18th International Conference on Space Operations, Montreal, Canada, 26 - 30 May 2025.

SpaceOps-2025, ID # 496

A flight software framework for providing safe, coordinated access to satellite resources

William Bergen[a]

[a] *Lynk Global, Falls Church, Virginia, United States of America, wbergen@lynk.world*

Abstract

Satellites have several on-board resources (sensors, actuators, powered components, processors, etc.) that need to be coordinated in different ways to perform various activities. As mission needs grow more complex and autonomous operations become more prevalent, the likelihood of competing processes requiring the same resource increases. This paper describes a flight software framework that allows separate processes to safely access satellite resources in a coordinated manner. It accomplishes this by enforcing a set of ownership rules on a set of configured resources. These can be acquired by different tasks in a shared or exclusive manner. In doing so, the framework prevents multiple owners of exclusive resources and preserves an ownership list of shared resources, only cleaning up the resource when the list is exhausted. Additionally, resources could be extended to be dependent on other resources, creating Directed Acyclic Graphs (DAGs) of connected resources. When a leaf node is acquired, all dependent resources up until the root will also be acquired.

Operationally, utilizing this framework makes it impossible for autonomous tasks to put the satellite into invalid or unintended configurations. For example, a process that takes exclusive ownership of an optical payload will prevent any other processes from accessing the same hardware. In a naive system, these competing processes could easily apply conflicting settings to the payload, resulting in misconfiguration and performance degradation. Furthermore, the dependency mechanism has the added benefit of abstracting specific hardware implementations across multiple satellite revisions. For instance, the same "payload" might be constructed with different components on different satellites. Functionally, a client is concerned with utilizing the "payload" and not necessarily the components required to access it. Simplifying acquisitions in this way reduces the logic required of client applications, which is especially beneficial to large constellations where hardware changes rapidly and maintaining complex logic is expensive.

Keywords: Flight Software, Framework, Automation, Safety

Acronyms/Abbreviations
DAG: Directed Acyclic Graph
IPC: Inter-process Communication
LEO: Low Earth Orbit
MNO: Mobile Network Operator
Protobuf: Protocol Buffer
SDR: Software Defined Radio
ZMQ: ZeroMQ

1. Introduction

The purpose of the satellite flight software is to ensure the mission gets executed in a correct, timely, and safe manner. Typically, low-level interfaces are provided for controlling power to components and subsystems on a satellite. This is likely sufficient in a more traditional operations environment where components and subsystems can be manually controlled to fulfil the demands of a relatively static mission. However, as constellations grow, complexity increases, and autonomous operations become more prevalent, it becomes more difficult to ensure that independent tasks are accessing the satellite resources in a consistent, and safe manner. For example, two different tasks might require access to an imaging payload composed of several individual components. If the tasks utilize low-level interfaces to achieve their purpose, there might be nothing in place to prevent them from utilizing the payload in an invalid configuration due to their potentially conflicting requirements. This places the burden of valid and correct usage on the autonomy implementation (such as a constellation activity scheduler) and does nothing to prevent invalid configurations from occurring on the satellite. Thus, it becomes necessary for the flight software to implement a mechanism that will allow for safe, coordinated access to onboard resources. Hereafter, the term "resource" will be used to describe a component, set of components, or subsystem that needs to be utilized by some onboard task or client.

Lynk Global is designing, building, and operating a Low Earth Orbit (LEO) constellation of satellites providing direct-to-device mobile network connectivity at a global scale. Currently, we operate a fleet of 5 satellites that provide this service to our partner Mobile Network Operators (MNOs) across the globe, with plans to launch thousands more satellites in the coming years. With a lean and agile engineering philosophy, we embraced automation early to achieve high utilization of our payload. In doing so, there were issues using low-level interfaces to access satellite resources during complex operations like simultaneously executing an attitude maneuver, transmitting through the primary payload for a cellular overpass, collecting spectrum data for analysis, and running a ground station contact. By default, most resources are powered off for nominal operations, thus, tasks must power on and off the various resources they require to achieve their goal. Since most of these tasks were developed as individual task flows, it became unclear which tasks should be deciding when to turn on and off shared resources. Additionally, a resource might be designed for exclusive use by a task, and competing usage requests could lead to invalid configuration of said resource. One approach was to develop monolithic tasks to execute a specific combination of activities, however, this is inflexible and would require individual implementations for every operational use-case. Another approach would be to implement the coordination as part of the task scheduling automation; however, this has the problem of not actually preventing invalid usage since it would be performed as part of the planning process and not enforced during task execution. Both options are further complicated by the fact that tasks might have different resource acquisition logic and timing across satellite generations. Ultimately, we ended up creating a flight software framework to provide safe, coordinated access to resources by multiple independent tasks.

2. Initial Software Requirements

There are two primary issues that need to be solved by the proposed software framework. The first is coordinating shared access to resources like a network switch, payload computer, voltage regulator, etc. Various tasks might need these types of resources for different amounts of time. It is important to note that it is completely valid for both tasks to simultaneously use a shared resource in this scenario. Figure 1 illustrates this use case for two independent tasks A and B. Task A requires the resource from t0 to t2, while Task B requires the resource from t1 to t3. It is immediately evident that the effective lifetime of the resource will be from t0 to t2 since Task A will power it off when it is done using the resource. This behavior will result in a defect because the resource will be suddenly powered off while Task B is still utilizing it. It is important to note that it is completely valid for both tasks to simultaneously use the resource in this case.

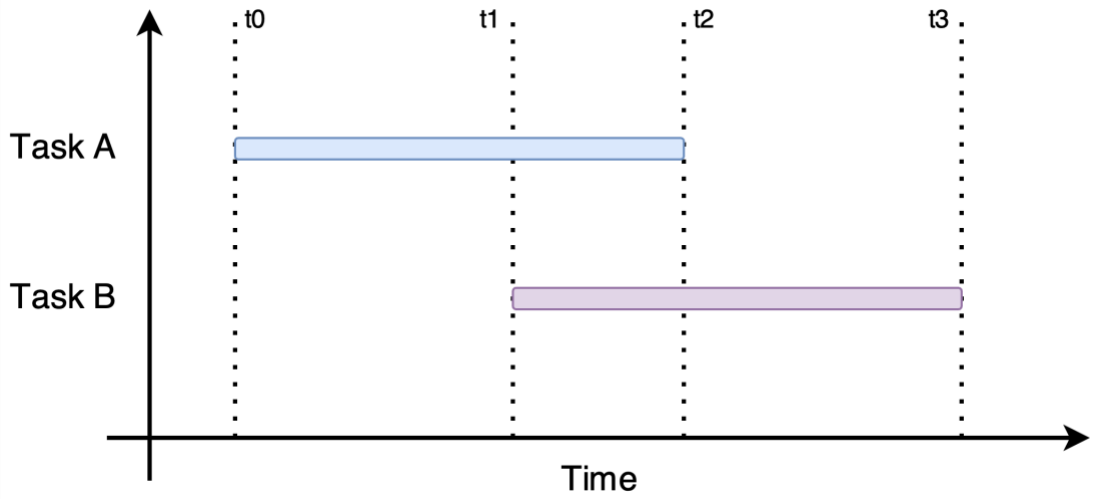

Figure 1 - Utilization of a shared resource by two independent tasks

The second issue is related to misconfiguration of a resource that should only be utilized exclusively by a single task. For example, independent tasks C and D might both require the use of a Software Defined Radio (SDR) to achieve their respective goals. Since both tasks are unaware of each other, they might try to configure the SDR in a conflicting way. For instance, Task A might want to set the carrier frequency to a specific value for transmitting to a device, while Task B might try to set the carrier frequency to a different value for receiving an IQ stream for analysis. At best, only one of these tasks succeeds while the other silently fails due to the misconfiguration, however, at the

other extreme, permanent damage could be caused due to misconfiguration of the RF hardware. As such, a mechanism should be put in place to prevent independent tasks from utilizing a resource in conflict.

3. Initial Design

A resource management framework was created to address the primary concerns presented in Section 2. The framework provides a public interface for clients and tasks to access satellite resources. We formally define the following terms to provide a consistent ground for discussion:

- ***Resource*** **–** a component, collection of components, or subsystem on the satellite that tasks require to perform their primary function. Resources can be grouped into two categories: *physical* and *virtual*. A physical resource is one that represents actual hardware. Some examples of physical resources are a payload computer, a network switch, a reaction wheel assembly, an SDR, etc. A virtual resource is, as the name implies, not representative of actual hardware, but rather, can be used to convey the idea of a logical checkpoint, or an abstract collection of physical resources. For example, while a payload computer might be a physical resource, we could define the CPU of that computer as a virtual resource to convey the idea that a resource intensive task might want to exclusively utilize the CPU, while also allowing other tasks to use the computer for other purposes.
- ***Acquire*** **–** the act of a client requesting permission from the resource management framework to utilize a particular resource for a requested duration.
- ***Release*** **–** the act of a client indicating to the resource management framework that it no longer requires a resource that it had previously required.
- ***Ownership*** **–** when a resource is acquired or released by a client, it transitions between three different "ownership" modes: exclusive ownership, shared ownership, and no ownership.

The core of the resource management framework is the "ownership model". This model is heavily inspired by C++ smart pointers (std::unique_ptr and std::shared_ptr), and is responsible for ensuring safe, and consistent access to all managed satellite resources. As discussed above, there are three different ownership modes. When a resource is not owned, it can be acquired in shared or exclusive mode by any client. When a resource is in shared ownership mode, it can be acquired by any number of clients simultaneously. In shared ownership mode, the underlying resource will only be powered on (if it is a physical resource) for the first acquisition. All subsequent acquisitions will simply be added to an internal ownership list for the resource and will immediately return to the caller. Only when all owners have released the resource will it be powered off. Referencing the scenario illustrated in Figure 1, had the two tasks utilized the resource management framework, Task A's request would have powered on the resource, while Task B's request would have been added to the ownership list. Thus, when Task A released the resource at t2, it would remain active until Task B released the resource at t3 (i.e. when all owners had released the resource). When a resource is in exclusive ownership mode, it cannot be acquired by any other clients until the current owner releases the resource. Table 1 describes the ownership model in more detail by illustrating how ownership requests will be handled depending on the current state of a resource.

Table 1 - a tabular view of the resource ownership model. Each row indicates the current ownership state of a resource, while each column maps to an ownership request by a client. The contents of the cells indicate if the request will be accepted or denied. For example, if the current ownership of a resource is "shared", then only shared acquisition requests will be accepted.

		Requested Ownership	
		Shared	*Exclusive*
Current Ownership	*None*	Accept	Accept
	Shared	Accept	Deny
	Exclusive	Deny	Deny

In addition to the underlying resource model, all acquisition requests must include a timeout duration which is intended to provide an estimate for the duration that the requested resource will be used for. If the client hasn't intentionally released the acquired resource before the timeout expires, the resource management framework will forcefully release the resource for the client. If the acquiring client does release the resource before the timeout has expired, the resource management framework cancels the timeout. This is particularly useful as it helps to manage resource lifetimes and prevents resources from being acquired indefinitely in the situation where a client process dies for an unexpected reason (error, cancellation, etc.). The resource management framework can also impose maximum durations for timeouts. This is particularly useful for high current draw components that could be extremely damaging to the satellite if left on for long periods of time.

4. Implementation

The resource management framework is implemented as a microservice called the resource manager. The Rust programming language was chosen for this project due to its excellent tooling, performance, and guarantees of memory safety, especially with respect to concurrency. While thread safety is often a "best effort" task in other languages, it is a first-class citizen of the Rust programming language. For example, mutexes in languages like C and C++ exist as independent objects that need to be associated during lock/unlock with the protected data. On the other hand, Rust mutexes take ownership of data. The only way to access the protected data is by locking the wrapping mutex. While this type of pattern can be implemented in other languages, it is not the standard. Since the resource manager is a concurrent application, mutexes are utilized to ensure that a resource can only be accessed by a single client at a time. Since the mutex owns the underlying resource object, other code cannot act on the resource without first locking the mutex.

The resource manager utilizes the ZeroMQ (ZMQ) messaging framework for inter-process communication (IPC). To achieve a high degree of concurrency, the resource manager implements the "asynchronous client/server pattern" by exposing a "router" socket publicly to clients on the satellite over the TCP transport. This socket forwards "jobs" to a series of worker threads that ultimately send results back to the client through the router socket. Figure 2 depicts this messaging pattern in greater detail.

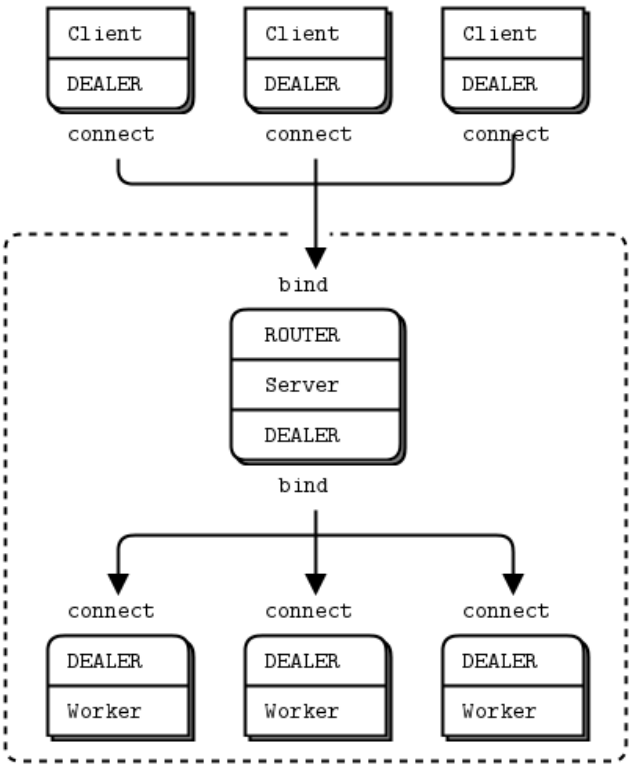

Figure 2 - Details of the asynchronous client/server messaging pattern *[1]*

Requests are sent to the resource manager as encoded protocol buffer (protobuf) messages. Protobufs are a language agnostic mechanism for serializing/deserializing structured data to/from a binary data format. This

technology was chosen due to wide adoption across multiple programming languages, the efficient binary format, the ability to define a message once and share that with any team that might need to use it. Listing 1 shows a protobuf message definition for the AcquireResource message. This message definition can be used to generate code definitions for several languages.

```
1  message AcquireResource {
2     string requester_id = 1;
3     ResourceId resource_id = 2;
4     OwnershipType ownership_type = 3;
5     uint32 timeout = 4;
6  }
```

Listing 1 - a sample protobuf message definition for acquiring a resource

These components make up the key aspects of the resource manager. To reiterate, clients make requests by serializing a protobuf message and sending it to the resource manager over a ZMQ socket. The resource manager handles these requests by spawning a worker thread that accesses the underlying resource management framework, locks the requested resource, and attempts to acquire the resource for the calling client. After the acquisition succeeds or fails, the result is propagated back to the client. Figure 3 provides a sequence diagram depicting a client executing an acquisition request to the resource manager.

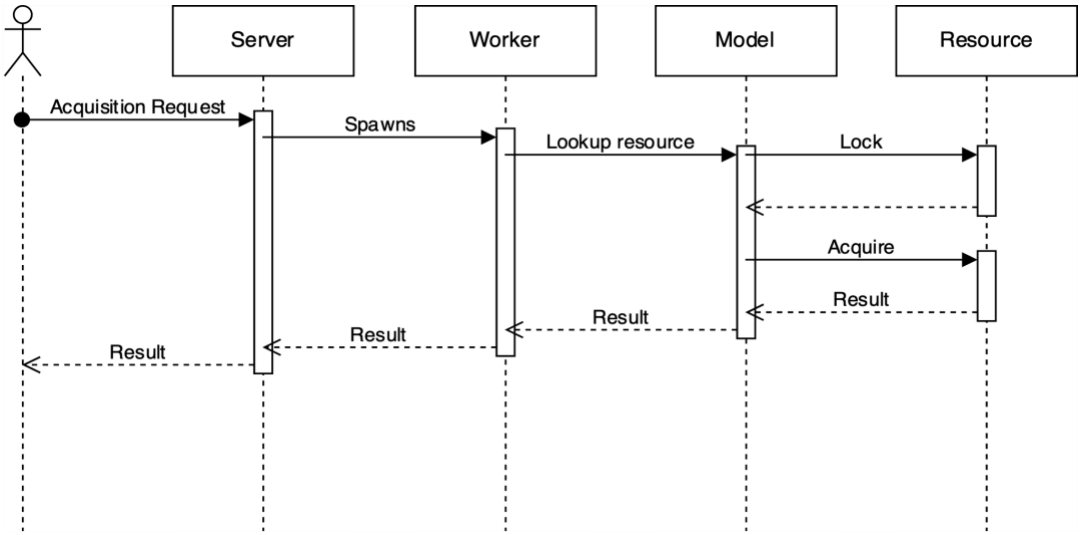

Figure 3 - A sequence diagram representing the request path for a client making an acquisition request for a resource.

5. Future Work

As designed, the resource manager already fully meets the initial requirements set out in Section 2. However, it has become quite cumbersome to maintain the same client code across multiple generations of satellites as the resource definitions can change frequently. This is especially true because Lynk strives to iterate their hardware designs at a rapid rate, often introducing new capabilities between launches. For example, a "payload" might require resources A, B, C, and D to be operated on satellite generation 1, but the same "payload" might require resources B, D, E, and F to be operated on satellite generation 2. Currently, client code would be required to account for these differences by using conditional logic to acquire a different set of resources for each generation of satellite. As the constellation grows and hardware becomes more diverse, this approach becomes unmanageable. Instead of placing this burden on clients, the resource manager could abstract these differences in hardware by exposing more abstract resource definitions to clients. Generally, a client only cares to "acquire the payload," and does not necessarily need a higher level of detail to know which resources comprise the payload. To achieve this, a "dependency" feature would be added to the resource model. Following the example, a virtual resource could be created to encapsulate the concept of the payload. This virtual resource could then have dependencies on the other physical resources that cumulatively make up the payload.

Figure 4 provides an example of such a dependency mechanism. In this situation, a client could acquire Resource A, and the resource manager would also acquire resources B, C, D, and E. The dependency model creates a Directed Acyclic Graph (DAG) between resources. Cycles must be forbidden as they would lead to deadlock when executing the resource model.

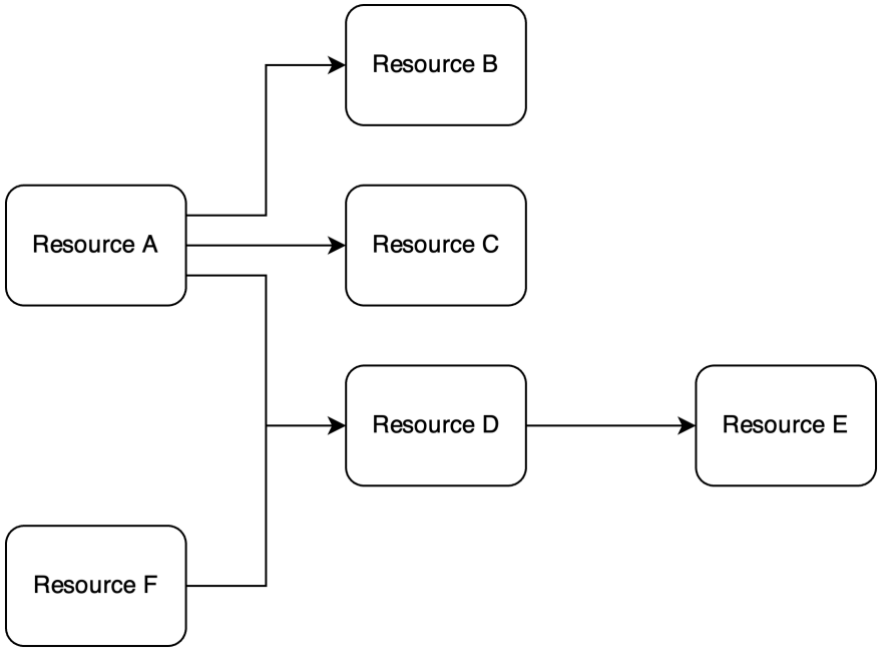

Figure 4 - a visual representation of resource dependencies. In this example, resource A directly depends on resources B, C, and D while transitively depending on resource E. Resource F depends directly on resource D, and transitively on resource E.

6. Conclusion

Autonomously executing independent tasks has caused challenges with effective resource management. The past method of utilizing low level interfaces for resource acquisition had led to undesired behavior in two primary use cases. First, there was no way to provide coordinated access to shared resources, leading to components being powered off before all tasks were finished using the resource. Second, there was no mechanism preventing multiple tasks from accessing a resource that should only be utilized by a single task at a time, leading to potential misconfigurations that could cause safety risks.

These issues were both fully addressed by the introduction of the on-board resource manager. Inspired by the concept of “smart pointers” in multiple programming languages, the resource manager enforces an ownership where resources can be acquired with shared or exclusive ownership. Independent tasks can now utilize the framework to share ownership of resources without worrying about coordination or having to concern themselves with managing the lifetime of the resource acquisitions. Similarly, the resource manager protects against tasks that fail unexpectedly by cleaning up active resources once the supplied timeout has expired. Tasks that require exclusive ownership of a resource can now confidently access that resource without worrying about other tasks potentially misconfiguring the resource. The resource manager will deny all other ownership requests for an exclusively owned resource. In the future, a dependency mechanism can be added to the resource model to abstract the hardware implementation of subsystems even further. This will have the benefit of simplifying all future client code as satellite hardware evolves.

The resource manager has significantly improved the operational consistency of Lynk satellites since its initial deployment. It has eliminated all resource lifetime issues for shared resources, and all misconfiguration issues for exclusively owned resources. Tasks can be developed and executed independently via automations with the guarantee that resource acquisitions will be safe and coordinated.

References

[1] P. Hintjens, "ØMQ - The Guide," [Online]. Available: https://zguide.zeromq.org/.