programming Interface)

El UGS, si bien es un prototipo de investigación, ha tenido en cuenta estos requerimientos. En trabajos anteriores [7] se han explorado distintos diseños y herramientas de almacenamiento para la telemetría histórica. En este trabajo se analiza e implementa una alternativa para la comunicación en tiempo real entre las herramientas de visualización y la telemetría en tiempo real obtenida. Con la estación terrena operativa será necesario para el UGS tener la capacidad de visualizar telemetría en tiempo real sobre su capa de visualización, derivada del producto

ISSN: 2525-1333. Vol.: 7 - Nro. 2 (DICIEMBRE-2022)

de código abierto de la NASA Open MCT, estando el ***back-end*** desarrollado principalmente en Python Django. El uso de websockets no es necesariamente obligatorio, pero es la opción explícitamente desarrollada y puesta como ejemplo por el equipo de Open MCT [8]. Aunque Django no soporta nativamente el uso de websockets, sí dispone de bibliotecas para hacerlo. El presente trabajo presenta las experiencias en la adaptación del UGS y el Open MCT para visualización de telemetría en tiempo real usando la biblioteca Django Channels (https://channels.readthedocs.io/en/stable/).

Mediante una serie de pruebas de estrés se busca conocer la posible degradación en función de la cantidad de conexiones, obtener los límites de operación para estas tecnologías y analizar la factibilidad de usarlas masivamente en un sistema de segmento terreno multimisión en la nube.

III. MÉTODOS

Los websockets ofrecen una solución para establecer una conexión bidireccional entre en un servidor y un cliente sobre internet, especialmente bien soportada por los navegadores web. El UGS actualmente dispone de un servicio web que puede recibir telemetría de múltiples fuentes, tanto sea de una estación terrena, mediante adaptadores que consumen u obtienen datos crudos de internet, o directamente desde archivos compartidos por agencias espaciales vinculadas. Toda telemetría novedosa recibida deberá ser informada a los clientes que se suscriban. El cliente puede ser una instancia de Open MCT, la cual, según qué variables este visualizando, deberá subscribirse, anunciando su intención de ser informado de las novedades.

Para cumplir con este objetivo, teniendo en cuenta la tecnología y frameworks utilizados, sería suficiente con la alternativa disponible e integrada por defecto en el ORM (Object Relational Mapper) de Django. Django channels propone utilizar un decorador (@receiver) asociado a una señal denominada “post_save”. Esta señal se encuentra incorporada por defecto y puede, mediante retrollamada, informar sobre el salvado de un objeto, en nuestro caso una variable de telemetría. Sin embargo, y por cuestiones de eficiencia, el UGS no realiza el salvado de cada variable de forma individual, sino que realiza el guardado en una única transacción de la totalidad de variables que se encuentran en un paquete recibido, utilizando el método disponible en el ORM de Django ***bulk_create***. Por defecto ***bulk_create*** no enviara ninguna señal [9].

Con este escenario se plantearon modificaciones en los frameworks tendientes a optimizar el funcionamiento. Siguiendo las alternativas que ofrece el framework Django [10] se desarrolló un ***“manager”*** propio (TlmyVarManager) para la clase TlmyVar (variable de telemetría) y una señal especial indicando que se guardaron telemetrías, agregando como parámetro cuales fueron. Para minimizar el tiempo de comunicación entre el cliente y el servidor las novedades se informan antes de su

almacenamiento físico. Los archivos fuentes del prototipo donde se realizaron las modificaciones y las pruebas se encuentran disponibles en github [9].

Una vez recibida la notificación será informada mediante un ***message bróker*** (REDIS) [11] a los ***"AsyncWebsocketConsumer"***, entidades del servidor responsables de administrar las conexiones websocket con los clientes, y puntualmente de decidir si debe informar a los clientes la novedad en función de si están o no subscriptos a las variables que se notificaron. Las instancias de "***AsyncWebsocketConsumer***" serán responsables de guardar la lista de variables a las que están subscritas cada uno de los clientes. La Figura 1- Diagrama de alto nivel del proceso de difusión de novedades a los clientes conectadosse presenta el diagrama de alto nivel del proceso de difusión de novedades a los clientes conectados, donde se muestra el proceso de recibir novedades (desde uno o más canales), para posteriormente proceder a la serialización de la información para ser comunicada vía REDIS a los diferentes procesos encargados de atender a los clientes.

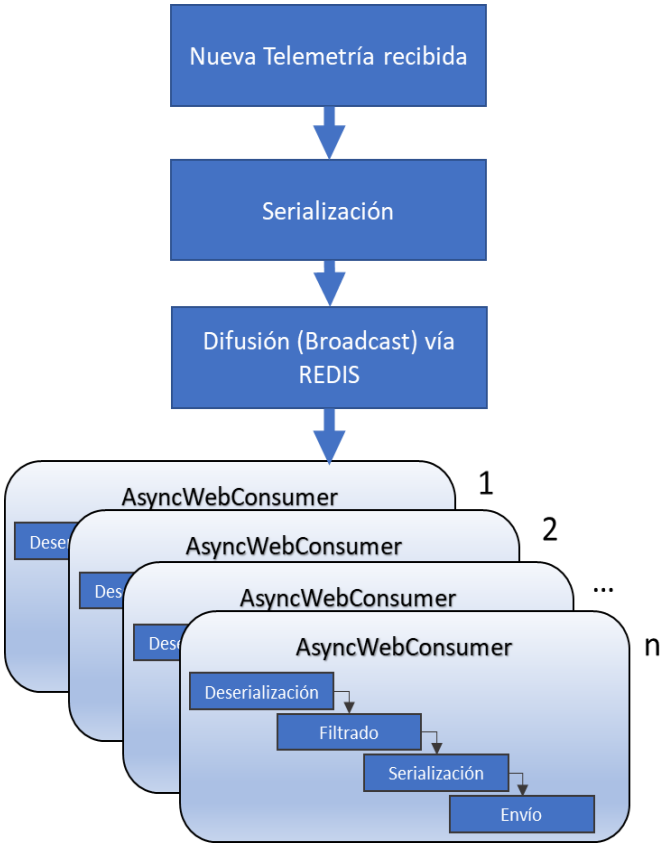

Figura 1- Diagrama de alto nivel del proceso de difusión de novedades a los clientes conectados

Las limitaciones que impone el GIL (Global Interpreter Lock) al uso de hilos en Python obligan a desarrollar procesos separados y a establecer una comunicación mediante datos serializados en lugar de compartir memoria hecho este que agrega un costo de procesamiento en la constante serialización y deserialización para el filtrado. La Tabla 1 - Pila de hardware y software detalla el hardware utilizado en la prueba, y ofrece además un detalle de las unidades de software que fue necesario instalar para simular un equipo de producción.

Hardware	
Procesador	AMD A10-7860K Radeon R7, 12 Compute Cores 4C+8G
Memoría	8 GB
HD 1	ScanDisk SSD PLUS *
HD 2	WDC WD6003FZBX-0
Software	
Servidor wsgi	Gunicorn
Servidor asgi	Uvicorn
Proxy reverso	Nginx
Base de datos	Postgresql 14
Otros	Redis

Tabla 1 - Pila de hardware y software

La cantidad de variables, 7500, se ha mantenido en relación con la cantidad esperable para un satélite de comunicaciones [5], la frecuencia de envió se ha definido en 10 segundos siendo este un número superior al tiempo necesario para el procesamiento y almacenamiento de las 7500 variables. El objetivo es que sin implementar corutinas o procesos distribuidos el servidor se encuentre en una situación de equilibrio antes de recibir las conexiones clientes sin encolados de ningún tipo. Con esta frecuencia, todos los paquetes pueden ingresar sin demoras, utilizando el hardware y software indicado en la Tabla 1 - Pila de hardware y software, incluyendo tiempos de transferencias http, autenticación y autorización, siendo el productor de los paquetes un solo proceso. Los clientes son simulados utilizando la biblioteca *websockets* [12] de Python. Para las pruebas se crearán clientes y se medirán los tiempos entre la llegada del paquete al servidor y la recepción de la novedad por parte del cliente.

Cada cliente suscribirá 30 variables aleatorias suponiendo que es la cantidad de variables que pueden observase de manera simultánea en una pantalla. Para simplificar la prueba los clientes mantienen vivas el total de las subscripciones. Se mide el promedio de los tiempos entre que la variable llega al servidor ***back-end*** (el paquete crudo es procesado) y que la novedad llega al cliente. Las variables de telemetría viajan con una marca de tiempo que permite calcula la diferencia y los tiempos de los equipos son sincronizados utilizando comandos para tal fin (timesyncd).

IV. RESULTADOS

En la Figura 2- Tiempos de retardo promedio según cantidad de conexiones clientes se muestran los tiempos de retardo promedio según cantidad de conexiones para 5, 25, 125 y 625 clientes con 30 variables subscriptas cada uno. Se han realizado las pruebas ejecutando el servidor ASGI (Asynchronous Server Gateway Interface) con 4 y 8 workers, o procesos paralelos. La Tabla 2 - Tiempos de retardo promedio según cantidad de conexiones

clientespresenta los tiempos de retardo promedio según cantidad de conexiones clientes.

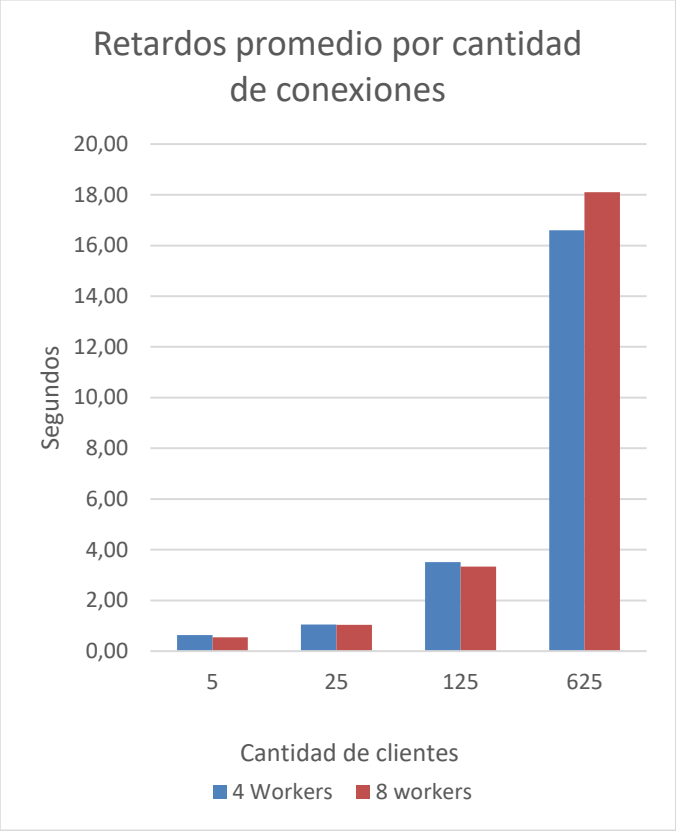

Figura 2- Tiempos de retardo promedio según cantidad de conexiones clientes

Clientes	Workers	
	4	8
5	0,63s	0,55s
25	1,05s	1,04s
125	3,51s	3,33s
625	16,59s	18,10s

Tabla 2 - Tiempos de retardo promedio según cantidad de conexiones clientes

VI. CONCLUSIONES

La implementación actual no ofrece rendimientos que permitan pensar en un sistema en la nube que pueda escalar utilizando hardware de bajo costo.

Durante las pruebas se detectaron inicialmente demoras importantes, al iniciar una serie de mediciones utilizando "cProfile" [13] se observó que los principales problemas radican la serialización, deserialización y filtrado de novedades correspondientes a cada cliente. Cada cliente subscribe a un subconjunto de variables y es necesario dividir que novedad es aplicable a cada cliente siendo inicialmente un problema de N*S*C, N novedades contra S variables subscriptas y eso para cada cliente C, en nuestras pruebas N=7500, S=30 y 5<C<625. Se debe observar que la información debe viajar serializada a cada instancia de ***"AsyncWebsocketConsumer"*** que puede ser ejecutado en un proceso separado.

Para atender a este problema se ha trabajado con el tipo integrado ***"set()"*** que demostró mejor rendimiento al momento de determinar la intersección entre las novedades recibidas y las que deben ser informadas a cada cliente. La naturaleza asincrónica de la clase ***"AsyncWebsocketConsumer"*** solo aplica al envió del paquete mediante sockets una vez este conformado.

ISSN: 2525-1333. Vol.: 7 - Nro. 2 (DICIEMBRE-2022)

A pesar de todas las optimizaciones aplicadas se observan retardos importantes cuando la cantidad de clientes crece. Los tiempos de deserialización, filtrado y serialización no son despreciables incluso aplicando optimizaciones y eligiendo los tipos integrados que mejor se comporten para estas operaciones. Estos procesos se deben realizar para cada cliente y haciéndolo especialmente sensible cuando tiene que atender a un alto número de conexiones.

La limitación del intérprete de Python para trabajar con múltiples hilos o procesos livianos y compartir memoria obliga a crear ***"workers"*** o procesos paralelos donde la información tiene que ser transmitida de forma serializada. Posteriormente debe ser deserializada para separar que información se debe comunicar a que cliente. Si bien cada ***"worker"*** puede deserializar, filtrar, serializar y enviar simultáneamente, cuando la cantidad de clientes supera las capacidades de paralelización del hardware se producen encolados.

VII. TRABAJO A FUTURO

Como trabajo a futuro se intentará aplicar optimizaciones y diseños alternativos. Las limitaciones del GIL impiden pensar en soluciones que eviten la serialización y deserialización en cada proceso, no parece posible salvar esa limitación y por tanto se pueden pensar alternativas que reduzcan tanto como sea posible los conjuntos a compartir entre procesos. Una primera alternativa implica enviar a los procesos o ***"workers"*** solo las variables que hayan sufrido cambios y/o enviar a los procesos o ***"workers"*** solo las variables que al menos estén subscriptas a algún cliente. Ambas alternativas requieren un preproceso, pero reducen los conjuntos que deben deserializar y filtrar en cada proceso. Estas opciones deben ser probadas para verificar que genera beneficios, aunque complejizan la implementación.

Una segunda solución de sencilla implementación implica generar una señal individual por cada variable Luego cada cliente decidirá si le pertenece o no esa variable puntual. Esta opción reduce el tamaño de los datos a serializar y los filtrados traduciendo la mayor carga de trabajo al ***"message broker"*** (REDIS). Adicionalmente también se puede aplicar un prefiltros para asegurarse que la variable esta subscripta a al menos un cliente.

Una tercera solución posible es enviar todas las novedades al cliente si está subscripto al satélite en lugar de estar subscripto individualmente a una variable, esta última alternativa puede presentar reparos en cuanto a la seguridad y confidencialidad de los datos y, por lo tanto, si bien es de muy sencilla implementación y puede ser aplicable en entornos de redes cerradas no resulta aceptable para soluciones en la nube. Finalmente mayores capacidades de hardware y procesos desacoplados de la aplicación del centro de control, particularmente serialización y desrealización con cada satélite pueden brindar

soluciones tentadoras de explorar para superar las limitaciones indicadas.

VIII. REFERENCIAS Y BIBLIOGRAFÍA

[1] https://ugs.unlam.edu.ar, *UNLaM Ground Segment,* 2020.

[2] J. Trimble y A. Henry, «Building a Community of Open Source Contributors,» de *International Conference on Space Operations (SpaceOps 2018)*, 2018.

[3] D. J. White, I. Giannelos, A. Zissimatos, E. Kosmas, D. Papadeas, P. Papadeas, M. Papamathaiou, N. Roussos, V. Tsiligiannis y I. Charitopoulos, «SatNOGS: satellite networked open ground station,» 2015.

[4] https://gidsa.unlam.edu.ar, *Grupo de Investigación y Desarrollo de Software Aeroespacial de la Universidad Nacional de La Matanza,* 2020.

[5] T. Morel, G. Garcia, M. Palsson y J. C. Gil, «High Performance Telemetry Archiving and Trending for Satellite Control Centers,» de *SpaceOps 2010 Conference Delivering on the Dream Hosted by NASA Marshall Space Flight Center and Organized by AIAA,* 2010.

[6] G. Pace, M. Schick, A. Colapicchioni, A. Cuomo y U. Voges, «EO ON-LINE DATA ACCESS IN THE BIG DATA ERA,» de *Proceedings of the 2019 conference on Big Data from Space,* 2019.

[7] P. Soligo, J. S. Ierache y G. Merkel, «Telemetría de altas prestaciones sobre base de datos de serie de tiempos,» 2020.

[8] «OPENMCT Getting Started,» [En línea]. Available: https://github.com/nasa/openmct-tutorial. [Último acceso: 5 12 2022].

[9] The Django Software, «https://docs.djangoproject.com/en/4.1/ref/models/querysets/,» [En línea]. Available: https://docs.djangoproject.com/en/4.1/ref/models/querysets/. [Último acceso: 24 10 2022].

[10] GIDSA, «UGS Backend,» [En línea]. Available: https://github.com/unlamgidsa/unlam_gs_backend.git. [Último acceso: 5 12 2022].

[11] Django Software Foundation, «https://docs.djangoproject.com/en/4.0/topics/db/managers/,» [En línea]. Available: https://docs.djangoproject.com/en/4.0/topics/db/managers/. [Último acceso: 24 10 2022].

[12] «Redis,» [En línea]. Available: https://redis.com/solutions/use-cases/messaging/. [Último acceso: 5 12 2022].

[13] websockets-the-project, «websockets,» [En línea]. Available: https://websockets.readthedocs.io/en/stable/. [Último acceso: 24 10 2022].

Recibido: 2022-12-20

Revista Digital del Departamento de
Ingeniería e Investigaciones
Tecnológicas de la Universidad
Nacional de la Matanza
ISSN: 2525-1333. Vol.: 7 - Nro. 2 (DICIEMBRE-2022)

Aprobado: 2022-12-28
Hipervínculo Permanente: https://doi.org/10.54789/reddi.7.2.5
Datos de edición: Vol. 7 - Nro. 2 -Art. 5
Fecha de edición: 2022-12-29