GODOTflow: an actor-based toolkit to support near real-time flight dynamics operations

Marco Lombardo[(1)], Ruaraidh Mackenzie[(2)], Andrea Sesta[(3)], Rory Tyrell[(2)], Marco Zannoni[(1)]

[(1)] Department of Industrial Engineering, University of Bologna
Forlì, Italy
Email: marco.lombardo14@unibo.it

[(2)] European Space Operations Centre (ESOC)
Darmstadt, Germany
Email: ruaraidh.mackenzie@esa.int

[(3)] Sapienza University of Rome
Rome, Italy
Email: andrea.sesta@uniroma1.it

Abstract – Flight dynamics operations of unmanned space missions have always required an important effort in terms of activities and procedures organization, teams' coordination, and human resources. In particular for those missions that require high precision, robustness, and efficiency to reach the objectives and ensure the safety of the spacecraft.
Typically, a flight dynamics team must monitor and control a spacecraft's orbit and attitude using data from a limited time span, while ensuring rapid response to critical events such as collision avoidance, tumbling recovery, deep space maneuvers or a close flyby of a celestial object. For example, during the Launch and Early Orbit Phase (LEOP) the flight dynamics team may be required to provide a report on the orbit and attitude status in a very short time using data that may still be arriving in real-time from the ground station. All the aspects described previously can become even more critical in the case of flight dynamics operations for satellites constellations, deep space small satellites, or flagship scientific missions like Cassini or Juice. To reduce the risks of operational and human errors, multiple dedicated software tools and a large flight dynamics team are usually employed, which leads to an increase in the complexity and costs of satellite operations. In addition to the use of accurate software and robust navigation strategies, a potential mitigation for the previous problems could be the use of an automated flight dynamics system for Near Real-Time (NRT) operations to be used in support to the navigators.
The GODOTflow project is aiming for this latter goal with the development of a software infrastructure, based on ESA/ESOC's GODOT and coded in Python, that would allow the user to perform different types of automated NRT flight dynamics tasks. Conceptually, data in GODOTflow is continuously transmitted and managed within the system such that any new received information passes through each parallel or subsequent process, depending on the user's chosen configuration. In this way the configured flight dynamics system reacts autonomously to any new information by providing constant updates about the tracked quantities and parameters. To perform the described activities in a scalable and extensible way, the actor model programming has been adopted as it allows for high scalability, fault tolerance, and responsiveness. In this programming technique, actors are identified as independent orchestrated entities that can communicate with each other by sending and receiving messages, without sharing any memory or state. The flight dynamics processes (actors) at the core of GODOTflow will include activities such as acquiring and storing real data from different sources (e.g. acquiring radiometric observations from the shared endpoint of a ground station), simulation of flight dynamics data (e.g. simulation of real-time LEOP tracking data), orbit determination, station keeping, as well as storage, stream, and visualization of the obtained results.

The strategy adopted in GODOTflow would help to increase the efficiency of the flight dynamics process by improving its scalability and organization, while reducing non-critical team workload and human errors.

I. INTRODUCTION

Flight dynamics operations of a spacecraft (S/C) are a complex and crucial aspect of space exploration. It involves the study and management of the movement and control of a S/C in outer space. This process is crucial for the success of any space mission, as it ensures the safe and efficient operation of the S/C.
These operations usually involve different kind of activities and tasks like pre-processing telemetry and observations, perform the Orbit Determination (OD), evaluate the trajectory uncertainties at a certain time, computing a correction maneuver. Many of these tasks are usually performed by different groups of people using dedicated tools which may not be in harmony with each other and may require additional procedures and interfaces.
Furthermore, due to the complexity of the organization of the described activities, the data flow within the flight dynamics process can be slow, and the entire navigation activity may be less responsive to the critical events. To deal with these critical issues, a large navigation team is

usually employed, which leads to further complexity in organization and procedures as well as an increase in the costs of satellite operations. However, in situations where high responsiveness from the navigation team is required, such as during a Launch and Early Orbit Phase (LEOP) or during a critical maneuver (e.g. orbit insertion maneuver), the described navigation system may require a considerable organizational and personnel effort with a consequent increase in the risk of operational and human errors. Nonetheless, scalability is also drastically affected since requiring, for example, the same high responsiveness for multiple S/C as in a low earth orbit (LEO) constellation would simply exponentially increase the described costs and efforts.
In that sense, performing flight dynamics operations for one or multiple S/C with a Near Real-Time (NRT) awareness requirement may be challenging or difficulty achievable without a proper automation and software system.
There are some examples of automated applications for flight dynamics operations, such as [1] and [2], mainly for Earth missions or constellations where multiple satellites are flown by a single flight dynamics team. However, these applications primarily use proprietary software usually compiled in Java and C++. The use of these programming languages offers reliability and robustness on the one hand, but less flexibility and extensibility on the other.
For example, using an interpreted programming language like Python allows to write software efficiently and quickly, ensuring high extensibility and adaptability to many types of problems. State-of-the-art Python-based navigation software with a C++ core layer, such as ESA's GODOT [2] and NASA/JPL's MONTE [3], have demonstrated great reliability in navigating near Earth and deep space missions, as well as high extensibility and adaptability to different flight dynamics problems.
The GODOTflow project fits into this context with the aim of proposing a highly scalable and automated software infrastructure, coded in Python and based on ESA's GODOT, for NRT flight dynamics operations, with a focus on the S/C navigation. This paper describes the concept and implementation of GODOTflow by proposing also a potential use case with a simulated example.

II. CONCEPT

The main purpose of the activity was to prototype a NRT astrodynamics data processing system based on GODOT which can interface with modern S/C operations applications, e.g. EGS-CC [4], and be used by other actors to process data for operations. The current vision of GODOTflow is to have an infrastructure that would allow the user to perform the following operations in NRT:

- Simulate data or get real data (e.g., radiometric observations) from other sources.
- Process them through a set of configurable activities like an OD process.
- Store the results into a database or stream them to other processes.
- Postprocessing results to perform evaluations such as computing orbital events, propagate the OD covariance, compute an orbital maneuver.
- Visualizing the results with modern web-based user interfaces.

Based on this vision, the GODOTflow project has been inspired by computational models for data streaming and data flow management, in particular the actor model paradigm [5]. As a result, GODOTflow abstracts each flight dynamics operation as a single independent execution unit, called *actor*, that receives input and generates output by reacting to the incoming data stream asynchronously.
The actors have their own internal state, they can be created, destroyed and exchanged dynamically during the execution of the program. From a functional point of view, the actor is a simple interface that is easy to be understood, developed and debugged. This model enables functional decomposition that allows the user to optimize and scale actors independently, which is not possible with monolithic components. Furthermore, multiple actors in parallel provide an excellent fault tolerance since the faulty actor can be easily identified and circumscribed without breaking the entire application. In that sense, GODOTflow exploits the actors in such a way that what becomes fundamental for the application's context is the data flow. A single GODOTflow application has been named "flow" because it is composed by actors that through their interaction creates a flow of information from the originator up to the very the last recipient.
The data is handled by using three main characterizations: static data, periodically updated files and procedural data. The static data refers to such data that is rarely updated like the planetary ephemerides or the ground stations database. The periodically updated files could be the atmospheric calibrations, or the observations received in batches from a ground station. Finally, the procedural data is the one that is generated in the flow during the NRT activity. For example, procedural data can be the residuals of the OD process, or the observations if they are flowing from a generator that is an actor of the flow (e.g. radiometric data simulator) or an external service attached to an input of the flow (e.g. ground station).
To efficiently manage the actors and flow execution an orchestrator is required. The role of the orchestrator is mainly to coordinate the interaction between the actors, manage the creation, termination and communication between them and monitor their status to resolve any potential conflict or problem. The GODOTflow orchestrator is then in charge of defining the order and execution logic of the defined actors, distributing the

actors across different threads to enable parallel execution, and managing resource allocation.

III. Implementation

The implementation of GODOTflow has been performed by using only open-source libraries and tools. Currently, because of the lack of a comprehensive actor system for Python, the implementation of the GODOTflow system prototype started with a core Python's library that contains the following elements: the actor model infrastructure, the application (flow) interface, a set of GODOT models and pre-defined actors to perform OD and radiometric data simulation, and a web-based metric visualizer (e.g. visualization of covariance propagation, residuals, observations). Furthermore, since GODOT does not currently support serialization of all constituent classes and quantities (e.g. the universe or partial derivatives are not serializable), it was necessary to add additional models and interfaces to the GODOTflow Python's library to temporarily circumvent this limitation that affects tasks like the stream or the storage of an OD solution.

The implemented basic actor interface is characterized by *receive* and *send* methods used to exchange data between actors, and a *process* method that represents the task that the actor will perform each time a new message is received. The flow interface is characterized by a *compile* method that takes a blueprint to create the connections between the actors as well as start them in the right order. The entire actor and flow configuration are then placed into a single Python script sent to the orchestrator for deployment.

Despite its high flexibility and ease of use, Python code runs on a Python interpreter that, through a system called Global Interpreter Lock (GIL), does not allow thread concurrency due to the risk of race conditions and memory corruption for concurrent object access.

To circumvent this limitation and guarantee parallelism, high scalability, and efficient orchestration, the Ray framework [6] has been used. Ray is a powerful unified computing tool for Python-based parallel programming, distribution of applications, and algorithms optimization. It natively supports the actor model paradigm, and it offers the possibility of orchestrating the deployed actors, control their status and manage the resources. Moreover, Ray can be deployed on a Kubernetes [7] cluster allowing for an operative ready environment that is completely scalable.

For GODOTflow, the use of a Ray cluster deployed on Kubernetes was adopted with the aim of having a test environment as close as possible to an operational case.

To manage the exchange of data between the actors the Apache Kafka [8] open-source data streaming platform has been used. Kafka runs on a dedicated service on the Kubernetes cluster, and it optimizes the transmission of the data through a distributed architecture of producer and consumers. This transmission system, that is named *topic*, can be thought as a pipe into which information is poured by producers and then downloaded by consumers at any stage. In the GODOTflow case, topics were only used for one-to-one communication between actors.

To enable a permanent results storage and easy access by services and users outside the cluster, the NoSQL [10] MongoDB [11] database system was used. The latter database allows for easy integration with Python and provides high availability, flexibility, horizontal scalability, and support for many other programming languages as well. Furthermore, given its nature as a document database, integration with the existing JSON/YAML file format supported by GODOT and GODOTflow was straightforward and allows for efficient storage of both observations and results, as well as configuration files for universe, trajectories, OD filter and *a-priori* covariance definition.

For the metrics visualization, a Python web-based service has been developed by using Streamlit [12], a modern Python library capable of turning Python scripts into shareable web apps. Thanks to this feature, the GODOTflow visualizer system takes advantage of modern web-based technologies without renounce the use of Python for an efficient integration with the rest of the software. Currently, the visualizer service is intended to run on a different host (outside the Kubernetes cluster) so that the resources required for multiple users to log in simultaneously are not taken away from those required by the actors of the flows. To obtain information from the flow, the viewer can interact with a designated actor via Python FastAPI [13] or by querying data stored in the MongoDB databases.

Finally, the GODOTflow library provides a set of pre-defined actors to perform several standard OD activities, in particular:

- *ObservationsSimulator*: an actor that calculates and transmits radiometric observations as if they came directly from a ground station in the form of data batches (e.g. data per tracking pass or five observations every five second).
- *OrbitDeterminationProcess*: an actor that receives the observations as input, performs the OD process by calculating the residuals and executing a least square filter, and transmits the computed residuals and the solution as output. This actor can also store the results to the MongoDB database.
- *StateProvider*: a post-processing actor that receives an OD solution as input and performs on-demand evaluation of useful quantities, such as covariance propagation and estimated state, via remote Ray calls or FastAPI requests. This is an example of an actor that can be designated for the visualizer activities.

A graphical representation of the described GODOTflow implementation is reported in Fig. 1. By looking at the figure it can be see that the three-actor flow application runs as a Ray job on KubeRay using the functionalities of Kafka and MongoDB that are hosted as separated nodes on the Kubernetes cluster. The persistent data is stored in a shared filesystem mount or uploaded directly with the job. The exchange of results with the external visualizers or users (dashed lines) is made through FastAPI with HTTP connections, or with the Python driver in the case of an access to MongoDB.

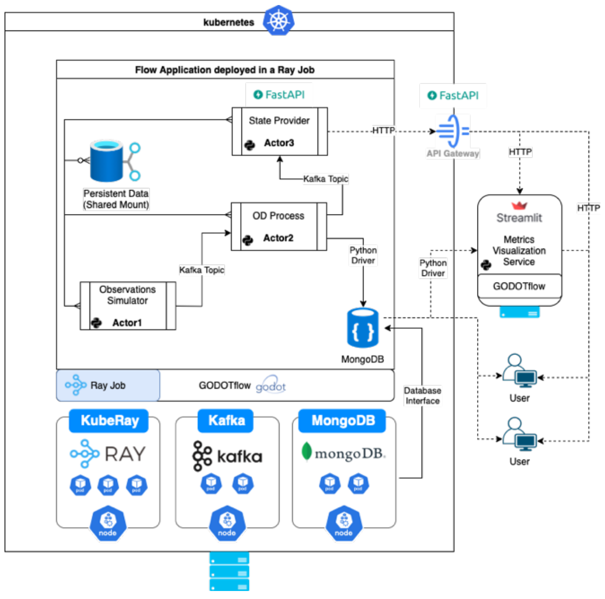


Fig. 1. Current GODOTflow implementation schema.

IV. USE CASE EXAMPLE

To test and analyze the entire GODOTflow system, an example use case was studied. The use case considers the NRT OD of a S/C that is orbiting the Earth in a Sun-synchronous polar orbit at 800 km of altitude. The S/C is tracked by the 4.5 m New Norcia 2 (NNO-2) antenna [14] which provides two-way Doppler observations in X-band with 1 s count time. The S/C performs also a test orbital maneuver of 5 m/s, one day after the initial epoch, in the direction of the orbital velocity vector. To simulate this use case, a four-actor GODOTflow application was implemented. The application was then deployed to the KubeRay cluster.
First, an observations simulator actor mimics the NNO-2 station by initially propagating the S/C trajectory and simulating Doppler observations in each viewing period computed in the selected time interval. The simulated observations are then corrupted with white Gaussian noise (with a standard deviation of 0.1 mm/s) and streamed in batches of 5 observations every 5 seconds.
The simulator actor is then connected to two OD process actors, one (named "passthrough") for directly computing the residuals (with only the *a-priori* dynamic model) and the other (named "estimator") for estimating the trajectory using a least squares filter. Both the OD actors store the residuals in the MongoDB database while the actor that is doing the estimation is also storing the obtained solution and the configuration files. The estimator actor estimates the S/C initial state and the test orbital maneuver (impulsive model) with the filter configuration provided in Tab. 1.
The OD estimator actor is also connected to a state provider actor to which it streams the obtained solution. The state provider actor replicates the OD solution in its internal memory and uses it to perform the required post-processing calculations via a Ray remote call or a FastAPI request. The implemented state provider actor allows to evaluate the S/C state and covariance in the required epochs and reference systems.

Table 1. Filter setup of the OD estimator actor.

Parameter	**Component**	***A-priori* Sigma**
S/C epoch state	Position Velocity	100 m 1 m/s
Test Maneuver	Delta-V Right ascension Declination	10% of nominal 0.1 deg 0.1 deg
Doppler Weight	N/A	2e-3 Hz (0.1 mm/s @ X-band)

A visualization system runs as a web service on a different host than the one where KubeRay is running on and provides two web pages for viewing OD results: state covariance viewer and residuals viewer. The state covariance viewer enables the plotting of the S/C state's uncertainty evolution by configuring the center (e.g. Earth), reference frame, coordinate system (e.g. Cartesian or Keplerian), and dates to which propagate the covariance. With the input that the user has provided, the covariance viewer sends a request to the designated state provider actor through the FastAPI interface. This information is updated as soon as a new solution is received by the state provider. Fig. 2 shows the covariance viewer page displaying the evolution of the S/C cartesian position uncertainty in the Earth-centered International Celestial Reference Frame (ICRF).

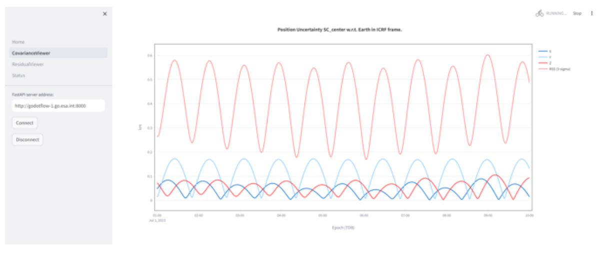
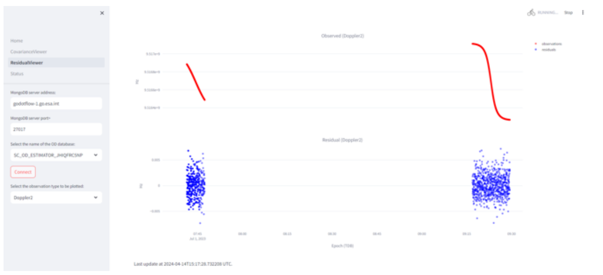


Fig. 2. S/C position uncertainty evolution (cartesian coordinates, with respect to the Earth in the ICRF frame) displayed in the web-based visualizer service. The red line represents the 3-sigma Root Sum of Squares (RSS) of the position uncertainties X, Y and Z.

The visualization of the residuals is performed by the residuals viewer tool that queries the databases of the OD actors within the MongoDB instance. The residuals stored in the databases are updated by the OD actors every time a new batch of data is processed. Therefore, the residual viewer page automatically shows the data as it is being processed and stored. Fig. 3 shows the residuals viewer page as the simulated two-way Doppler data comes through the flow application.



MongoDB server address:
godotflow-1.go.esa.int
MongoDB server port:
27017
Select the name of the OD database:
SC_OD_ESTIMATOR_JHIQFRCSNP
Select the observation type to be plotted:
Doppler2

Last update at 2024-04-14T15:17:28.732208 UTC.

Fig. 3. The simulated two-way Doppler observations and OD residuals plotted by the web-based visualizer as the data is processed by the OD actors.

The simulation was carried out with batches of observations of different sizes (e.g. 5 observations per batch, 30 observations per batch, one tracking pass per batch) and transmitted with different frequencies to test and analyze the responsiveness of the system as well as the backpressure on the OD actors. The results showed that, in the NRT case where the batch size is relatively small, the passthrough actor can process the data batch as soon as it is received without slowing down the output rate. This is mainly due to the fact that the passthrough actor computes only the residual, as observed minus computed, for each observation, without re-running filtering or trajectory propagation. On the other hand, the output ate of the estimator actor is influenced by the very nature of the least squares filter which requires multiple iterations and trajectory propagations also using all previous data. This introduces a backpressure on the estimator which tends to generate output with lower frequency the more data is consolidated and processed. In any case, for a few tracking passes the effect of the backpressure is negligible from an operational point of view. In the analyzed case, after four tacking passes, the delay in the estimator output was approximately one minute compared to the date of receipt of the most recent batch of data.
One possible solution to the backpressure of the estimator would be to consolidate an OD solution after each tracking pass (or at selected delivery epochs) so that the actor does not have to reprocess all previous data. Additionally, using a Kalman filter instead of a least squares filter would allow NRT processing of the data without reprocessing all the previous data every time a new batch is received.
The performance of the results visualization service benefits greatly from the employed web-based technology and application-independent execution. Therefore, the visualization is generally smooth while the times depend on the connection speed of the client (e.g. viewing from a laptop via virtual private network on a mobile connection may be slower than a direct wired connection). Thanks to the scalability of the GODOTflow system, the additional load from an increase in clients requests would be handled, for example, by deploying more state actor providers.

V. Conclusion

GODOTflow is a prototype NRT flight dynamics system based on Python and GODOT. The work carried out so far has focused on the development of the architecture based on the actor model and has explored the different existing technologies for implementing this system in Python. The impossibility of serializing certain GODOT objects and the calculated partial derivatives required the development of additional features to be able to distribute certain operations across different processes.
The study demonstrated that the use of GODOT, Ray, Apache Kafka, MongoDB and Kubernetes can be an optimal approach to implement the proposed conceptual model and combine scalability, reliability, and flexibility. The adopted automation strategy would also help to reduce human errors and the non-critical team workload. However, further analysis is needed to carefully evaluate performance, reliability and scalability by studying other cases such as that of a satellites constellation or a deep space mission.
Future work will include, but is not limited to, implementing a Kalman filter version of the OD actors, improving the visualization service by adding other useful flight dynamics parameters (e.g. comparison of reference trajectories, uncertainty on station view periods, 3D trajectory display) and implement a station-keeping actor that automatically proposes a series of correction maneuvers to follow a reference trajectory.
Furthermore, a future test with a real near-Earth mission (e.g. in a LEOP phase) is also planned.

VI. References

[1] Y. T. Yoon, P. Ghezzo, C. Hervieu, and I. D. A. Palomo, "Navigating a large satellite constellation in the new space era: An operational perspective", *Journal of Space Safety Engineering*, Volume 10, Issue 4, pp. 531-537, 2023.

[2] X. Marc, and M. A. Garcìa Matatoros, "Automated Flight Dynamics Support to Earth Observations Operations at ESA / ESOC", *American Institute of Aeronautics and Astronautics*, SpaceOps Conference, AIAA 2008-3224, 2008.

[3] "GODOT (General Orbit Determination and

Optimsiation Toolkit),” GODOT Documentation, https://godot.io.esa.int/godotpy/index.html (accessed Apr. 10, 2024).

[4] S. Evans, W. Taber, T. Drain, J. Smith, H. C. Wu, M. Guevara, et al., “MONTE: the next generation of mission design and navigation software”, *CEAS Space Journal 10*: 79-86, 2018.

[5] M. Pecchioli, A. Walsh, “The EGS-CC based Mission Control Infrastructure at ESOC”, *Workshop on Simulation for European Space Programmes (SESP) Conference*, 2017.

[6] G. Agha, “Actors: a model of concurrent computation in distributed systems”, *MIT press*, 1986.

[7] P. Moritz, R- Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, et al., “Ray: A distributed framework for emerging {AI} applications”, *13th USENIX symposium on operating systems design and implementation*, OSDI 18, pp. 561-577, 2018.

[8] D. Rensin, “Kubernetes”, *O'Reilly Media*, Incorporated, 2015.

[9] N. Garg, “Apache kafka”, *Birmingham, UK: Packt Publishing*, 2013.

[10] C. Strauch, U. L. S. Sites, and W. Kriha, “NoSQL databases”, *Lecture Notes, Stuttgart Media University*, 20(24), 79, 2011.

[11] A. Chauhan, “A review on various aspects of mongodb databases”, *International Journal of Engineering Research & Technology (IJERT)*, 8.05: 90-92, 2019.

[12] T. Richards, “Getting Started with Streamlit for Data Science: Create and deploy Streamlit web applications from scratch in Python”, *Packt Publishing Ltd*, 2021.

[13] M. Lathkar, “High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python”, *Apress*, 2023.

[14] R. Martin and M. Warhaut, “ESA's 35-meter Deep Space Antennas at New Norcia/Western Australia and Cebreros/Spain”, *2004 IEEE Aerospace Conference Proceedings* (IEEE Cat. No.04TH8720), Big Sky, MT, USA, 2004.